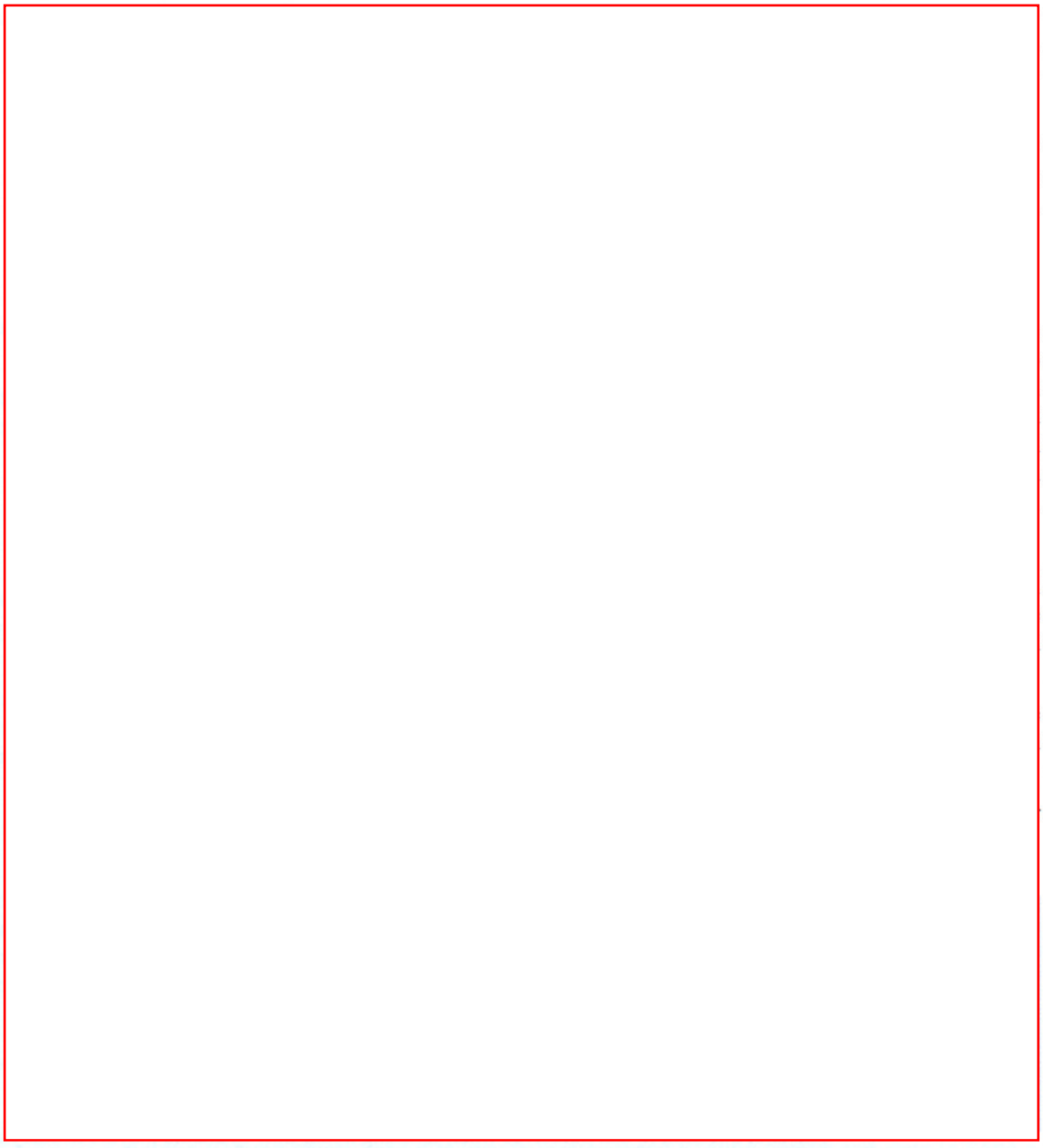

7.12.2 An example of refactoring

As an example, we shall continue with the extension of adding items to the game. In Section 7.11.2 we started adding items, suggesting a structure in which rooms can contain any number of items. A logical extension to this arrangement is that a player

should be able to pick up items and carry them around. Here is an informal specification of our next goal:

- The player can pick up items from the current room.
- The player can carry any number of items, but only up to a maximum weight.
- Some items cannot be picked up.
- The player can drop items in the current room.

To achieve these goals, we can do the following:

- If not already done, we add a class `Item` to the project. An item has, as discussed above, a description (a string) and a weight (an integer).
- We should also add a field `name` to the `Item` class. This will allow us to refer to the item with a shorter name than the description. If, for instance, there is a book in the current room, the field values of this item might be:

  name: `book`
  description: `an old, dusty book bound in gray leather`
  weight: `1200`

  If we enter a room, we can print out the item's description to tell the player what is there. But, for commands, the name will be easier to use. For instance, the player might then type *take book* to pick up the book.
- We can ensure that some items cannot be picked up, by just making them very heavy (more than a player can carry). Or should we have another `boolean` field `canBePickedUp`? Which do you think is the better design? Does it matter? Try answering this by thinking about what future changes might be made to the game.
- We add commands *take* and *drop* to pick up and drop items. Both commands have an item name as a second word.
- Somewhere, we have to add a field (holding some form of collection) to store the items currently carried by the player. We also have to add a field with the maximum weight the player can carry, so that we can check it each time we try to pick up something. Where should these go? Once again, think about future extensions to help you make the decision.

This last task is what we will discuss in more detail now, in order to illustrate the process of refactoring.

The first question to ask ourselves when thinking about how to enable players to carry items is: Where should we add the fields for the currently carried items and the maximum weight? A quick look over the existing classes shows that the `Game` class is really the only place where it can be fitted in. It cannot be stored in `Room`, `Item` or `Command`, since there are many different instances of these classes over time, which are not all always accessible. It does not make sense in `Parser` or `CommandWords` either.

Reinforcing the decision to place these changes in the Game class is the fact that it already stores the current room (information about where the player is right now), so adding the current items (information about what the player has) seems to fit with this quite well.

This approach could be made to work. It is, however, not a solution that is well designed. The Game class is fairly big already, and there is a good argument that it contains too much as it is. Adding even more does not make this better.

We should ask ourselves again which class or object this information should belong to. Thinking carefully about the type of information we are adding here (carried items, maximum weight) we realize that this is information about a *player!* The logical thing to do (following responsibility-driven design guidelines) is to create a Player class. We can then add these fields to the Player class and create a Player object at the start of the game to store the data.

The existing field currentRoom also stores information about the player: the player's current location. Consequently, we should now also move this field into the Player class.

Analyzing it now, it is obvious that this design better fits the principle of responsibility-driven design. Who should be responsible for storing information about the player? The Player class, of course.

In the original version we had only a single piece of information for the player – the current room. Whether we should have had a Player class even back then is up for discussion. There are arguments both ways. It would have been nice design, so yes, maybe we should. But having a class with only a single field and no methods that do anything of significance might be regarded as overkill.

Sometimes there are gray areas like this where either decision is defensible. But after adding our new fields, the situation is quite clear. There is now a strong argument for a Player class. It would store the fields and have methods such as dropItem and pickUpItem (which can include the weight check and might return false if we cannot carry it).

What we did when we introduced the Player class and moved the currentRoom field from Game into Player was refactoring. We have restructured the way we represent our data to achieve a better design under changed requirements.

Programmers not as well trained as us (or just being lazy) might have left the currentRoom field where it was, seeing that the program worked as it was and there did not seem to be a great need to make this change. They would end up with a messy class design.

The effect of making the change can be seen if we think one step further ahead. Assume we now want to extend the game to allow for multiple players.

With our nice new design, this is suddenly very easy. We already have a Player class (the Game holds a Player object), and it is easy to create several Player objects and store in Game a collection of players instead of a single player. Each player object would hold its own current room, items, and maximum weight. Different players could even have different maximum weights, opening up the even wider concept of having players with quite different capabilities – their carrying capability being just one of possibly many.

The lazy programmer who left `currentRoom` in the `Game` class, however, has a serious problem now. Since the whole game has only a single current room, current locations of multiple players cannot be easily stored. Bad design usually bites back later to create more work for us in the end.

Doing good refactoring is as much about thinking in a certain mindset as it is about technical skills. While we make changes and extensions to applications, we should regularly question whether an original class design still represents the best solution. As the functionality changes, arguments for or against certain designs change. What was a good design for a simple application might not be good any more when some extensions are added.

Recognizing these changes and actually making the refactoring modifications to the source code usually saves a lot of time and effort in the end. The earlier we clean up our design, the more work we usually save.

We should be prepared to *factor out* methods (turn a sequence of statements from the body of an existing method into a new, independent method) and classes (take parts of a class and create a new class from it). Considering refactoring regularly keeps our class design clean and saves work in the end. Of course, one of the things that will actually mean that refactoring makes life harder in the long run is if we fail to test adequately the refactored version against the original version. Whenever we embark on a major refactoring task it is essential to ensure that our existing test coverage is adequate beforehand, and that it is kept up to date through the refactoring process. Bear this in mind as you attempt the following exercises.