COMP 110

Prasun Dewan[1]

5. Style

In last chapter, we learnt how to implement a state-based spreadsheet. In this chapter, we focus on new ways of developing a spreadsheet, rather than learn how to write a new class of applications. Thus, the concepts we learn here have more do with the *style* of the program, that is, how it is written, rather than what function it implements.

We will continue with our spreadsheet example, and show an alternative way of implementing it. We will then introduce the notion of an interface as a way of uniting multiple solutions to a problem and separating the notion of specification from implementation. In English, the word interface means a boundary between two surfaces. In Java, this word also denotes a boundary, only not between two surfaces but between a class and its users. The spreadsheet example will also show that the reason for creating multiple competitive implementations is that they are efficient in different respects. In particular, some take more space to execute user commands while others take more time.

We will look at several additional concepts and principles for improving the style of a program. We will get further practice with named constants and removing code duplication, learn how to comment a program, and study a principle that determines what rights a piece of code has.

Alternative Implementations

Figure 1 outlines the approach we took in `ABMISpreadsheet` to implement the spreadsheet user-interface. Figure 2 illustrates another approach to solve the BMI spreadsheet problem. In this solution, we:

1. Declare *three* instance variables, `weight` and `height` as before, and an additional variable, `bmi`, that stores the BMI value.
2. As before, provide two setter methods, which assign new weight and height values to the variables `weight` and `height`, respectively, and in addition, compute and assign the new BMI value to the variable, `bmi`.
3. Provide *three* getter methods that return the values of the three instance variables.

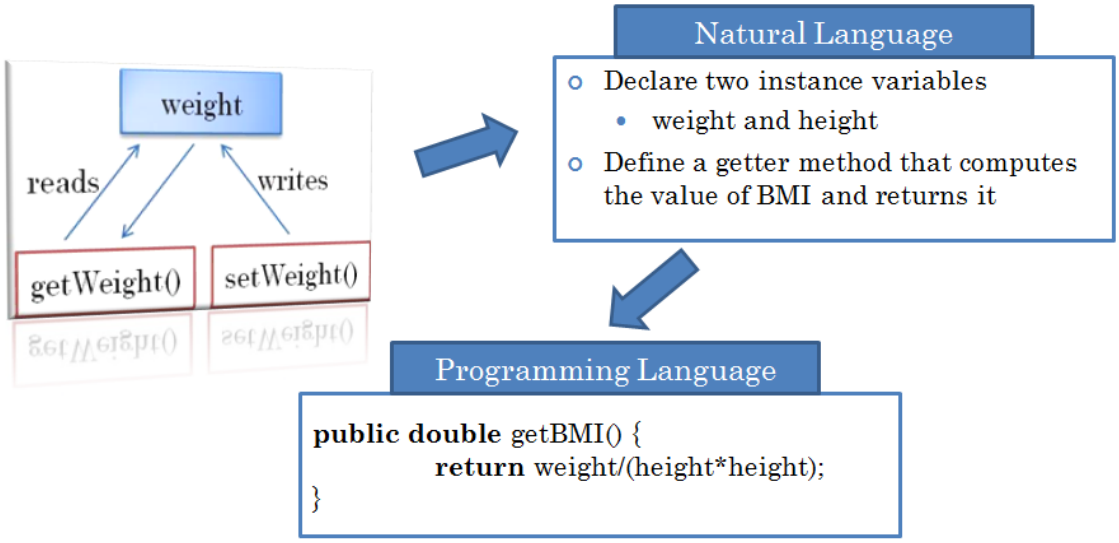

Figure 1. Original solution to `ABMISpreadsheet`

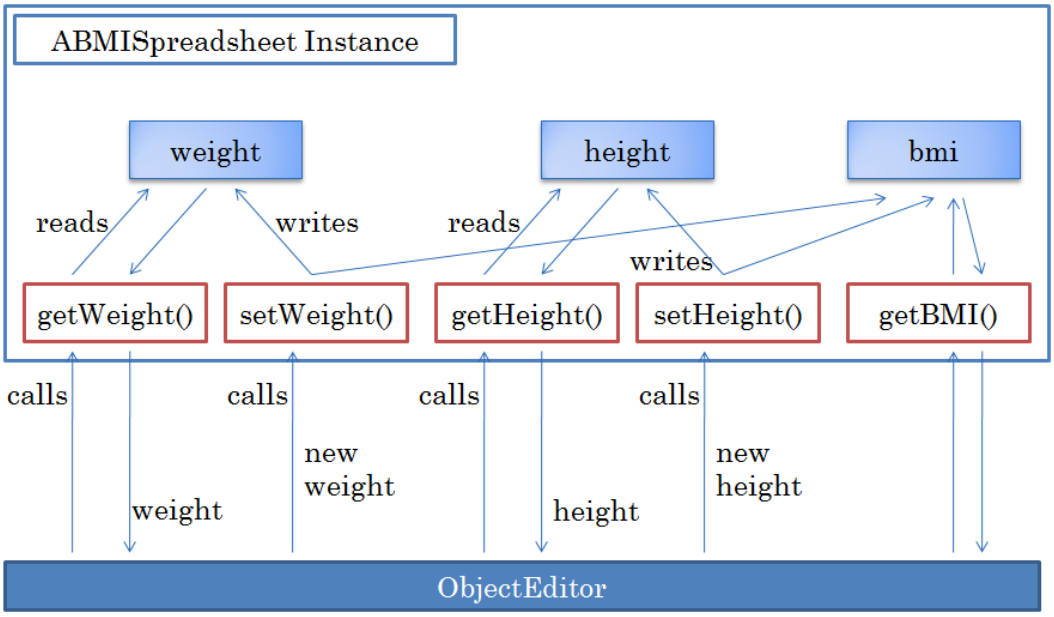

Figure 2. Solution to `ABMISpreadsheet` with three instance variables

Thus, the difference between this and the previous solution is that we create an instance variable also for the dependent property. The getter method for this property now simply returns the value of this variable, instead of calculating its value from the two properties on which it depends. Whenever a setter method changes one of these properties, it calculates the new value of the dependent property, and stores it in the corresponding instance variable, `bmi`. The getter method for BMI, thus, is guaranteed to return the correct value of the property the next time it is called.

The class, `AnotherBMISpreadsheet`, shown below, is a coding of this algorithm:

```
public class AnotherBMISpreadsheet {
      double height, weight, bmi;
      public double getHeight() {
            return height;
      }
      public void setHeight(double newHeight) {
            height = newHeight;
            bmi = weight/(height*height);
      }
      public double getWeight() {
            return weight;
      }
      public void setWeight(double newWeight) {
            weight = newWeight;
            bmi = weight/(height*height);
      }
      public double getBMI() {
            return bmi;
      }
}
```

Specifying Classes Through Interfaces

Thus, we now have different implementations of a BMI spreadsheet whose functionality is identical. As shown in Figure 3, if we compare the edit windows ObjectEditor creates for instances of the two classes, we cannot tell the difference between the two. This is the case despite the fact that second class contains an additional variable, `bmi`.

The reason the two classes look identical from the outside is that that the same set of public methods can be invoked on instances of the two classes.[2] This situation corresponds to two different factories producing the same kind of cars, that is, cars on which identical operations can be invoked.

Two factories, of course, do not accidentally produce the same cars, they are designed to do so, by carefully defining a car specification and ensuring that the factories follow the specification (Figure 4). To follow a similar process when declaring classes, we would like a way of specifying the functionality implemented by them.

[2] They also look the same from the outside because they implement the same properties. The set of methods is used to compare classes rather than the set of properties, for two reasons: First, properties are a Beans concept, not a Java concept. Second, we may wish to compare all methods of two classes, not just the getter and setter methods.

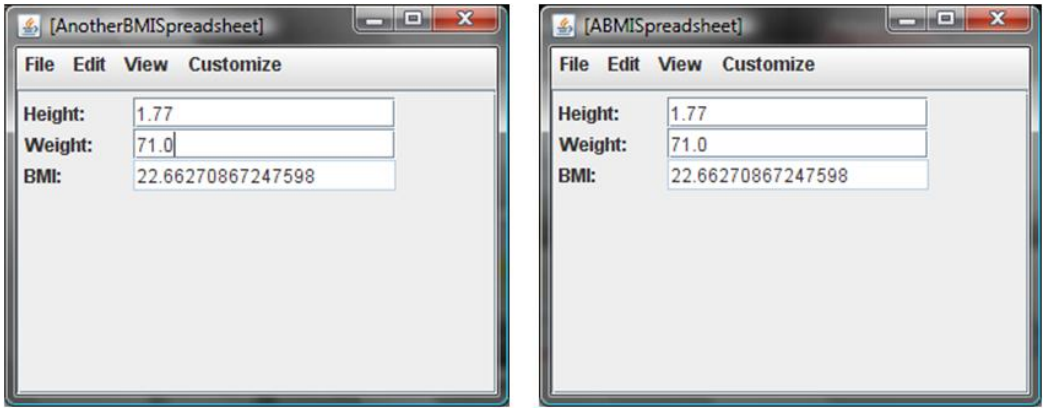

Figure 3. User-interfaces of `AnotherBMISpreadsheet` **and** `ABMISpreadsheet`

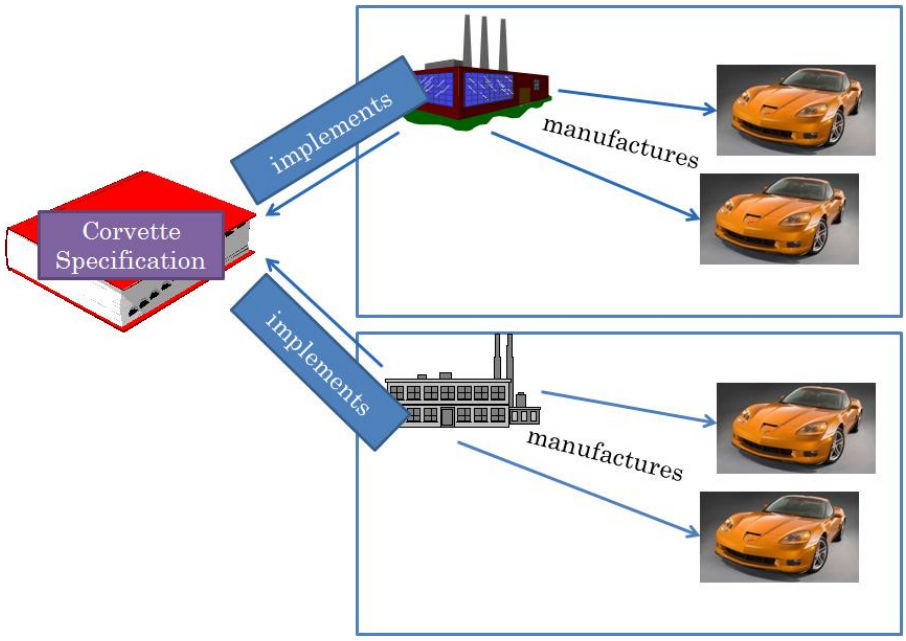

Figure 4. Solution to `ABMISpreadsheet` **with three instance variables**

In Java such specification takes the form of an *interface*. The specification of the two BMI Spreadsheet classes shown in Figure 5 illustrates the nature of an interface. The keyword **interface** tells us that we are declaring an interface. An interface declaration documents the headers of public methods of the classes it specifies. It does not contain the bodies of these methods since those are considered implementation details that different classes can choose differently. It does not contain non-public methods (such as calculateBMI method we will add later to this class) as they are not visible to users of the class, and thus make no difference to them. For similar reasons it does not contain declarations of instance variables. It can, however, declare named constants, which are accessible in all implementations of it, and can ensure that these implementations use the same values for them.

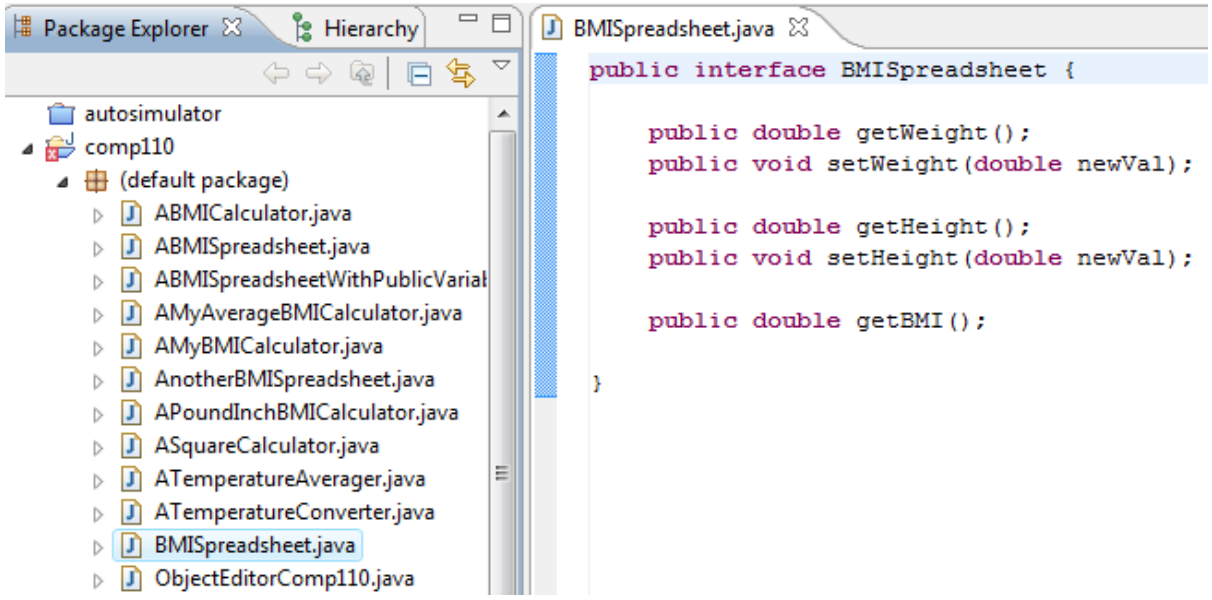

Figure 5. `BMISpreadsheet` **interface**

Once an interface has been defined, classes that intend to implement this specification can declare so in their headers:

```
public class ABMISpreadsheet implements BMISpreadsheet  { … }

public class AnotherBMISpreadsheet implements BMISpreadsheet  { … }
```

If a class header says that it implements an interface, then the Java compiler checks that the class actually implements each method whose header is specified in the interface. Thus, if we comment out the method `getBMI` in `ABMISpreadsheet`, the Java compiler will flag an error. This corresponds to a factory inspector ensuring that the factory actually implements all car features specified in the car specification. On the other hand, if we comment out the `getWeight` header from the interface, Java will not flag an error, even though the class implements this method. This corresponds to a factory not advertising all the features it provides, which is not a good idea. Therefore, it is our responsibility to make sure that the headers of all of the public methods of the class are in the interface(s) it implements.[3]

As in the case of a class, the name of the file in which an interface is saved is the interface name followed by the suffix .java. As shown in Figure 5, we have included the interface also in the `bmi` project, since it is related to the other two items in the project, specifying their methods.

We have motivated an interface above as a mechanism for uniting two classes that implement the same set of public methods. In fact, it has other several other important uses. Someone trying to understand

[3] A class can implement more than one interface, just as a factory can produce more than one kind of product.

what a class does can just look at the interface it implements, rather than look at the complete class. It also breaks down the process of defining a class into two steps: first define the interface, and then the class itself. Thus, it supports another important form of stepwise refinement. For these reasons, in the future, we will define the interface of each class we implement; and will insist that you do so also.

If a class implements an interface, we will refer to an instance of the class as also an instance of the interface, because like the class, the interface describes the operations that can be performed on the instance. Thus, we will refer to instances of `ABMISpreadsheet` and `AnotherBMISpreadsheet` as also instances of `BMISpreadsheet`. This usage is consistent with labeling a car manufactured according to the Accord specification at the Ohio Accord factory as both an Ohio Factory Accord and also simply as an Accord.

Naming an Interface

We should relate the name of an interface with the names of the classes that implement it. In the two examples above, the name of a class that implements an `<Interface>` has the form:

```
<Qualifier><Interface>
```

We can distinguish between among implementations of an interface by using different qualifiers for the implementations such as: `ABMISpreadsheet`, `AnotherBMISpreadsheet`, `AThreeVariableBMISpreadsheet`, `ATwoVariableBMISpreadsheet`, and so on.

A more popular convention for naming the class is:

```
<Interface><Qualifier>Impl
```

Examples of the use of this convention would be `BMISpreadsheetImpl`, `BMISpreadsheetAnotherImpl`, and `BMISpreadsheetAThreeVariableImpl`. The second approach requires more typing and is a bit awkward to read but has the advantage that in an alphabetical listing of classes and interfaces, the name of classes appear next to the interfaces they implement. In a small project such as the one above, this is not an issue. You should choose whichever convention seems more natural to you. In this book, we will continue to follow the former approach because we will be creating small projects.

Some create an interface name based on a class that implements it. Usually the interface name has the form:

```
<Class>Interface
```

Thus, under this approach, the interface implemented by `ABMISpreadsheet` would be called:

```
ABMISpreadsheetInterface
```

This approach, however, ties the interface name to a particular class. It does not tell us how to relate the name of another implementation of the interface to the name of the interface or the original class. It

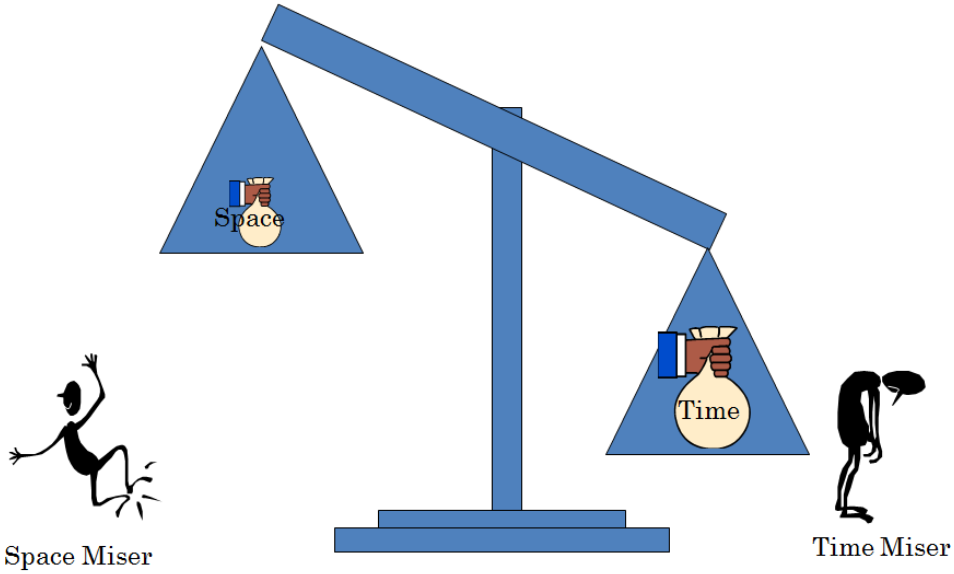

Figure 6. `BMISpreadsheet` **interface**

works as long as an interface has a single implementation. It is short-sighted, however, to expect a single implementation of an interface. Thus, this naming scheme is not recommended.

Space & Time Efficiency

Why create multiple implementations of the same interface? Why create multiple factories that implement the same car specification? In the latter case, the answer often has to do with the fact there are limitations on how many cars can be produced by a factory, extra factories are needed because the total demand cannot be met by a particular factory. There is no analogue of this situation in the class case, however, since there are no practical limitations on the number of instances of instances we can create from a class.[4]

The answer to our question lies somewhat in the second reason for creating an additional factory, which has to do with cost. Sometimes we create a factory in some location because it is more efficient to do so. Efficiency is an important reason also for creating multiple classes that implement the same interface.

Let us compare the efficiency of the two implementations of `BMISpreadsheet`. The first implementation takes less *space* in memory in that it creates one less instance variable for each object. On the other hand, the second one often takes less *time*, that is, executes faster. The reason is that in the first case we calculate the value of BMI each time it is required, whereas in the second case, we calculate it only when it changes. Consider what happens when we ask ObjectEditor to do successive

4 They must all fit in memory, of course. This is equivalent to finding parking space for all cars produced, which is independent of how many factories produce them.

refreshes without changing any property. Recall that each refresh results in the invocation of all getter methods of the object displayed by ObjectEditor. In the first case, the same BMI value will be calculated multiple times, on each invocation of `getBMI` that results from a refresh. In the second case, on the other hand, the BMI value will be calculated only when the weight or height changes, not on all of the subsequent refreshes that happen afterwards.

An implementation that takes less space is called more *space efficient*, and one that takes less time is called more *time efficient*. What we see here is an example of a classic tradeoff in programming, called the *space-time tradeoff*: more space efficiency often implies less time efficiency. For this reason, we often create multiple solutions to a problem that make this tradeoff differently, and allow the users to determine which solution suits their needs.

The efficiency concerns here are fundamentally different from the ones in the factory case. In the latter, efficiency had to do with the manufacturing process, not with the cars produced. Here they have to with the instances created by the classes: we are interested in the space and time taken by an instance, not the class. This corresponds to a car produced by one factory being faster or bulkier than one produced by another, which would unacceptable if they are considered the same car. Such a situation is acceptable in our case because it is the only way to address the space-time tradeoff.

Syntactic vs. Semantic Specification

While the two example classes above offer identical functionality, not all implementations of an interface are guaranteed to do so. For instance, a class that "implements" this interface may provide the following erroneous implementation of `getBMI`:

```
public double getBMI() {
      return weight*weight/height;
}
```

In general, we need to give more information than method headers and constants when specifying a class. Ideally, we should be able to specify the exact semantics of each of the operations and Java should be able to check that an implementation conforms to these semantics. However, as it turns out, programs are so complex, that is impossible to do so in general.[5] An interface only specifies the syntactic details about the operations. It cannot specify the semantics of these operations. For this reason, cannot ensure that the `getBMI` method returns the correct body mass index, and not, say, the Bombay Market Index. We will rely on identifier names, comments (discussed below), and a shared understanding of certain concepts (such as BMI) to specify the semantics of an operation. Java will ensure that all method headers defined in an interface have bodies in the implementation. There is nothing to prevent differences in the behavior of different implementations of these headers. The advantages of interfaces are based on the assumption that they are implemented as "expected."

[5] Java cannot even tell if a program will halt!

Comments

The code in `AnotherBMISpreadsheet` is fairly self-explanatory. However, to clear up any possible confusion a reader of it may have, we have added *comments* to it. Returning to the theater analogy, think of comments as notes left by scriptwriters (in a language of their choice) for themselves (so that they can understand it later), other scriptwriters and reviewers, but not for the performers. In fact, the compiler removes them before creating object code – so the CPU never sees them and they do not occupy memory space.

There are two kinds of comments in Java: *single-line* and *arbitrary*. A single-line comment begins after the comment marker `//` and spans the rest of the line, as shown below:

```
double bmi; //computed by setWeight and setHeight
```

An arbitrary comment is written between the comment markers `/*` and `*/` and can span either one line:

```
/* recompute dependent properties */
bmi = weight / (height*height);
```

or multiple lines:

```
/* This version recalculates the bmi
   when weight or height change, not when
   getBMI is called
*/
public class AnotherBMISpreadsheet {…}
```

Javadoc Conventions

While the above style of commenting for multiple lines is legal in Java, the following style illustrates the convention:

```
/* This version recalculates the bmi
 * when weight or height change, not when
 * getBMI is called
 */
public class AnotherBMISpreadsheet {…}
```

If we use this convention, a Java tool called `javadoc` will automatically extract the comments from the class, allowing a reader trying to understand it to focus on them while ignoring the coding details. In general, it is a good idea to insert the algorithm coded by a class as comments in the class. If we do so, `javadoc` will essentially extract the algorithm, allowing a reader to focus only on the algorithm. You should use this convention, even though it does require you to make the extra effort of ensuring that `*`'s are aligned.

Commenting out Debugging Code

Comments are useful not only for documentation, but also commenting out debugging code. To illustrate what this means, consider the following comment in `AnotherBMISpreadsheet`:

```
/*
      System.out.println(newHeight); // debugging statement
*/
```

Here, we commented out the print statement once we finished debugging. By enclosing the code within comments, we ensure that it will not actually be seen by the CPU. Commenting out code is better than deleting it, because if we later need to debug it, all we have to do is remove the comment markers rather than retype the code.

Single-line vs. Arbitrary Comments

Why support both kinds of comments? Single-line comments save us from typing an extra end-comment marker for a short comment. Arbitrary comments, on the other hand, are necessary for long comments. Moreover, we need two kinds of comments to allow us to "comment out'' code that has comments. If we had only one kind of comment, say the arbitrary comment, we might have tried to comment out the print statement as follows:

```
/*
      System.out.println(newHeight);  /*debugging statement */
*/
```

Unfortunately, this does not work. When the compiler sees the beginning of a comment, it ignores everything until it finds the end of the comment. Therefore, in this example, it will ignore the start of the inner comment and assume that the end of the inner comment is, in fact, the end of the outer comment. It will then flag the end of the outer comment as an error. With two types of comments, this problem does not arise.

What to Comment

In general, we should comment any code fragment that *needs explanation*, such as a:

- Class declaration – the comment should explain characteristics of the whole class such as its purpose, design goals, and the top-level algorithm implemented by it. It can also give the author of the class and the date(s) it was modified.
- Variable declaration – the comment should explain the purpose of the variable, how its value is computed, and where it is used.
- Method declaration – the comment should explain the algorithm implemented by the method, and its parameters and return values. It should also give the author of the method and the date(s) it was modified, if more than one person has written the class in which the method is defined.

- Complicated statement sequence within a method – the comment would explain what the statement sequence achieves.

Comments that elaborate on a piece of code come after the code, while those that summarize it to tend to come before it. Comments about a variable declaration tend to elaborate on the declaration, and thus, come after the declaration. On the other hand, comments about a class, method, or some complicated statement sequence summarize it, and therefore, come before it.

Long Identifiers vs. Comments

What we write in comments depends on the identifiers we choose. For instance, if we use `w` as the name of the instance variable storing the weight, we would need a comment to explain the role of the variable:

```
double w; // weight
```

However, if we name it `weight`, we do not need any further explanation:

```
double weight;
```

In fact, we should never make redundant comments that add no information to the identifier name:

```
double weight; // weight
```

Such redundant comments serve only to clutter the code.

Thus, with longer identifiers, we often reduce the need for comments about identifier declarations. We do not eliminate it, however, because not every aspect of an identifier can be explained by its name, as illustrated below:

```
double bmi; //computed by setWeight() and setHeight()
```

Javadoc Tags

Javadoc defines special kinds of comments regarding an author, parameter or return value of a method. A comment identifying an author is for the form:

```
@author <author name>
```

and a comment about a parameter is of the form:

```
@param <parameter name> <parameter description>
```

as illustrated below:

```
/*
 * @author Prasun Dewan
 * @param newWeight the new value of the property, weight.
```

```
 * sets new values of the variables, weight and bmi
 */
public void setWeight (double newWeight) {
      …
}
```

A comment about a return value is of the form:

```
@return <return value description>
```

as illustrated below:

```
/*
 * @author Prasun Dewan
 * @return the value of the variable, weight
 */
      public double getWeight () {
            …
}
```

Here, `@param`, `@return`, and `@author` are *javadoc tags,* which can be thought of as "reserved words." They are reserved, not by Java, but by Javadoc. They allow the tool to show only a certain kind of comments, such as only the comments about parameters and return values, thereby providing a useful form of filtering.

In the examples above, the comments are probably redundant because the commented methods are simple and we have been careful in giving names to parameters and methods that are *self-commenting*. In more complex examples, comments about a method can be very useful, allowing the reader to understand what the method does without looking at the code.

Commenting an Interface

Because an interface provides only syntactic information, it is crucial to comment it to provide the semantic details that are not obvious from the identifiers declared in it. For example, we might wish to clarify, in the declaration of `BMISpreadsheet`, that the BMI stands for the Body Mass Index. Interface comments can include an overall comment and comments about individual method headers. The overall comment would be similar to the overall comment for a class except that it would not describe any algorithm, which is implementation-dependent. Similarly, the comments about method headers in an interface would be similar to comments about complete methods in a class except, again, that they would not give an algorithm.

To illustrate, the difference between comments in an interface and those in a class, consider the comments we wrote for the `setWeight` method:

```
/*
 * @author Prasun Dewan
 * @param newWeight the new value of the property, weight.
```

```
 * sets new values of the variables, weight and bmi
 */
public void setWeight (double newWeight) {
      …
}
```

(Recall that the comment about the parameter is really redundant, and was given purely for illustration purposes.)

Now consider commenting header of this method in the interface. The author comment is probably redundant, because it is likely that all method headers were put in by the same person(s). If this is not the case, then it would indeed be useful. The variables, `weight` and `bmi` are implementation-specific. Thus these comments do not apply to the header. The only comment does apply is the one about the parameter:

```
 /*
  * @param newWeight the new value of the property, weight.
  */
public void setWeight(double newWeight);
```

Should this comment be repeated in both the interface and the class? Strictly speaking, the answer is no. While it useful to see this comment when looking at the method implementation, a tool can easily extract this comment from the interface and put it in the class. However, such a tool probably does not currently exist. Therefore, it would be useful to replicate the interface comment in the class until such a tool is developed.

Encapsulation

In each of the two classes implementing the BMI spreadsheet, the instance variables of the class could not be accessed directly by other classes such as ObjectEditor. They were accessed indirectly by other classes through the getter and setter methods of the class. In fact, we could have allowed direct access by other classes to these variables by declaring them as public:

```
public class ABMISpreadsheetgWithPublicVariables  {
      public double height, weight, bmi;
      …
}
```

Like the properties of an object, the public instance variables and named constants of the object are displayed by ObjectEditor. Moreover, the displays of the public instance variables can be edited by the user to change the values of the variables, as shown in the Figure 7.

However, instance variables of a class should not be declared as public, for two main reasons.

- *Maintaining consistency constraints:* Other classes can assign inconsistent values to public variables. For instance, as shown in the figure above, it is possible for other classes to assign to the bmi variable a value that is inconsistent with the values of the other two variables. By not

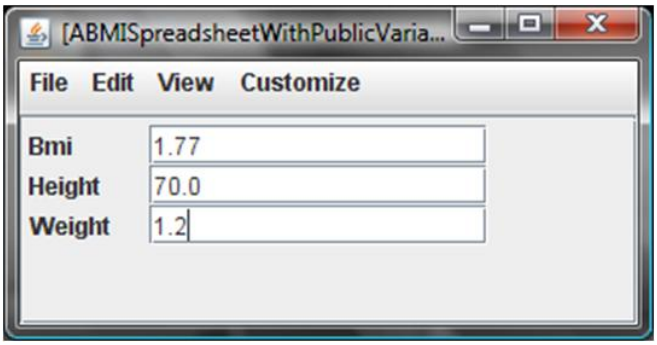

Figure 7. `BMISpreadsheet` **interface**

making the variables public, the class declaring them can ensure that its consistency constraints are not violated. This is akin to not allowing drivers of a car to directly access all aspects of the engine, for fear that they may damage it.

- *Ease of change*: Suppose that we decide to change the implementation of a class by changing its instance variables. We may have first created an implementation of the BMI spreadsheet using three variables. Later, as we port the program to handheld computers with small memories, we may realize that we need to save on space by using only two variables. Rather than create a new class, as we did above, we may wish to change the current class, so that other classes that use it do not change. If the current class does not make any of its variables public, it would indeed be possible to localize the changes only to it. If it does have public variables, then other classes might be depending on them. In this case, it would necessary to change all classes that access public instance variables that no longer exist. For instance, in our BMI example, it would be necessary to change all classes that access the `bmi` variable, which would not exist in the new implementation. By not declaring the variables of a class as public, it is possible to change the implementation of the class without affecting other classes as long as its public methods remain the same. This is akin to not publicizing the implementation of a car engine so that new manuals do not have to be written when the implementation changes.

A class that does not make its instance variables public is said to *encapsulate* them. In all of the examples you create, you should encapsulate instance variables for the reasons given above.

Avoid Code Repetition by Using Internal Methods

Consider the two setter methods in the class `AnotherBMISpreadsheet`. The code to calculate BMI:

```
weight/(height*height)
```

is repeated twice, once in each of the two setter methods. We can, of course, use the `calculateBMI` method in `ABMICalculator` instead of re-implementing the formula. However, we may want to have control over how the formula is calculated. In particular, we may want to change our mind about whether we use double or integer values for weight and whether we use the metric system or not. Thus, instead of using an existing method in another class, we can put the formula calculation in a new

method in this class, and call the method wherever we need the formula calculated. In this example, because the repeated code returns a value, we will put it in a separate function:

```
double calculateBMI () {
       return weight/(height*height);
}
```

Now we replace the two occurrences of:

```
bmi = weight/(height*height);
```

with two occurrences of:

```
bmi = calculateBMI();
```

Instead of repeating the body of the function, we now only repeat the call to it. Typically, the body of a function will be much larger than a call to it. Thus, we save on typing effort.

After using our example application, our clients may tell us that they would prefer to enter the weight and height in pounds and inches rather than kilograms and meters. In the previous version of `AnotherBMISpreadsheet`, we may change the `setWeight` method but not the `setHeight` method, leading to an erroneous program. On the other hand, in the new version, we would only have to change `calculateBMI`:

```
double calculateBMI() {
       return (weight/2.2)/(height * 2.54/100*height*2.54/100);
}
```

Thus, we see again the advantage of reusing code.

Unlike the other methods in this class, `calculateBMI` is not a getter or setter method. As a result, it is not automatically invoked by ObjectEditor. It is invoked explicitly by our program when we assign a new value to `bmi`:

```
bmi = calculateBMI();
```

After defining getter and setter methods, you may think that it is sufficient to just declare methods – Java somehow automatically calls them at the correct time. A method declaration simply tells Java that we have defined a new operation. Java does not know when it should execute this operation unless we tell it explicitly do so. This is akin to defining an accelerate operation in a car but not invoking it until a driver presses the accelerator. The reason why we did not have to explicitly call the getter and setter methods is that ObjectEditor (which is not part of Java) did that for us to support display and editing of the properties. A method such as `calculateBMI` that is not a getter or setter method must always be called explicitly by us – ObjectEditor never calls it automatically. Of course, we saw this idea before when we composed functions and also when we called the `println` procedure in our setters and getters.

Least Privilege / Need-to-know Basis / Information Hiding

The method `calculateBMI` is the first method whose header does not contain the keyword **public**. We have not made it accessible to other classes because we see no need for them to call it. It has been defined solely to satisfy an internal need of the class, not to add functionality accessible to other classes such as ObjectEditor. In general, a class will contain both public methods, accessible from outside the class, and non-public methods[6], accessible only to other methods in the same class. Only the public methods of a class can be invoked from an edit window created by ObjectEditor, because non-public methods are not visible to it. Moreover, only the headers of the public methods of a class appear in its interface.

By not making `calculateBMI` public, we are following the principle of least privilege, which says that a piece of code should be given the least rights or *privileges* it needs to do its job. ObjectEditor and other classes do not need to directly call `calculateBMI`, therefore, why give them the right to do so?

The main problem with unnecessary privileges is that they can accidentally hurt us, just as an accidentally loaded gun can. In this example, if the method were public, ObjectEditor would clutter the methods menu with an item for this method. Moreover, some class can accidentally call `calculateBMI` when it really intends to call `getBMI`.[7] Furthermore, a reader trying to understand how to use instances of `AnotherBMISpreadsheet` would be forced to unnecessarily look at `calculateBMI`.

Another problem with giving unnecessary privileges is that it becomes difficult to change code. Suppose, after writing `calculateBMI`, we decided to make it a procedure that assigns the computed BMI value to the `bmi` variable:

```
void calculateBMI() {
      bmi = height / (weight*weight);
}
```

After making this change, we would have to modify all calls to it, because it is now a procedure rather than a function. Thus instead a call of the form:

```
bmi = caculateBMI();
```

we would make the call:

```
calculateBMI();
```

---

[6] In fact, non-public methods and variables can be given three kinds of accesses called private, protected, and default, which allow some but not all classes to access them. In this book, we will not distinguish between these three kinds of accesses and assume that only the declaring class can access a non-public method or variable.

[7] These methods return the same value but the former is more inefficient, because unlike the latter, it calculates the value each time it is called, rather than returning the value of the `bmi` variable.

If the method were public, we would have to make such a change to all classes that refer to it. By not making it public, we restrict the changes to the class that defines it.

Recall that it was for the same reasons – to maintain consistency constraints and support ease of change – that we did not make instance variables public. We were following this principle then too, making sure the access to variables declared in a class was given only to methods declared in that class. This principle is also called the "*need to know*" principle because we ensure that code is not given more information than it needs to know. It is also called the *information hiding* principle, because we hide unnecessary information from code.

These problems of giving unnecessary privileges are fairly minor in this example. However, in general, several subtle problems, including security breaks, can arise when this principle is not followed.