So Hacker's way of testing—writing his own test class—was one step in the right direction, but was, flawed. We know now that his problem was that although his class contained reasonable method calls for testing, it did not include any assertions on the method results, and thus did not detect test failure. Using a dedicated unit test class can solve these problems.

Exercise 9.37 Open your project again and add better testing by replacing Hacker's test class with a unit test class attached to the `CalcEngine`. Add similar tests to those Hacker used (and any others you find useful), and include correct assertions.

9.13 Putting the techniques into practice

This chapter has described several techniques that can be used either to understand a new program or to test for errors in a program. The *bricks* project provides a chance for you to try out those techniques with a new scenario. The project contains part of an application for a company producing bricks. Bricks are delivered to customers on pallets (stacks of bricks). The `Pallet` class provides methods telling the height and weight of an individual pallet, according to the number of bricks on it.

Exercise 9.38 Open the *bricks* project. Test it. There are at least four errors in this project. See if you can find them and fix them. What techniques did you use to find the errors? Which technique was most useful?

9.14 Summary

When writing software, we should anticipate that it will contain logical errors. Therefore, it is essential to consider both testing and debugging to be normal activities within the overall development process. BlueJ is particularly good at supporting interactive unit testing of both methods and classes. We have also looked at some basic techniques for automating the testing process and performing simple debugging.

Writing good JUnit tests for our classes ensures that errors are detected early, and they give a good indication which part of the system an error originates in, making the resulting debugging task much easier.

Terms introduced in this chapter:

syntax error, logical error, testing, debugging, unit testing, JUnit, positive testing, negative testing, regression testing, manual walkthrough, call sequence

Concept summary

- **testing** Testing is the activity of finding out whether a piece of code (a method, class, or program) produces the intended behavior.
- **debugging** Debugging is the attempt to pinpoint and fix the source of an error.
- **unit testing** refers to tests of the individual parts of an application, such as methods and classes.
- **positive testing** Positive testing is the testing of cases that are expected to succeed.
- **negative testing** Negative testing is the testing of cases that are expected to fail.
- **test automation** simplifies the process of regression testing.
- **assertion** An assertion is an expression that states a condition that we expect to be true. If the condition is false, we say that the assertion fails. This indicates an error in our program.
- **fixture** A fixture is a set of objects in a defined state that serves as a basis for unit tests.
- **walkthrough** A walkthrough is an activity of working through a segment of code line by line while observing changes of state and other behavior of the application.