– How do they know the load limit on bridges, Dad?
– They drive bigger and bigger trucks over the bridge until it breaks.
Then they weigh the last truck and rebuild the bridge.

— BILL WATTERSON,
Calvin and Hobbes (1997)

Chapter 6

SCIENTIFIC DEBUGGING

Once we have reproduced and simplified the problem, we must understand how the failure came to be. The process of obtaining a theory that explains some aspect of the universe is known as *scientific method*. It is also the appropriate process for obtaining problem diagnostics. We introduce the basic techniques for creating and verifying hypotheses, for making experiments, for conducting the process in a systematic fashion, and for making the debugging process explicit.

6.1 HOW TO BECOME A DEBUGGING GURU

Some people are true *debugging gurus*. They look at the code and point their finger at the screen and tell you: "Did you try X?" You try X and, voilà!, the failure is gone. Such *intuition* comes from experience with earlier errors — one's own errors or other people's errors — and the more experience you have the easier it is to identify potential error causes and set up accurate hypotheses. Thus, the good news is that you too will eventually become a debugging guru — if you live long enough to suffer through all of the failures it takes to gather this experience.

We can speed up this process by training our reasoning. How can we systematically find out why a program fails? And how can we do so without vague concepts of "intuition," "sharp thinking," and so on? What we want is a method of finding an explanation for the failure — a method that:

- *does not require a priori knowledge* (that is, we need no experience from earlier errors)

- *works in a systematic and reproducible fashion* such that we can be sure to eventually find the cause and reproduce it at will.

The key question for this chapter is thus:

HOW DO WE SYSTEMATICALLY FIND AN EXPLANATION FOR A FAILURE?

6.2 THE SCIENTIFIC METHOD

If a program fails, this behavior is initially just as surprising and inexplicable as any newly discovered aspect of the universe. Having a program fail also means that our abstraction fails. We can no longer rely on our model of the program, but rather must explore the program independently from the model. In other words, we must approach the failing program as if it were a *natural phenomenon.*

In the natural sciences, there is an established method for developing or examining a theory that explains (and eventually predicts) such an aspect. It is called *scientific method* because it is supposed to summarize the way (natural) scientists work when establishing some theory about the universe. In this very general form, the scientific method proceeds roughly as follows.

1. Observe (or have someone else observe) some aspect of the universe.
2. Invent a tentative description, called a *hypothesis,* that is consistent with the observation.
3. Use the hypothesis to make *predictions.*
4. Test those predictions by *experiments* or further *observations* and modify the hypothesis in the light of your results.
5. Repeat steps 3 and 4 until there are no discrepancies between hypothesis and experiment and/or observation.

When all discrepancies are gone, the hypothesis becomes a *theory.* In popular usage, a theory is just a synonym for a vague guess. For an experimental scientist, though, a theory is a conceptual framework that explains earlier observations and predicts future observations — such as relativity theory or plate tectonics, for instance.

In our context, we do not need the scientific method in its full glory, nor do we want to end up with grand unified theories for everything. We should be perfectly happy if we have a specific instance for finding the causes of program failures. In this debugging context, the scientific method operates as follows.

1. Observe a failure (i.e., as described in the problem description).
2. Invent a *hypothesis* as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make *predictions.*
4. Test the hypothesis by *experiments* and further *observations*:
   - If the experiment satisfies the predictions, refine the hypothesis.
   - If the experiment does not satisfy the predictions, create an alternate hypothesis.
5. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

The entire process is illustrated in Figure 6.1. Again, what you eventually get is a theory about how the failure came to be:

- It explains earlier observations (including the failure).
- It predicts future observations (for instance, that the failure no longer appears after applying a fix).

In our context, such a theory is called a *diagnosis.*

6.3 APPLYING THE SCIENTIFIC METHOD

How is the scientific method used in practice? As an example in this chapter, consider the `sample` program as discussed in Chapter 1 "How Failures Come to Be." The `sample` program is supposed to sort its command-line arguments, but some defect causes it to fail under certain circumstances:

```
$ sample 11 14
Output: 0 11
$ _
```

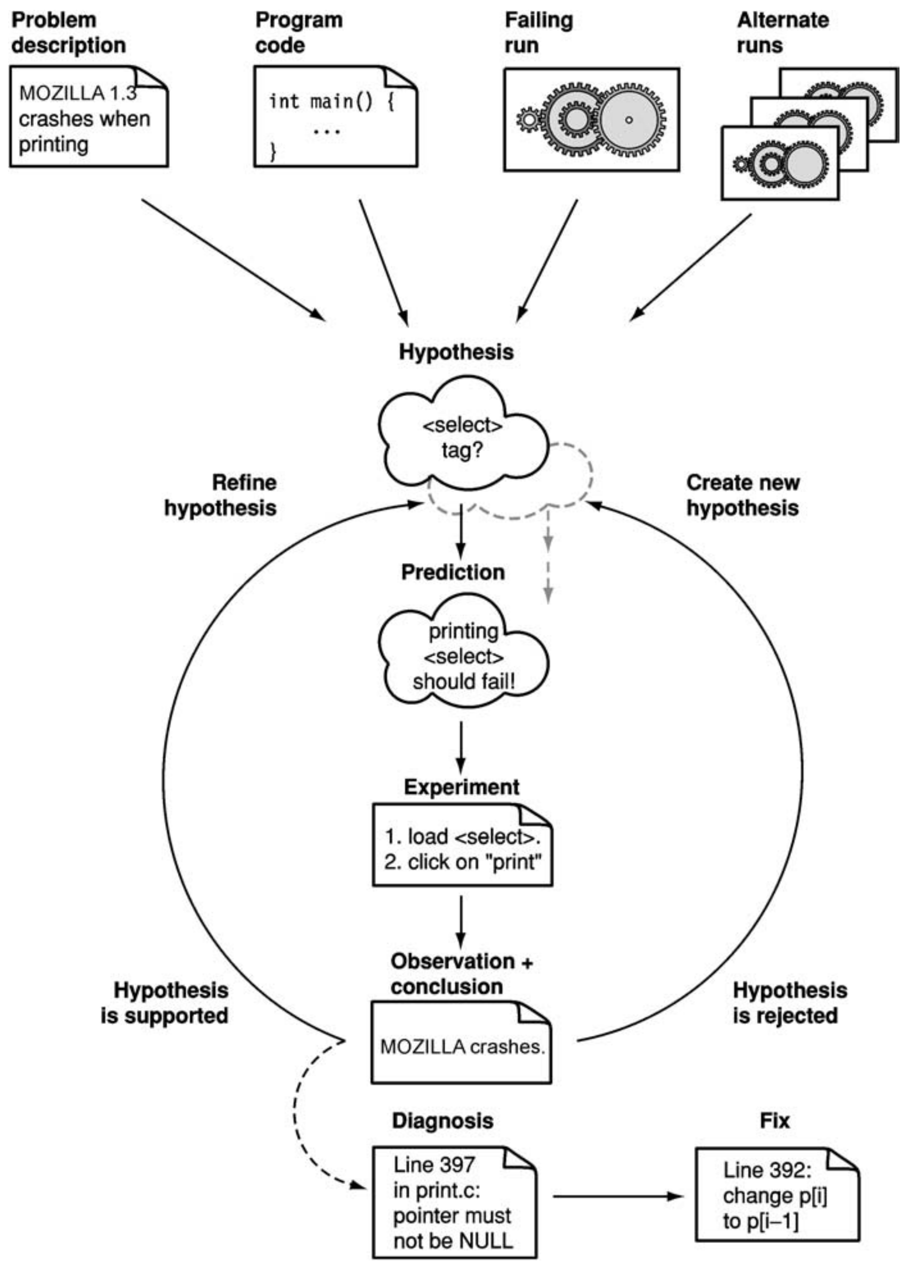

FIGURE 6.1 The scientific method of debugging.

In Section 1.4 we saw how to find the defect in the `sample` program — but in a rather *ad hoc* or unsystematic way. Let's now retell this debugging story using the concepts of scientific method.

6.3.1 Debugging `sample` — Preparation

We start with *writing down the problem*: what happened in the failing run and how it failed to meet our expectations. This easily fits within the scientific method scheme by setting up an initial hypothesis "The program works," which is then rejected. This way, we have observed the failure, which is the first step in the scientific method.

- *Hypothesis:* The `sample` program works.
- *Prediction:* The output of `sample 11 14` is `"11 14"`.
- *Experiment:* We run `sample` as previously.
- *Observation:* The output of `sample 11 14` is `"0 11"`.
- *Conclusion:* The hypothesis is *rejected.*

6.3.2 Debugging `sample` — Hypothesis 1

We begin with a little verification step: Is the zero value reported by `sample` caused by a zero value in the program state? Looking at Example 1.1, lines 38 through 41, it should be obvious that the first value printed (the zero) should be the value of `a[0]`. It is unlikely that this output code has a defect. Nonetheless, if it does we can spend hours and hours on the wrong trail. Therefore, we set up the hypothesis that `a[0]` is actually zero.

- *Hypothesis:* The execution causes `a[0]` to be zero.
- *Prediction:* `a[0] = 0` should hold at line 37.
- *Experiment:* Using a debugger, observe `a[0]` at line 37.
- *Observation:* `a[0] = 0` holds as predicted.
- *Conclusion:* The hypothesis is confirmed.

(What does "using a debugger" mean in practice? See Section 8.3.1 to find out.)

6.3.3 Debugging `sample` — Hypothesis 2

Now we must determine where the infection in `a[0]` comes from. We assume that `shell_sort()` causes the infection.

- *Hypothesis:* The infection does not take place until `shell_sort()`.
- *Prediction:* The state should be sane at the beginning of `shell_sort()` — that is, `a[] = [11, 14]` and `size = 2` should hold at line 6.
- *Experiment:* Observe `a[]` and `size`.
- *Observation:* We find that `a[] = [11, 14, 0]`, `size = 3` hold.
- *Conclusion:* The hypothesis is *rejected.*

6.3.4 Debugging `sample` — Hypothesis 3

Assuming we have only one infection site, the infection does *not* take place within `shell_sort()`. Instead, `shell_sort()` gets bad arguments. We assume that these arguments cause the failure.

- *Hypothesis:* Invocation of `shell_sort()` with `size = 3` causes the failure.
- *Prediction:* If we correct `size` manually, the run should be successful — the output should be `"11 14"`.
- *Experiment:* Using a debugger, we:
  1. Stop execution at `shell_sort()` (line 6).
  2. Set `size` from `3` to `2`.
  3. Resume execution.
- *Observation:* As predicted.
- *Conclusion:* The hypothesis is confirmed.

6.3.5 Debugging `sample` — Hypothesis 4

The value of `size` can only come from the invocation of `shell_sort()` in line 36 — that is, the `argc` argument. As `argc` is the size of the array *plus 1*, we change the invocation.

- *Hypothesis:* Invocation of `shell_sort()` with `size = argc` (instead of `size = argc - 1`) causes the failure.
- *Prediction:* If we change `argc` to `argc - 1`, the "Changing argc to argc -1" run should be successful. That is, the output should be `"11 14"`.
- *Experiment:* In line 36, change `argc` to `argc - 1` and recompile.
- *Observation:* As predicted.
- *Conclusion:* The hypothesis is confirmed.

After four iterations of the scientific method, we have finally refined our hypothesis to a theory; the diagnosis "Invocation of `shell_sort()` with `argc` causes the failure." We have proven this by showing the two alternatives:

- With the invocation `argc`, the failure occurs.
- With the invocation `argc - 1`, the failure no longer occurs.

Hence, we have shown that the invocation with `argc` caused the failure. As a side effect, we have generated a *fix* — namely, replacing `argc` with `argc - 1` in line 36.

Note that we have not yet shown that the change induces *correctness* — that is, `sample` may still contain other defects. In particular, in programs more complex than `sample` we would now have to validate that this fix does not introduce new problems (Chapter 15 "Fixing the Defect" has more on this issue). In the case of `sample`, though, you can do such a validation by referring to a higher authority: Being the author of this book, I claim that with the fix applied there is no way `sample` could ever sort incorrectly. Take my word.

6.4 EXPLICIT DEBUGGING

In Section 6.3 we saw how to use the scientific method to establish the failure cause. You may have noticed that the process steps were quite *explicit*: we explicitly stated the hypotheses we were examining, and we explicitly set up experiments that supported or rejected the hypotheses.

Being explicit is an important means toward understanding the problem at hand, starting with the problem statement. Every time you encounter a problem, write it down or tell it to a friend. Just *stating* the problem in whatever way makes you rethink your assumptions — and often reveals the essential clues to the solution. The following is an amusing implementation, as reported by Kernighan and Pike (1999):

One university center kept a Teddy bear near the help desk. Students with mysterious bugs were required to explain them to the bear before they could speak to a human counselor.

Unfortunately, most programmers are implicit about the problem statement, and even more so within the debugging process (they keep everything in their mind). But this is a dangerous thing to do. As an analogy, consider a *Mastermind game* (Figure 6.2). Your opponent has chosen a secret code, and you have a number of guesses. For each guess, your opponent tells you the number of tokens in your guess that had the right color or were in the right position.

FIGURE 6.2 A Mastermind game.

If you have ever played Mastermind and won, you have probably applied the scientific method.

However, as you may recall from your Mastermind experiences, you must remember all earlier experiments and their outcomes, in that this way you can keep track of all confirmed and rejected hypotheses. In a Mastermind game, this is easy, as the guesses and their outcomes are recorded on the board. In debugging, though, many programmers do not explicitly keep track of experiments and outcomes, which is equivalent to *playing Mastermind in memory.* In fact, forcing yourself to remember all experiments and outcomes prevents you from going to sleep until the bug is eventually fixed. Debugging this way, a "master mind" is not enough — you also need a "master memory."

6.5 KEEPING A LOGBOOK

A straightforward way of making debugging explicit and relieving memory stress is to write down all hypotheses and observations — that is, *keep a logbook*. Such a logbook can be either on paper or in some electronic form. Keeping a logbook may appear cumbersome at first, but with a well-kept logbook you do not have to keep all experiments and outcomes in memory. You can always quit work and resume next morning.

In *Zen and the Art of Motorcycle Maintenance*, Robert M. Pirsig writes about the virtue of a logbook in cycle maintenance:

> Everything gets written down, formally, so that you know at all times where you are, where you've been, where you're going, and where you want to get. In scientific work and electronics technology this is necessary because otherwise the problems get so complex you get lost in them and confused and forget what you know and what you don't know and have to give up.

And beware — this quote applies to *motorcycle maintenance.* Real programs are typically much more complex than motorcycles. For a motorcycle maintainer, it would probably appear amazing that people would debug programs *without* keeping logbooks.

And how should a logbook be kept? Unless you want to share your logbook with someone else, feel free to use any format you like. However, your notes should include the following points, as applied in Section 6.3.

- *Statement* of the problem (a problem report, as in Chapter 2 "Tracking Problems," or, easier, a report identifier)
- *Hypotheses* as to the cause of the problem

Hypothesis	Prediction	Experiment	Observation	Conclusion
Infection in `shell_sort()`	At `shell_sort()` (Line 6), expect `a[] = [11, 14]` and `size = 2`	Observe `a[]` and `size[]`	`a[] = [11,14,0]` and `size = 3`	*rejected*
Invocation of `shell_sort()` with `size = 3` causes failure	Setting `size = 2` should make `sample` work	Set `size = 2` using debugger	As predicted	*confirmed*

FIGURE 6.3 A debugging logbook (excerpt).

- *Predictions* of the hypotheses
- *Experiments* designed to test the predictions
- *Observed results* of the experiments
- *Conclusions* from the results of the experiments

An example of such a logbook is shown in Figure 6.3, recapitulating hypotheses 2 and 3 of Section 6.3. Again, quoting Robert Pirsig:

This is similar to the formal arrangement of many college and high-school lab notebooks, but the purpose here is no longer just busywork. The purpose now is precise guidance of thoughts that will fail if they are not accurate.

6.6 DEBUGGING QUICK-AND-DIRTY

Not every problem needs the full strength of the scientific method or the formal content of a logbook. Simple problems should be solved in a simple manner — without going through the explicit process. If we find a problem we suppose to be simple, the gambler in us will head for the lighter process. Why bother with formalities? Just think hard and solve the problem.

The problem with such an implicit "quick-and-dirty" process is to know when to use it. It is not always easy to tell in advance whether a problem is simple or not. Therefore, it is useful to set up a *time limit*. If after 10 minutes of quick-and-dirty debugging you still have not found the defect, go for the scientific method instead and write down the problem statement in the logbook.

Then, straighten out your head by making everything formal and exact — and feel free to take a break whenever necessary.

6.7 ALGORITHMIC DEBUGGING

Another way of organizing the debugging process is to *automate* it — at least partially. The idea of *algorithmic debugging* (also called *declarative debugging*) is to have a tool that guides the user along the debugging process *interactively.* It does so by asking the user about possible infection sources:

1. Assume an incorrect result R has the origins $O_1, O_2, \ldots, O_n$.
2. For each of the origins O_i, algorithmic debugging enquires whether the origin O_i is correct or not.
3. If one of the origins O_i is incorrect, algorithmic debugging restarts at step 1 with $R = O_i$.
4. Otherwise, all origins O_i are correct. Then, the infection must have originated at the place where R was computed from the origins. The process terminates.

Let's illustrate algorithmic debugging via an example. Example 6.1 shows a PYTHON sorting function: `sort(L)` is supposed to return a sorted copy of the list L. Unfortunately, `sort()` does not work properly: `sort([2, 1, 3])` returns `[3, 1, 2]` rather than `[1, 2, 3]`.

Our `sort()` function is based on *insertion sort.* It thus relies on a function `insert(X, L)`, which returns a list where X is inserted between the elements of L: `insert(2, [1, 3])` should return `[1, 2, 3]`.

Figure 6.4 summarizes the execution of `sort([2, 1, 3])` (a line stands for functions being invoked). Each invocation of `sort(L)` first calls itself for the tail of L and then calls `insert()` to insert the first element of L into the freshly sorted list.

The execution tree shown in Figure 6.4 now becomes the base for the algorithmic debugging process. Assume we have a tool that implements algorithmic debugging for PYTHON , working on the console. Such a tool would first ask us whether the end result is correct, which we decline:

```
sort([2, 1, 3]) = [3, 1, 2]? no
```

```
def insert(elem, list):
    """Return a copy of LIST with ELEM sorted in"""
    if len(list) == 0:
        return [elem]

    head = list[0]
    tail = list[1:]
    if elem <= head:
        return list + [elem]

    return [head] + insert(elem, tail)

def sort(list):
    """Return a sorted copy of LIST"""
    if len(list) <= 1:
        return list

    head = list[0]
    tail = list[1:]
    return insert(head, sort(tail))
```

EXAMPLE 6.1 A buggy insertion sort program.

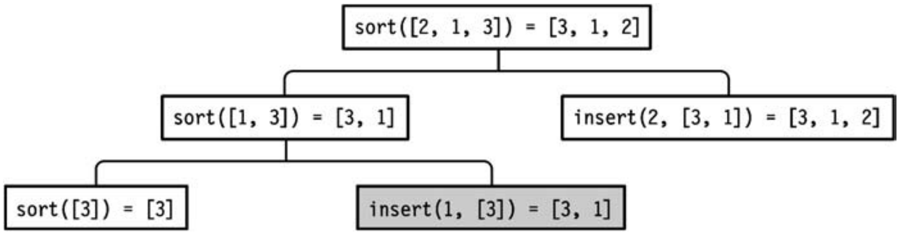

FIGURE 6.4 Execution tree of the sorting function in Example 6.1. Each computation of a function (top) relies on further function calls (below).

The error can originate either from `sort([1, 3])` or the subsequent `insert()` call. Algorithmic debugging starts querying about the first origin:

```
sort([1, 3]) = [3, 1]? no
```

Again, the error could originate from the earlier `sort([3])` call. Is the result correct?

```
sort([3]) = [3]? yes
```

Because the call `sort([3]) = [3]` was fine but `sort([1, 3]) = [3, 1]` is wrong, the error could have originated in the `insert()` call. It actually does, because `insert(1, [3])` should return `[1, 3]`, and not `[3, 1]`:

```
insert(1, [3]) = [3, 1]? no
```

As `insert(1, [3])` invokes no further functions, we have isolated the defect. The infection originates at the call `insert(1, [3]) = [3, 1]` (shown in gray). Our algorithmic debugging tool reports:

```
An error has been localized in the body of insert().
```

We can even narrow down the infection to the code executing in the call `insert(1, [3])`. This leaves us with the return statement:

```
if elem <= head:
    return list + [elem]
```

This statement is wrong. If the element to be inserted is smaller than the head of the list, it should be inserted at the beginning rather than at the end. The statement thus must read:

```
if elem <= head:
    return [elem] + list
```

This fixes the `sort()` function from Example 6.1. With this fix applied, it sorts just fine.

The general idea of having an algorithm drive the debugging process is applicable to arbitrary debugging techniques. Wherever we search an error — and need to rely on human input to decide what is correct, right, or true — algorithmic debugging can drive the search in a systematic way. Unfortunately, algorithmic debugging has not been demonstrated to be effective for real-world applications:

- *The process does not scale.* In a large imperative program, there are millions and millions of functions being executed. Many of these functions communicate via shared data structures, rather than simple arguments and returned values. Worse yet, the data structures being accessed are far too huge to be checked manually. Imagine debugging a compiler: "Are these 4 megabytes of executable code correct? (yes/no)?"

For these reasons, algorithmic debugging works best for functional and logical programming languages. Functional and logical programs have few or no side effects — that is, there is no shared state that is updated, and the user does not have to check the entire program state. For logical languages such as PROLOG , an execution tree (Figure 6.4) becomes a proof tree, which is part of every program execution.

- *Programmers prefer driving to being driven.* The algorithmic debugging process, as implemented in early tools, is extremely rigid. Many programmers do not like being instrumented in such a mechanical way. Making the process user friendly in the sense that it provides *assistance* to programmers (rather than having the programmer assist the tool) is an open research issue.

  It is conceivable that future tools, combined with the analysis techniques defined in this book, will provide guidance to the programmer by asking the right questions. In particular, one can think of programmers providing specifications of particular function properties — specifications that can then be reused for narrowing down the incorrect part.

All of these problems disappear if we replace the programmer being queried by an *oracle* — an automatic device that tells correctness from noncorrectness. To determine its result, such an oracle would use an external specification. In this book, however, we assume that there is no such specification (except from the final test outcome) — at least not in a form a mechanical device could make use of it. Therefore, scientific method still needs human interaction.

6.8 DERIVING A HYPOTHESIS

Scientific method gives us a general process for turning a hypothesis into a theory — or, more specifically, an initial guess into a diagnosis. But still, within each iteration of the scientific method we must come up with a new hypothesis. This is the creative part of debugging: thinking about the many ways a failure could have come to be. This creative part is more than just mechanically enumerating possible origins, as in algorithmic debugging.

Unfortunately, being creative is not enough: we must also be *effective.* The better our hypotheses the less iterations we need and the faster the diagnosis is done. To be effective, we need to leverage as many knowledge sources as possible. These are the *ingredients of debugging*, as shown in Figure 6.1.

- *The description of the problem:* Without a concise description of the problem, you will not be able to tell whether the problem is solved or not. A simplified problem report also helps. In Chapter 2 "Tracking Problems" we saw examples of such descriptions, and discussed the issues of tracking problems. Chapter 5 "Simplifying Problems" provided details on the simplification of problem reports.

- *The program code:* The program code is the common abstraction across all possible program runs, including the failing run. It is the basis for almost all debugging techniques.

  Without knowledge about the internals of the program, you can only observe concrete runs (if any) without ever referring to the common abstraction. Lack of program code makes understanding (and thus debugging) much more difficult. As you cannot recreate the program from code, you must *work around defects,* which is far less satisfactory than fixing the code itself.

  As an example, consider the `sort()` algorithmic debugging session in Section 6.7. In principle, we (as users) could have run the session without knowing the source code. To determine whether a result is correct or not, all we need is a specification. However, the tool itself must have access to the code (Example 6.1) in order to trace and instrument the individual function calls. Chapter 7 "Deducing Errors" discusses techniques for reasoning from the (abstract) program code to the (concrete) program run — including the failing run.

- *The failing run:* The program code allows you to *speculate* about what may be going on in a concrete failing run. If you actually *execute* the program such that the problem is reproduced, you can observe actual *facts* about the concrete run. Such facts include the code being executed and the program state as it evolves. These observation techniques are the bread and butter of debugging.

  Again, debugging the `sort()` code in Example 6.1 becomes much easier once one can talk about a concrete (failing) run. In principle, one could do without observation. This is fine for proving abstract properties but bad for debugging concrete problems.

Chapter 8 "Observing Facts" discusses techniques with which programmers can observe concrete runs. Chapter 10 "Asserting Expectations" extends these techniques to have the computer detect violations automatically.

- *Alternate runs:* A single run of a nontrivial program contains a great deal of information, and thus we need a means of focusing on specific aspects of the execution. In debugging, we are most interested in *anomalies* — those aspects of the failing run that *differ* from "normal" passing runs. For this purpose, we must know which "normal" runs exist, what their common features are, and how these differ in the failing run.

  In the `sort()` example, algorithmic debugging has used alternate runs of individual functions to narrow down the defect. From the fact that `sort([3])` worked, and `sort([1, 3])` failed, algorithmic debugging could deduce that the error must have originated in the `insert()` call taking place between the two `sort()` calls.

  In practice, we seldom have a specification available to tell us whether some aspect of a run is correct or not. Yet, with a sufficient number of alternate runs we can classify what is "normal" or not. Chapter 11 "Detecting Anomalies" discusses automated techniques for detecting and expressing commonalities and anomalies across multiple runs.

- *Earlier hypotheses:* Depending on the outcome of a scientific method experiment, one must either *refine* or *reject* a hypothesis. In fact, every new hypothesis must

  – *include* all earlier hypotheses that passed (whose predictions were satisfied) and

  – *exclude* all hypotheses that failed (whose predictions were not satisfied).

  Any new hypothesis must also explain all earlier *observations,* regardless of whether the experiment succeeded or failed — and it should be different enough from earlier hypotheses to quickly advance toward the target. Again, the algorithmic debugging session is a straightforward example of how the results of earlier tests (i.e., answers given by the user) drive the scientific method and thus the debugging process. The final diagnosis of `insert()` having a defect fits all passed hypotheses and explains all earlier observations.

  To automate the process, we would like to reuse earlier hypotheses without asking the user for assistance. If a hypothesis is about a cause (such as a failure cause), the search for the actual cause can be conducted systematically by narrowing the difference between a passing and a failing scenario. These techniques can be automated and applied to program runs. Chapter 13 "Isolating Failure Causes" discusses automating the search for failure-inducing circumstances. Chapter 14 "Isolating Cause-Effect Chains" does the same for program states.

6.9 REASONING ABOUT PROGRAMS

Depending on the ingredients that come into play, humans use different reasoning techniques to learn about programs. These techniques form a *hierarchy,* as shown in Figure 6.5.

- *Deduction:* Deduction is reasoning from the general to the particular. It lies at the core of all reasoning techniques. In program analysis, deduction is used for reasoning from the program code (or other abstractions) to concrete runs — especially for deducing what can or cannot happen. These deductions take the form of mathematical proofs. If the abstraction is true, so are the deduced properties. Because deduction does not require any knowledge about the concrete, it is not required that the program in question actually be executed.

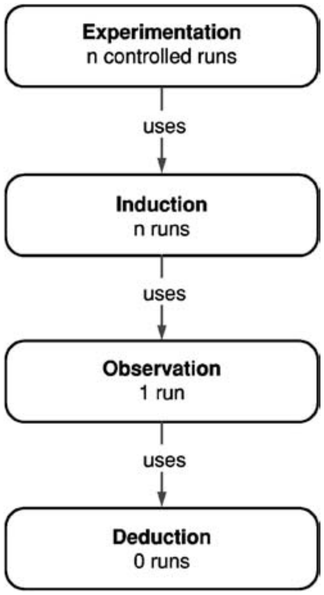

FIGURE 6.5 A hierarchy of program analysis techniques.

In this book, we call any technique *static analysis* if it infers findings without executing the program — that is, the technique is based on deduction alone. In contrast, *dynamic analysis* techniques use actual executions.

As Nethercote (2004) points out, this distinction of whether a program is executed or not may be misleading. In particular, this raises the issue of what exactly is meant by "execution." Instead, he suggests that *static techniques predict approximations of a program's future; dynamic analysis remembers approximations of a program's past.* Because in debugging we are typically concerned about the past, most interesting debugging techniques fall into the "dynamic" categories, which we discuss next.

- *Observation:* Observation allows the programmer to inspect arbitrary aspects of an individual program run. Because an actual run is required, the associated techniques are called *dynamic.* Observation brings in actual *facts* of a program execution. Unless the observation process is flawed, these facts cannot be denied.

  In this book, we call a technique *observational* if it generates findings or approximations from a *single execution* of the program. Most observational techniques make use of the program code in some form or another and thus also rely on deduction.

- *Induction:* Induction is reasoning from the particular to the general. In program analysis, induction is used to *summarize* multiple program runs (e.g., a test suite or random testing) to some abstraction that holds for all considered program runs. In this context, a "program" may also be a piece of code that is invoked multiple times from within a program — that is, some function or loop body.

  In this book, we call a technique *inductive* if it generates findings from *multiple executions* of the program. By definition, every inductive technique makes use of observation.

- *Experimentation:* Searching for the cause of a failure using scientific method (Chapter 6 "Scientific Debugging") requires a series of *experiments*, refining and rejecting hypotheses until a precise diagnosis is isolated. This implies multiple program runs that are *controlled* by the reasoning process.

  In this book, we call a technique *experimental* if it generates findings from *multiple executions* of the program that are *controlled* by the technique. By definition, every experimental technique uses induction and thus observation.

In the following chapters, we examine the most important of these techniques. We start with Chapter 7 "Deducing Errors," deducing hypotheses from the program code without actually executing the program. Chapter 8 "Observing Facts," Chapter 9 "Tracking Origins," and Chapter 10 "Asserting Expectations" focuses on observational techniques. Chapter 11 "Detecting Anomalies" discusses inductive techniques. Finally, Chapter 13 "Isolating Failure Causes" and Chapter 14 "Isolating Cause-Effect Chains" introduce experimental techniques.

6.10 CONCEPTS

✎ *To isolate a failure cause*, use *scientific method* (Section 6.2): **HOW TO**

1. Observe a failure (i.e., as described in the problem description).
2. Invent a *hypothesis* as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make *predictions.*
4. Test the hypothesis by *experiments* and further *observations:*
    - If the experiment satisfies the predictions, refine the hypothesis.
    - If the experiment does not satisfy the predictions, create an alternate hypothesis.
5. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

✎ *To understand the problem at hand*, make it explicit. Write it down or talk to a friend (Section 6.4). **HOW TO**

✎ *To avoid endless debugging sessions*, make the individual steps explicit. Keep a logbook (Section 6.5). **HOW TO**

✎ *To locate an error in a functional or logical program*, consider algorithmic debugging. **HOW TO**

✎ Algorithmic debugging drives the debugging process by proposing hypotheses about error origins, which the user (or some oracle) must individually judge.

✎ *To debug quick-and-dirty*, think hard and solve the problem — but as soon you exceed some time limit go the formal way (Section 6.6). **HOW TO**

✎ *To derive a hypothesis*, consider: **HOW TO**

- The problem description
- The program code
- The failing run
- Alternate runs
- Earlier hypotheses

See Section 6.8 for details.

HOW TO

✎ *To reason about programs*, one can use four different techniques:

- Deduction (zero runs)
- Observation (one single run)
- Induction (multiple runs)
- Experimentation (multiple controlled runs)

All of these are discussed in further chapters.

6.11 FURTHER READING

Algorithmic debugging as a semiautomation of scientific method was conceived by Shapiro (1982) for logical programming languages such as PROLOG. In 1992, Fritzson et al. extended the approach to imperative languages, using program slicing (Section 7.4) to determine data dependences, and demonstrated the feasibility on a subset of PASCAL. The algorithmic debugging example session is based on Fritzson et al. (1992). In 1997, Naish generalized algorithmic debugging to the more general concept of declarative debugging.

Whereas the *scientific method* is the basis of all experimental science, it is rarely discussed or used in computer science. The reason is that computer science is concerned with artifacts, which are supposed to be fully under control and fully understood. However, as an unexpected failure occurs the artifact must be explored just like some natural phenomenon. For an early but still excellent book on experimental and statistical methods for data reduction, see *An Introduction to Scientific Research* by Wilson (1952). A more general book from the same period that remains useful today is *The Art of Scientific Investigation* by Beveridge (1957).

For philosophy of science, the undisputed classic is the work of Popper (1959), who coined the term *falsifiability* as the characteristic method of scientific investigation and inference. For Popper, any theory is scientific only if it is refutable by a conceivable event — which is why experiments play such a role in obtaining diagnoses.

The definitions of cause and effect in this book are called based on *counterfactuals,* because they rely on assumptions about nonfacts. The first counterfactual definition of causes and effects is attributed to Hume (1748): "If the first object [the cause] had not been, the second [the effect] never had existed." The best-known counterfactual theory of causation was elaborated by Lewis (1973), refined in 1986.

Causality is a vividly discussed philosophical field. Other than the counterfactual definitions, the most important alternatives are definitions based on *regularity* and *probabilism.* I recommend Zalta (2002) for a survey.

6.12 EXERCISES

EXERCISE 6.1. "We have reached a state where many programs are just as unpredictable as natural phenomena." Discuss.

EXERCISE 6.2. Using the logbook format (Section 6.5), describe the individual steps of the algorithmic debugging run in Section 6.7. Which are the hypotheses, predictions, and experiments?

EXERCISE 6.3. Simplification of tests, as discussed in Chapter 5 "Simplifying Problems," can be seen as an application of the scientific method. What are the hypotheses, predictions, and tests being used?

EXERCISE 6.4. Set up a logbook *form sheet* with entries such as "Prediction," "Observation," and so on such that programmers only need to fill in the gaps. Give them to fellows and collect their opinions.

EXERCISE 6.5. "I want to *archive* logbook entries, such that in case a similar problem occurs I may find hints on which hypotheses to use and which experiments to conduct." Discuss.