CHAPTER

Asserting Expectations 10

Observation alone is not enough for debugging. One must *compare* the observed facts with the expected program behavior. In this chapter, we discuss how to automate such comparisons, using well-known *assertion* techniques. We also show how to ensure the sanity of important system components such as memory.

10.1 AUTOMATING OBSERVATION

Observing what is going on in a program can be a burden for a programmer. First, there are many states and events that can and must be observed. Second, for each new run (or each new defect) the same items must be observed again. The burden is not so much the observation itself but the act of *judging* whether the observed state is sane or not—which of course multiplies with the amount of data observed.

In the past, where computing time was expensive, having such judging done by humans was commonplace (see Bug Story 9, for instance). Given our current wealth of computing power, though, it becomes more and more reasonable to *shift*

BUG STORY 9

Examining a Lot of Data

Holberger drives into Westborough. The sun is in his eyes this morning, and he wonders in a detached sort of way where it will be hitting his windshield when they finish this job. Debugging Eagle has the feel of a career in itself. Holberger isn't thinking about any one problem, but about all the various problems at once, as he walks into the lab. What greets him there surprises him. He shows it by smiling wryly. A great heap of paper lies on the floor, a continuous sheet of computer paper streaming out of the carriage at Gollum's system console. Stretched out, the sheet would run across the room and back again several times. You could fit a fairly detailed description of American history from the Civil War to the present on it. Veres sits in this midst of this chaos, the picture of a scholar. He's examined it all. He turns to Holberger. "I found it," he says.

Source: Kidder (1981).

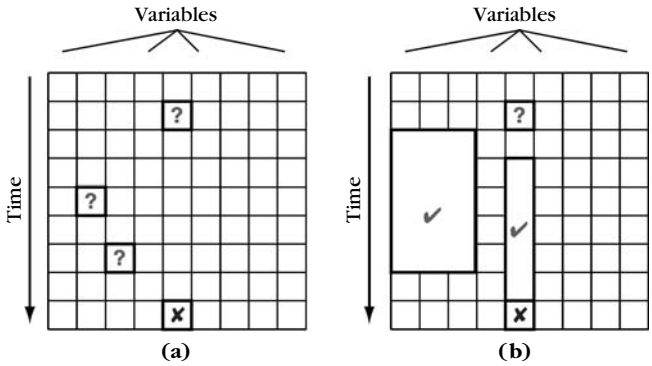

FIGURE 10.1

Observation (a) versus assertion (b). Whereas observation is limited to small probes in space and time, assertions can automatically cover a large area.

the burden of observation over to the computer—that is, to have the computer check whether the program state is still sane, or whether an infection has occurred. This is the same approach as in automated testing (Chapter 3). Rather than observe and judge for ourselves, we have the program test its own state continuously.

The first advantage of automated observation is *scalability.* In contrast to manual observation, where we can only examine a small number of variable values during execution (Figure 10.1a), having the computer observe and judge allows us to check large parts of the execution automatically (Figure 10.1b). Each automated observation acts like an *infection detector*, catching infections before they can propagate and obscure their origins. This effectively narrows down possible infection sites, and thus speeds up debugging.

The second advantage is, of course, *persistence.* Once we have specified the properties of a sane state, we can reuse this specification over and over again. This not only includes debugging but documentation and general program understanding. Of all debugging techniques, expressing and checking what a program is supposed to do is probably the best long-term investment in code quality. Let's thus explore some common assertion techniques. Here, we ask:

How do we automate observation?

10.2 BASIC ASSERTIONS

To have the program ensure its sane state automatically is deceptively simple. The basic idea is to insert appropriate code that *checks for infections.* For instance, to ensure that a divisor is nonzero, one could write:

```
if (divisor == 0) {
    printf("Division by zero!");
    abort();    // or some other exceptional behavior
}
```

Such infection-handling code has been used since the dawn of computing. It is common and useful—and yet somewhat clumsy. As discussed in Section 8.2 in Chapter 8, it is wise to explicitly separate code concerned with debugging from the normal program code. One way to do so is to have a special function that explicitly checks for a specific condition. The common name of such a function is `assert(x)`, with the functionality that it aborts execution if `x` should be false.

```
assert(divisor != 0);
```

The `assert()` function could be implemented in a straightforward way, as in:

```
void assert(int x)
{
    if (!x)
    {
        printf("Assertion failed!\n");
        abort();
    }
}
```

In practice, though, simply having assertions marked as such does not suffice. Just as with logging functions (see Section 8.2.1 in Chapter 8), we want to be able to turn assertions off, and we want them to report diagnostic information—at least more than just "assertion failed."

```
$ ./my-program
divide.c:37: assertion 'divisor != 0' failed
Abort (core dumped)
$ _
```

The techniques that realize these features in logging functions can easily be used for assertions. The following is a typical definition of `assert()` for C++ programs.

```
#ifndef NDEBUG
#define assert(ex) \
((ex) ? 1 : (std::cerr << __FILE__ << ":" << __LINE__ \
                  << ": assertion '" #ex "' failed\n", \
                  abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```

This definition uses the `__FILE__` and `__LINE__` macros to report the location of the assertion in the source file. It also uses the "stringize" mechanism of the C preprocessor to output the assertion that actually failed (`#ex`). Finally, by setting the `NDEBUG` preprocessor variable the assertion can be turned off (it compiles to a no-op).

Other languages come with assertions built into the language. In JAVA, assertions are specified using the `assert` keyword. They work just like the C++ macro, but throw an exception rather than aborting the program. Note that in the JAVA interpreter `java` assertions are turned off by default. They must be enabled explicitly using the `-enableassertions` option.

How does one use assertions during debugging? The basic idea of assertions is to have the computer do the observation. Thus, assertions can be spread across the code just like logging functions. The principles of observation, as discussed in Section 8.1 in Chapter 8, still apply. Assertions should not interfere with the actual run (and thus have no side effects), and should be used *systematically* rather than sprinkled randomly across the code. This brings us to two major (systematic) uses of assertions in debugging:

- *Data invariants* that ensure data integrity.
- *Pre- and postconditions* that ensure function correctness.

10.3 ASSERTING INVARIANTS

The most important use of assertions in debugging is to ensure *data invariants*—properties that must hold throughout the entire execution. As an example, consider a C++ class `Time` that manages times of the day—say, a time such as "5pm, 40 minutes, and 20 seconds" or brief, 17:40:20. We do not care about how time is actually represented internally (that is the secret of the class), but we know that it provides an interface that allows us to access the individual components of the current time.

```
class Time {
public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (including leap seconds)

    void set_hour(int h);
    ...
}
```

In the case of `Time`, a sane state is a valid time from 00:00:00 to 23:59:60. From the client view, this is an invariant that holds for all `Time` objects for all times. In practice, this means that the invariant should hold at the beginning and end of each public method. For this purpose, we can easily write an assertion at the end of the `set_hour()` method. This ensures that whatever `set_hour()` does the invariant is not violated.

```
void Time::set_hour(int h)
{
    // precondition
    assert(0 <= hour() && hour() <= 23) &&
```

```
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
        ...

        // postcondition
        assert(0 <= hour() && hour() <= 23) &&
            (0 <= minutes() && minutes() <= 59) &&
            (0 <= seconds() && seconds() <= 60);
    }
```

(Note that we use the public interface of `Time`, rather than accessing the internals. This way, we can check three more functions.)

With these assertions, we can ensure that no `set_hour()` invocation will ever make a `Time` object inconsistent. The violated assertion would immediately flag the infection. However, putting such a three-line assertion at the beginning and end of each `Time` method induces redundancy and makes the code difficult to read. A more elegant way is to introduce a specific *helper function*, which checks the sanity of a `Time` object.

```
bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}
```

`sane()` is more than just a helper function. `sane()`, being true, is an *invariant* of the `Time` object. It should always hold before and after each public function. We can now ensure that this invariant holds for the current `Time` object whenever some method is called—for instance, at the beginning and end of `set_hour()`—and thus ensure that the method did not infect the object state.

```
void Time::set_hour(int h)
{
    assert(sane()); // precondition

    // Actual code goes here

    assert(sane()); // postcondition
}
```

If one of these assertions now fails, we can immediately narrow down our hypothesis on the location of the defect.

- If the precondition is violated, the infection must have taken place before `set_hour()`.
- If the postcondition is violated, the infection must have taken place within `set_hour()`.
- If the postcondition holds, the infection cannot have taken place in `set_hour()`.

To have the entire class continuously checked for sanity, all one needs to do is to wrap each public method that changes the state into two assertions—both checking the sanity as described previously. This ensures that any infection that takes place in these methods is properly caught—and if all assertions pass, we can rule out `Time` as an infection site.

If data structures get more complex, the invariants become more complex, too—but also ensure more properties. Example 10.1 shows an excerpt of a class invariant of a JAVA red/black tree—the base of the JAVA `TreeMap` class. Every property of the tree is checked in an individual helper function. The `sane()` method calls them all together. If anything ever goes wrong in a red/black tree, this `sane()` invariant will catch it.

EXAMPLE 10.1: Ensuring the invariant of a red/black tree

```
boolean sane() {
    assert (rootHasNoParent());
    assert (rootIsBlack());
    assert (redNodesHaveOnlyBlackChildren());
    assert (equalNumberOfBlackNodesOnSubtrees());
    assert (treeIsAcyclic());
    assert (parentsAreConsistent());

    return true;
}

boolean redNodesHaveOnlyBlackChildren() {
    workList = new LinkedList();
    workList.add(rootNode());
    while (!workList.isEmpty()) {
        Node current = (Node)workList.removeFirst();
        Node cl = current.left;
        Node cr = current.right;
        if (current.color == RED) {
            assert  (cl == null || cl.color == BLACK);
            assert  (cr == null || cr.color == BLACK);
        }
        if (cl != null) workList.add(cl);
        if (cr != null) workList.add(cr);
   }

   return true;
}

boolean rootHasNoParent() { ... }
```

Ideally, we would set up our class with `assert(sane())` statements at the beginning and end of each public method. Unfortunately, this clutters the code somewhat. To reduce clutter, we can use an aspect (see Section 8.2.3 in Chapter 8). For

the red/black tree in Example 10.1, one single aspect can ensure that the invariant holds before and after each modifying method (`add...` or `del...`).

```
public aspect RedBlackTreeSanity {
    pointcut modify():
        call(void RedBlackTree.add*(..)) ||
        call(void RedBlackTree.del*(..));

    before(): modify() {
        assert(sane());
    }

    after(): modify() {
        assert(sane());
    }
}
```

By applying or not applying the aspect, one can easily turn the assertions on and off—for all red/black tree methods now and in the future.

Once one has a function that checks data invariants, one can also invoke it in an interactive debugger to check data sanity on-the-fly. A conditional breakpoint in GDB such as

```
(gdb) break 'Time::set_hour(int)' if !sane()
Breakpoint 3 at 0x2dcf: file Time.C, line 45.
(gdb) _
```

acts like an assertion. It will interrupt execution as soon as the breakpoint condition holds at the specified location—that is, the `Time` invariant is violated. This even works if the assertions have been disabled.

10.4 ASSERTING CORRECTNESS

In addition to data invariants, the major use of assertions in debugging is to *ensure that some function does the right thing.* In the `set_hour()` example, for instance, we can assert that `set_hour()` does not result in an invalid `Time` state. However, how do we know that `set_hour(h)` is correct—that is, that it sets the hour to `h`?

Such properties are expressed as *postconditions*—conditions over the state that must hold at the end of the function. As an example, consider a divide function for computing a *quotient* and a *remainder* from a *dividend* and a *divisor*. For such a function, the precondition is $divisor \neq 0$, whereas the postcondition is $quotient \times divisor + remainder = dividend$. Again, a postcondition can be translated into an assertion

```
assert(quotient * divisor + remainder == dividend);
```

at the end of the divide function code to check whether the computed quotient and divisor values are actually correct. In our `set_hour()` example, this reads:

```
void Time::set_hour(int h)
{
   // Actual code goes here

   assert(hour() == h);   // postcondition
}
```

Whereas a postcondition applies to the state that holds at the end of a function, a *precondition* expresses conditions over the state that must hold at the *beginning* of a function. For instance, to make our divide function work properly the *divisor* must not be zero. This condition is again expressed as an assertion

```
assert(divisor != 0);
```

at the beginning of the divide function. The following is the precondition for `set_hour()`.

```
void Time::set_hour(int h)
{
   assert(0 <= h && h <= 23);

   // Actual code goes here
}
```

For complex data structures, specifying a pre- or postcondition may require the use of *helper functions* that check for the individual properties. For instance, if a sequence is to be sorted we need a helper function that checks whether the postcondition is satisfied.

```
void Sequence::sort()
{
    // Actual code goes here

    assert(is_sorted());
}
```

Helper functions used in postconditions usually make useful public methods.

```
void Container::insert(Item x)
{
    // Actual code goes here

    assert(has(x));
}
```

And, of course, *invariants* (as discussed in Section 10.3) are essential pre- and postconditions.

```
void Heap::merge(Heap another_heap)
{
    assert(sane());
```

```
    assert(another_heap.sane());

    // Actual code goes here

    assert(sane());
}
```

Sometimes, a postcondition involves an *earlier state*—that is, a state that occurred at the beginning of the function. In the case of the `set_hour(h)` function, for instance, we might want to specify that `set_hour(h)` only sets the hours (i.e., it does not change the minutes or the seconds). Asserting this requires us to save the original values at the beginning, such that we can compare against them at the end.

```
void Time::set_hour(int h)
{
   int old_minutes = minutes();
   int old_seconds = seconds();
   assert(sane());

   // Actual code goes here

   assert(sane());
   assert(hour() == h);
   assert(minutes() == old_minutes &&
          seconds() == old_seconds);
}
```

This works fine, but is somewhat cumbersome. In particular, if we turn off the assertions we end up with two unused variables, which will result in a compiler warning (see Section 7.5 in Chapter 7 on code smells).

In other languages, specifying pre- and postconditions is much more elegant. The EIFFEL language incorporates the concept of *design by contract*, where a contract is a set of preconditions that must be met by the caller (the *client*) and a set of postconditions that are guaranteed by the callee (the *supplier*). In EIFFEL, a contract regarding `set_hour()` would be specified as:

```
set_hour (h: INTEGER) is
        -- Set the hour from 'h'
    require
        sane_h: 0 <= h and h <= 23
    ensure
        hour_set: hour = h
        minute_unchanged: minutes = old minutes
        second_unchanged: seconds = old seconds
```

These conditions are again checked at runtime, just like assertions (although a true EIFFEL afficionado would shudder at the low-level nature of assertions). Note that the contract is part of the interface description (in contrast to `assert` calls). It is thus visible to the client and thus serves as documentation. In addition to `require`

and `ensure`, EIFFEL provides an `invariant` keyword the condition of which is checked before and after every invocation of a public method.

10.5 ASSERTIONS AS SPECIFICATIONS

The EIFFEL example in Section 10.4 can already serve as a *specification* of what the function should do. In short, the assertions become part of the *interface.* A user of the `set_hour()` function must satisfy its precondition. A supplier of the `set_hour()` function must satisfy the postcondition under the assumption the precondition holds. (The invariants, if any, must also hold.) This idea of assertions, written using program code, that serve as specifications is something quite recent. In the past, program specifications were of two forms:

1. *Natural language.* The great majority of all specifications are written in natural language: "`set_hour(h)` sets the current hour to `h`, where `h` is an integer in the range 0–23." This style of specification is easy to read by humans, but is difficult to make complete and unambiguous. Furthermore, compliance with a natural-language specification cannot be validated automatically.
2. *Formal systems.* The most complete and unambiguous language we have is mathematics, and discrete mathematics is the basis for the several specification languages, such as *Z*. Figure 10.2 shows a *Z* specification for the `Date` class and its `set_hour()` method. It is easy to recognize the invariants as well as the (intermixed) pre- and postconditions. Such a specification is obviously far more precise than the natural-language version.

   However, validating that the program code satisfies the specification is difficult. It requires us to prove that the concrete code requires no more than the abstract specification and that it provides no less than the specification. Such formal proofs can be tedious, and must be redone after each change to the code. (On the other hand, once your code is proven correct there would be no reason to change it again—unless the requirements change.)

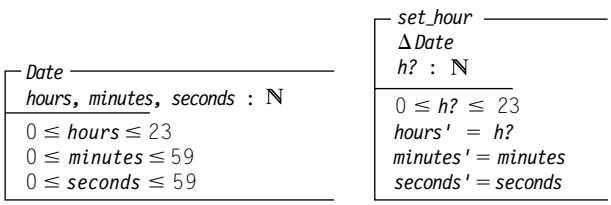

FIGURE 10.2

A *Z* specification for `Date` and `set_hour()`. The specification states the invariants that hold for `Date` as well as the pre- and postconditions for `set_hour()`.

However, both approaches have in common that specification and code are *separated*—leading to huge efforts when it comes to mapping one onto the other. In contrast, EIFFEL-style specifications (and, more generally, all assertions) integrate the specification within the code. Thus, they allow us to *validate* correctness simply by running the program. For every program run where the assertions hold, the assertions guarantee the correctness of the result. For the large bulk of mainstream software, this is already largely sufficient. (There are some programs, though, where a failing assertion is not an option. A computer controlling an airbag, for instance, simply *must* produce a result in a specific time frame. For such dependable systems, we must still prove program correctness for all runs.)

10.6 FROM ASSERTIONS TO VERIFICATION

The success of specification languages largely depends on their expressive power as well as the quantity and quality of available tools. A language that excels in both areas is JML, the Java Modeling Language. JML assertions are written as special comments in the JAVA code, recognized by JML tools alone and ignored by ordinary JAVA compilers. Using `requires` and `ensures` keywords, one can specify the pre- and postconditions of an individual JAVA method in EIFFEL style.

```
/*@ requires 0 <= h && h <= 23
  @ ensures  hours() == h &&
  @          minutes() == \old(minutes()) &&
  @          seconds() == \old(seconds())
  @*/
void Time::set_hour(int h) ...
```

As in this example, assertions are written as ordinary Boolean JAVA expressions together with some extra operators such as `\old`, which stands for the value of the variable at the moment the method was entered. Using JMLC, the JML compiler, such a specification can be translated into assertions that are then checked at runtime.

JML assertions are more than syntactic sugar around ordinary assertions, though. For one thing, they can serve as specifications. What does the following fragment of JML specify?

```
/*@ requires x >= 0.0;
  @ ensures JMLDouble
  @         .approximatelyEqualTo
  @         (x, \result * \result, eps);
  @*/
```

Example 10.2 shows a more complex JML example—a specification for a debit card. Note the use of invariants (`invariant`) to assert data sanity (as in EIFFEL, the invariant is checked before and after every invocation of a public method), the use of quantifiers (`\forall`) to express conditions that span multiple variables, and the specification of exceptional behavior (`signals`).

EXAMPLE 10.2: A debit card specified in JML

```
public class Purse {
  final int MAX_BALANCE;
  int balance;
  //@ invariant  0 <= balance && balance <= MAX_BALANCE;

  byte[] pin;
  /*@ invariant pin != null && pin.length == 4 &&
    @           (\forall int i; 0 <= i && i < 4;
    @                         0 <= byte[i] && byte[i] <= 9)
    @*/

  /*@ requires   amount >= 0;
    @ assignable balance;
    @ ensures    balance == \old(balance) - amount &&
    @            \result == balance;
    @ signals    (PurseException) balance == \old(balance);
    @*/

  int debit(int amount) throws PurseException {
  ...
  }
```

Source: Burdy et al. (2003).

Again, JMLC translates all of these conditions into runtime assertions and thus ensures that no violation passes by unnoticed. However, there is even more to gain from JML, such as the following.

- *Documentation.* The JMLDOC documentation generator produces HTML containing both JAVADOC comments and JML specifications. This is a great help for browsing and publishing JML specifications.
- *Unit testing.* JMLUNIT combines the JML compiler JMLC with JUNIT (see Chapter 3) such that one can test units against JML specifications.
- *Invariant generation.* The DAIKON invariant-detection tool (see Chapter 11) can report detected invariants in JML format, thus allowing simple integration with JML.
- *Static checking.* The ESC/JAVA static checker checks simple JML specifications statically, using the deduction techniques laid out in Chapter 7. In particular, it can leverage specified invariants to detect potential null pointer exceptions or out-of-bound array indexes.
- *Verification.* JML specifications can be translated into proof obligations for various theorem provers. The more that properties are explicitly specified the easier it is to prove them.

Finally, the extended use of assertions also improves the software process, as assertions establish a contract between developers—or, more generally, between

BUG STORY 10

The Lufthansa A320 Accident in Warsaw

On September 14, 1993, a Lufthansa A320 landed at Warsaw airport in a thunderstorm. The landing appeared to be normal and smooth, albeit somewhat fast. The pilots were unable to activate any of the braking mechanisms (spoilers, reverse thrust, wheel brakes) for 9 seconds after touchdown, at which point the spoilers and reverse thrust deployed. The wheel brakes finally became effective 13 seconds after touchdown. The aircraft was by this time way too far along the runway to stop before the runway end. It ran off the end, and over an earth bank near the end of the runway, before stopping. The first officer died in the accident, as did a passenger who was overcome by smoke from the burning aircraft.

The investigation of the accident found that the aircraft logics prohibited actuation of reverse thrust unless the shock absorbers were compressed at *both* main landing gears. At Warsaw, due to windshear, the shock absorbers of one landing gear were not compressed sufficiently. The aircraft software would not allow actuation of reverse thrust, in compliance with its specification.

As a consequence, Lufthansa had concluded there was a problem with the specification, and was talking with Airbus on a change in the braking logic to reduce the weight-on-wheels load criterion from 12 metric tons to 2 metric tons. In the meantime, Lufthansa required their pilots to land relatively hard in such weather and runway conditions, thus compressing the shock absorbers and "fooling" the specification.

clients and suppliers of software. As the contract is unambiguous and complete, it allows for blame assignment. Rather than discussing who is wrong, one can immediately focus on making the program conform to its specification. Overall, few techniques are as helpful for debugging as assertions, and no other technique has as many additional benefits. For every program, there is every reason to use assertions lavishly.

By definition, specifications guarantee correctness. However, they do not protect against *surprises*—simply because the specification does not match what is actually desired. The accident of a Lufthansa A320 (see Bug Story 10) in Warsaw is a tragic example of a situation in which everything performed according to the specification—a specification that was in error and had to be altered after the accident. Therefore, be sure to have all of your assertions (and specifications) *carefully reviewed*. Do not fall into the trap of adapting your code to a faulty specification.

10.7 REFERENCE RUNS

In some cases, the correct behavior of a program P_1 is not specified explicitly but by *reference* to another program P_0, the behavior of which is defined as "correct."

LIST 10.1: Sources of Reference Runs

The program has been modified. After a change to the program, we want to preserve the original behavior. The old version of the program thus becomes the reference version P_0, and the central behavior of the new version P_1 must not differ from the central behavior of P_0.

As an example, consider a program P_0 where a security leak has been detected. Before you release a patched version P_1, you would like to ensure that P_1 behaves like P_0 in every aspect—except for the security leak, of course.

The environment has changed. After a larger change to the environment, we want to ensure that the old behavior is also present in the new environment. Therefore, P_0 becomes the program in the old environment, and P_1 is the program in the new environment.

The most famous example of this situation is the year 2000 problem, where several programs had to deal with the coming of the new century. If a system in the simulated year 2000 to the current year 1999 showed any differences (except for the date, that is), the system would have a defect.

The program has been ported. After moving a program from one machine to another, we want to ensure that the program P_1 on the new machine behaves like P_0 on the old machine. System architectures and environments have many differences that can impact the program's behavior. In addition to all of the possible influences listed in Chapter 4, changes in data representation, memory organization, or simply different versions of the used libraries and tools can all induce differing, and sometimes incorrect, behavior.

The program has been cloned. The program P_1 may be a complete reimplementation of P_0—for instance, because the source code of P_0 is not (or no longer) available, or because one needs a component P_1 that acts like P_0 for the purpose of interoperability.

Suppose I wanted to write a PERL compiler—that is, a tool that translates PERL programs into machine code. To ensure correctness of my compiler, I have to compare the behavior of the compiled programs with their behavior when being executed by the original PERL interpreter. (Actually, this is how PERL's semantics are defined: by referring to the implementation of the PERL interpreter.)

This means that P_1 is supposed to behave like P_0 in some central aspect, or even in every aspect. List 10.1 outlines a number of situations in which this happens. The most common is that P_1 is a new revision or variant of the original P_0.

Reference programs are mainly used as *oracles*—that is, as universal devices that tell us what is correct, right, or true. For instance, consider *testing*. Any testing method needs to assess whether the result produced by a program is correct or not. Using a reference program P_0, one can simply *compare* against the result of the reference program. This is a common scenario for *regression testing*. We feed P_0 and P_1 with the same input. Any unexpected difference in output is a potential error—or at least an anomaly to keep in mind for further investigation.

During *debugging*, we frequently need to tell whether some program state is infected or not. Having a reference program P_0 as an oracle, we can compare its

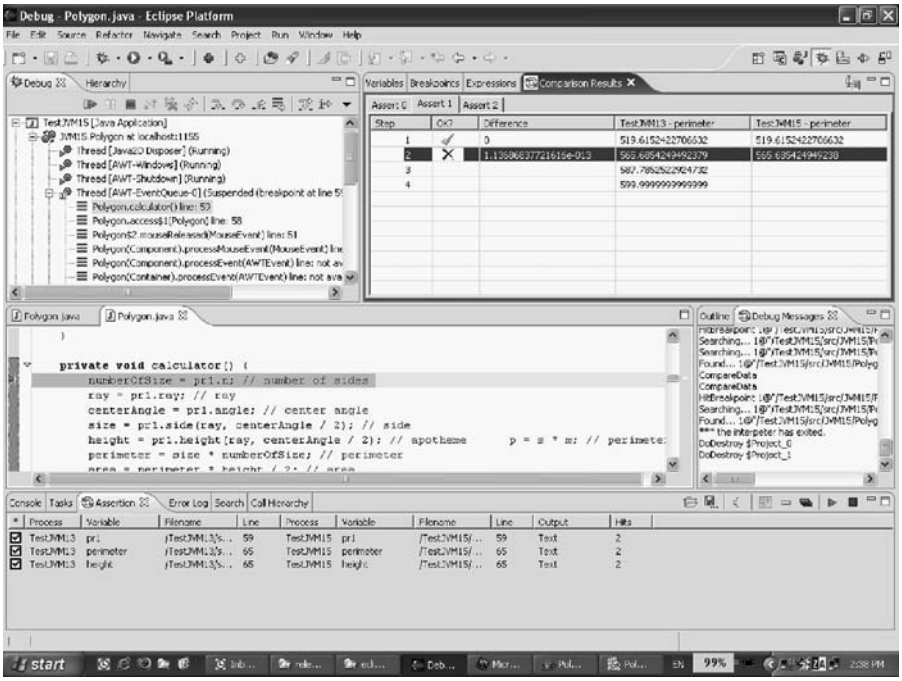

FIGURE 10.3

The GUARD relative debugger. The assertion window highlights differences between the states of two program runs.

state against the debugged program P_1—and again, any difference is a potential infection or anomaly.

Typically, such a comparison is done by having two interactive debuggers run in parallel, with the programmer comparing the individual results. This is a tedious and error-prone process. A better alternative is to *automate* the comparison. This concept has been explored under the name of *relative debugging*—that is, debugging a program relative to a reference version. The key idea of relative debugging is to execute P_0 and P_1 *in parallel*—and flagging any difference between the program states.

As an example, Figure 10.3 shows the relative debugger GUARD debugging a JAVA `Polygon` program under two JAVA virtual machines: the reference P_0 (using JVM 1.3) and its variant P_1, using JVM 1.5. Both processes can be stepped through individually. However, specific variables (such as `perimeter`) can be set up to be compared.

How does GUARD know when and where to compare variable values? This is specified by the programmer. Essentially, the programmer sets up a *relative assertion*—an assertion that two variables have the same value at a specific location. For instance, the GUARD assertion

```
assert p1::perimeter@Polygon.java:65 == p0::perimeter@Polygon.java:65
```

ensures that in process `p1` the value of the variable `perimeter` at the location `Polygon.java:65` is equal to the value of the same variable at the same location in process `p0`. GUARD evaluates every relative assertion as execution reaches the locations, and flags an error if it is violated—very much like the GDB assertion in Section 10.3.

The `Comparison Results` window shows the result of this comparison. In the first step, the `perimeter` variable was identical in both programs. In the second step, though, GUARD found a difference (the relative assertion has failed). The `perimeter` values differ by 1.13. Because JVM 1.3 is the reference, the JVM 1.5 variant is wrong here.

In practice, the two programs being compared may differ in more than just their control flow. Their data structures may also be organized differently. For instance, one implementation may choose to store elements in a tree, whereas another chooses a hash table. Therefore, GUARD lets the user define individual comparison functions that can compare using a common abstraction, such as sets.

All in all, relative debugging exploits the existence of a reference run in a classical interactive debugging session. The more of the state and the run covered by relative assertions the easier it will be to catch infections early. Best results are achieved when porting an otherwise identical program from one environment to another.

10.8 SYSTEM ASSERTIONS

Some properties of a program must hold *during the entire execution.* Good operating systems take care that a program does not access or change the data of other processes, that mathematical exceptions do not go by unnoticed, and that a program stays within the limits set by its privileges. One could think of these properties as *invariants* that are continuously checked at runtime.

In addition to increasing security for the user, such properties are immensely useful for debugging, as they limit the scope of the search. Assume I experience some memory error on a machine where the individual processes are not clearly separated. In such a case, I must extend my search to all processes that ran in conjunction with my program—a situation that is difficult to reproduce and to debug.

Even within a single process, though, it is advisable to have certain properties that are guaranteed during the entire run. The most important of these properties is the *integrity of the program data.* If the fundamental techniques for accessing memory no longer work, it becomes difficult to isolate individual failure causes.

In C and C++ programs, misuse of the *heap* memory is a common source of errors. In C and C++, the heap is a source for memory. This is the place where new objects are allocated. If an object is no longer required, the appropriate heap memory must be deallocated (or *freed*) explicitly. The memory thus becomes available for other objects.

The programmer must take care, though, that deallocated memory is no longer used. In addition, deallocated memory must not be deallocated again. Both actions result in *time bombs*—faults that manifest themselves only millions of instructions later and are thus difficult to isolate.

Fortunately, a number of useful tools help validate the state of the heap. It is a good idea to always have these tools ready during development, and to apply them at the slightest suspicion. It makes little sense to reason about individual variable values if the structure of the heap is not sound. The catch of these tools is that they increase memory and time requirements and thus cannot be used in production code.

10.8.1 Validating the Heap with `MALLOC_CHECK_`

Using the GNU C runtime library (default on Linux systems), one can avoid common errors related to heap use simply by setting an environment variable called `MALLOC_CHECK_`. For instance, one can detect multiple deallocation of heap memory.

```
$ MALLOC_CHECK_=2 ./myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
```

The `core` file generated at program abort can be read in by a debugger, such that one is directly led to the location where `free()` was called the second time. This postmortem debugging was discussed in Section 8.3.3 in Chapter 8.

10.8.2 Avoiding Buffer Overflows with ELECTRICFENCE

The ELECTRICFENCE library effectively prohibits buffer overflows. Its basic idea is to allocate arrays in memory such that each array is preceded and followed by a nonexisting memory area—the actual "electric fence." If the program attempts to access this area (i.e., an overflow occurred), the operating system aborts the program.

Using ELECTRICFENCE, one can quickly narrow down the overflowing array in `sample` (Example 1.1). We compile `sample` using the `efence` library and call the resulting `sample-with-efence` program with two arguments. As soon as `a[2]` is accessed, the program is aborted.

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ ./sample-with-efence 11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```

Again, the `core` file can be read in by a debugger—unless one runs `sample-with-efence` directly within the debugger.

10.8.3 Detecting Memory Errors with VALGRIND

VALGRIND (named after the holy entrance to Valhalla, the home of Odin) provides the functionality of ELECTRICFENCE, plus a little more. VALGRIND detects:

- Read access to noninitialized memory
- Write or read access to nonallocated memory
- Write or read access across array boundaries
- Write or read access in specific stack areas
- Detection of memory leaks (areas that were allocated but never deallocated)

If we apply VALGRIND to the `sample` program from Example 1.1, we obtain a message stating that `sample` accesses memory in an illegal manner. This access takes place in `shell_sort()` (line 18), called by `main` and `__libc_start_main`.

```
$ valgrind sample 11 14
Invalid read of size 4
  at 0x804851F: shell_sort (sample.c:18)
  by 0x8048646: main (sample.c:35)
  by 0x40220A50: __libc_start_main (in /lib/libc-2.3.so)
  by 0x80483D0: (within /home/zeller/sample)
```

The remaining message gives some details about the invalid memory area. It is close to the memory area allocated by `main` (line 32)—the memory area `malloc`'ed for `a[0 ... 1]`.

```
  Address 0x40EE902C is 0 bytes after a block alloc'd
  at 0x4015D414: malloc (vg_clientfuncs.c:103)
  by 0x80485D9: main (sample.c:32)
  by 0x40220A50: __libc_start_main (in /lib/libc-2.3.so)
  by 0x80483D0: (within /home/zeller/sample)
$ _
```

How does this work? VALGRIND is built around an *interpreter* for x86 machine code instructions. It interprets the machine instructions of the program to be debugged, and keeps track of the used memory in so-called *shadow memory*.

- Each memory bit is associated with a controlling *value bit* (V-bit). Each V-bit is initially unset. VALGRIND sets it as soon as the associated memory bit is being written.
- In addition, each byte is associated with an *allocated bit* (A-bit), which is set if the corresponding byte is currently allocated. When some memory area is deallocated, VALGRIND clears the A-bits.

Whenever the program tries to read some memory with A-bits or V-bits that are not set, VALGRIND flags an error.

Figure 10.4 shows the situation in which VALGRIND generates the previous error message for the `sample` program: `a[0]` and `a[1]` are allocated and initialized—their A- and V-bits set (shown in gray). In contrast, `a[2]` is neither allocated nor initialized. Accessing it causes VALGRIND to issue an error message.

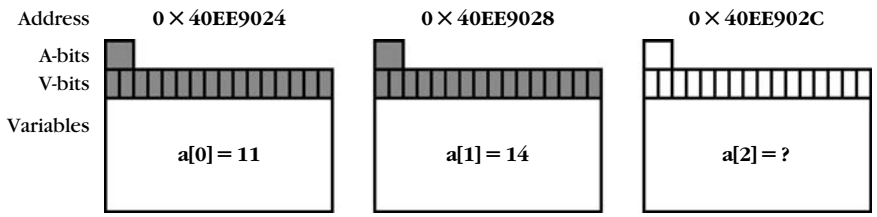

FIGURE 10.4

A- and V-bits in VALGRIND. A-bits are set if the associated byte is allocated; V-bits are set if the associated bit has been written.

Using VALGRIND is not without drawbacks. The code size can increase up to 12 times, and execution times can increase up to 25 times. Memory usage doubles due to shadow memory. A more efficient way is not to interpret the machine code but to *instrument it*—that is, to include extra code at memory accesses to update and check shadow memory. This approach is realized in the PURIFY tool, which detects the same class of errors as VALGRIND but in a more effective way. Programs instrumented with PURIFY have a typical slowdown factor of only 5 to 10. (PURIFY also comes with a nice GUI as well as phone support from IBM.)

The relatively low overhead is, in general, acceptable for debugging purposes. Most of the Linux KDE project, for instance, is checked with VALGRIND. PURIFY has a long record of successfully checking arbitrary programs. Experienced programmers routinely validate the heap integrity with VALGRIND, PURIFY, or similar tools—just for easy elimination of a failure source and an extra ounce of confidence.

There is one point where VALGRIND is different from PURIFY and other memory checkers. VALGRIND acts as a general *framework*, where VALGRIND *plug-ins* can execute arbitrary code while the original program is interpreted. This allows for much more than just memory check. In fact, memory checking is just one of VALGRIND's plug-ins. For instance, it is easy to write a VALGRIND plug-in that logs the current execution position, in a fashion similar to PIN (see Section 8.2.4 in Chapter 8). The DAIKON tool, discussed in Section 11.5 in Chapter 11, uses a specialized VALGRIND plug-in to capture execution traces of Linux binaries.

10.8.4 Language Extensions

At this point, one may wonder why we might bother with system assertions at all. Shouldn't one simply switch to a programming language in which such problems do not occur? Indeed, languages with managed memory and garbage collection, such as JAVA or C#, do not suffer from the memory problems described in this section. A more conservative migration path from low-level languages is to use a *safer dialect* of an existing programming language. Such a dialect brings *extensions* that allow programmers to specify further properties of language entities. These extensions can then be checked (at runtime, but also statically) to catch errors early.

As an example, consider CYCLONE, a safer dialect of the *C* programming language. CYCLONE's central extension to *C* is the concept of *special pointers*—that is, *C* pointers with special properties. For instance, in CYCLONE one can declare a pointer that can never be `NULL` by using `@` instead of `*`. The `getc()` function, retrieving a character from a file, is thus declared as

```
int getc (FILE @);
```

Calling `getc()` with a potentially `NULL` pointer results in a runtime check being triggered: If `getc()` is called with a `NULL` pointer, the runtime check will terminate the program rather than having `getc()` fail in an uncontrolled fashion. In addition, the CYCLONE compiler will give a warning about having inserted such a runtime check.

```
extern FILE *f;
char c = getc (f); // warning: NULL check inserted
```

Another interesting CYCLONE feature is *fat pointers*—pointers that not only record a location but bound information (such as the size of the area being pointed to). Such a pointer is declared by using `?` instead of `*`. Using fat pointers, a function such as `strlen()`, determining the length of a string, can be declared as

```
int strlen (const char? s);
```

In contrast to the original `strlen()` function, the CYCLONE implementation need not scan the string for a terminating `NUL` character. Instead, it can access the bounds of the string `s` using `s.size`. This also implies that, unlike the original C variant, the CYCLONE version will not scan past the end of strings that lack a `NUL` terminator. All memory accesses via a fat pointer will have bounds automatically checked at runtime. If a violation can be detected at compile time already, the CYCLONE compiler will flag an error.

To detect errors at compile time, CYCLONE imposes a number of restrictions on *C* programs (List 10.2)—restrictions that effectively deal with the caveats outlined in List 7.3 and that still enable CYCLONE to support low-level programming. All in all, a few extensions suffice to make C-style programming almost as safe as programming in JAVA, or other high-level languages, and to prevent memory misuse as described in this section.

10.9 CHECKING PRODUCTION CODE

We have now seen that the computer can automate much of the observation for us, and that large parts of the program state can be checked during execution. This is helpful in debugging and increases our confidence in the program. When it comes to *releasing* the program, though, should we still have all checks enabled? First, the following are some checks that should *never* be turned off.

Critical results. If your program computes a result that people's lives, health, or money depends on, it is a good idea to validate the result using some additional

LIST 10.2: Restrictions Imposed by CYCLONE to Preserve Safety

- `NULL` checks are inserted to prevent segmentation faults.
- Pointer arithmetic is restricted.
- Pointers must be initialized before use.
- Dangling pointers are prevented through region analysis and limitations on `free()`.
- Only "safe" casts and unions are allowed.
- `goto` into scopes is disallowed.
- `switch` labels in different scopes are disallowed.
- Pointer-returning functions must execute `return`.
- `setjmp()` and `longjmp()` are not supported.

Source: Jim et al. (2002).

computation. As a lesser alternative, one may also use *n-version programming*—that is, one computes the result a second time (using an alternate implementation or algorithm) and compares the results automatically.

Obviously, an assertion is not the best way of checking critical results, in that an assertion can be turned off—and you do not want to turn off warnings on critical results.

External conditions. Any conditions that are not within our control must be checked for integrity. This especially holds for external input, which must be verified to satisfy all syntactic and semantic constraints. In case of error, the user must be notified.

Again, an assertion is not the right way to check external conditions. Think of an assertion that checks whether the length of the input stays within the buffer length, for instance. Turning off such an assertion results in a security leak. Furthermore, the input to a program is typically under the control of the user, and when the user makes a mistake it is better to tell him or her "A PIN has exactly four digits" rather than to have the program abort with a message such as `assertion 'length == 4' failed`.

What do we do with the other assertions in production code, though? The following are some arguments to consider:

The more active assertions there are, the greater the chances of catching infections. Because not every infection need result in a failure, assertions increase your chances of detecting defects that would otherwise go by unnoticed. Therefore, assertions should remain turned on.

The sooner a program fails the easier it is to track the defect. The larger the distance between defect and failure, the more difficult it is to track the infection chain. The more active assertions there are, the sooner an infection will be caught, which significantly eases debugging. This idea of making code "fail fast" is an argument for leaving assertions turned on.

Defects that escape into the field are the most difficult to track. Remember that failures that occur at the user's site are often difficult to reproduce (see Chapter 4). Failing assertions can give the essential clues on how the infection spread.

By default, failing assertions are not user friendly. The message from a failing assertion may be helpful for programmers but will appear cryptic to most users—and the fact that the program simply aborts (which is the default) is not what you would call a helpful behavior.

However, this is not yet a reason to turn off assertions. An unnoticed incorrect behavior is far more dangerous than a noticed aborted behavior. When something bad may happen, do not shoot the messenger (and turn assertions off), but make sure the program *gracefully fails.* For instance, a global exception handler could state that a fatal error occurred and offer some means of recovery.

Assertions impact performance. This argument is true, but should be considered with respect to the benefits of assertions. As with every performance issue, one must first detect how much performance is actually lost. Only if this amount is intolerable one should specifically check for *where* the performance is lost.

An assertion executed several times as an invariant, for instance, may impact performance far more than a single postcondition executed once at the end of a complex function. Regarding performance, it is thus a wise strategy to turn off those assertions that do have an impact on performance (as proven by earlier measurements) and to leave on those assertions that prohibit a widespread infection—for instance, those assertions that control the integrity of a result (unless already checked as a critical result).

Note that the current trend in software development is to trade performance for runtime safety wherever possible. JAVA and .NET have introduced the concept of *managed code* and *managed data*, whereby the integrity of both code and data is constantly monitored and verified. Given the security issues in our networked world, and given the continuing explosion of computing power, the cost of checking as much as possible becomes more and more irrelevant when compared to the risk of not checking enough. Eventually, *proving correctness* may turn out to be a strategy for optimization. If it can be proven that an assertion always holds, it can easily be eliminated.

10.10 CONCEPTS

How To

To automate observation, use assertions.

Assertions catch infections before they can propagate through the program state and cover their origins.

Like observation statements, assertions must not interfere with the actual computation.

To use assertions, check preconditions, data invariants, and postconditions:

- *Preconditions* document the requirements for calling a function. A successful check means the requirements are met.
- *Data invariants* specify properties over data that hold before and after each public function that operates on that data. A successful check means sane data.
- *Postconditions* describe the effects of a function. A successful check means correctness of the result.

Assertions can serve as specifications (as in EIFFEL or JML) and thus document interfaces.

In contrast to "external" specification languages, assertions are interwoven with the code and can be easily checked at runtime.

Rich specification languages such as JML provide a smooth transition from assertions (checked at runtime) to static checking and verification (checked at compile time).

To check a program against a reference program, use relative debugging.

To check memory integrity, use specialized tools to detect errors in memory management. Such tools should be applied before all other methods of debugging.

The most sophisticated tools detect memory misuse by tracking memory usage via *shadow memory.*

To prevent memory errors in a low-level language, consider using a safer dialect such as CYCLONE for the C language.

Use assertions to make your code fail as fast as possible. This increases the chances of catching infections. It also shortens the chain from defect to failure.

Assertions cause a performance loss. You gain benefits for debugging, though, and avoid risks of erroneous computing—advantages that frequently outweigh the costs. Therefore, leave lightweight assertions on in production code—offering a user-friendly recovery from failed assertions.

Do not use assertions for critical results or external conditions. Use hard-coded error handling instead.

10.11 TOOLS

JML. The Iowa State University JML tools include the JML compiler (JMLC), the JMLUNIT unit testing program, and the JMLDOC documentation generator. All are publicly available at *http://www.jmlspecs.org/.*

ESC/JAVA. The ESC/JAVA tool combines static checking with JML. The version that fully supports JML is ESC/JAVA version 2, developed by David Cok and Joe Kiniry.

ESC/JAVA version 2 is available at *http://kind.ucd.ie/products/opensource/ESCJava2/*.

GUARD. The GUARD relative debugger was presented by Sosič and Abramson (1997), who also pioneered the concept of relative debugging. Their web site contains more on the concept as well as the debugger. This is found at *http://www.csse.monash.edu.au/~davida/guard/*.

VALGRIND. The VALGRIND tool for Linux is part of Linux distributions for x86 processors. It is available at *http://www.valgrind.org/*.

PURIFY. PURIFY, marketed by IBM, is also available for Linux/Unix and Windows. Information on PURIFY is available at *http://www.ibm.com/software/awdtools/purify/*.

INSURE++. INSURE++ is a commercial tool that detects memory problems by instrumenting *C* and C++ source code. It is therefore available on a number of platforms. It may be found at *http://www.parasoft.com/*.

CYCLONE. The CYCLONE dialect was developed by Jim et al. (2002). An open-source compiler for Linux can be downloaded at *http://www.research.att.com/projects/cyclone/*.

CCURED. The CCURED language by Necula et al. (2002) takes an approach similar to that of CYCLONE, but moves control from the programmer to the system. For this purpose, it has to extend data representations by *metadata* to enable even better dynamic book-keeping. Such metadata would, for instance, record how pointers are supposed to be used. Condit et al. (2003) describe the use of CCURED on real-world software such as network demons. CCURED, as well as an online demo, are available at *http://manju.cs.berkeley.edu/ccured/*.

10.12 FURTHER READING

Assertions are as old as computers. It is reported that even John von Neumann used them. To see interesting discussions on the use of assertions, have a look at the "People, Projects, and Patterns" WIKI at *http://c2.com/cgi/wiki*. You can contribute, too! The following are some starting points:

- *http://c2.com/cgi/wiki?WhatAreAssertions*
- *http://c2.com/cgi/wiki?UseAssertions*
- *http://c2.com/cgi/wiki?DoNotUseAssertions*

Although people generally believe that assertions are a good thing, there is only one study that has actually conducted controlled experiments to validate this claim. Müller et al. (2002) found that assertions indeed do increase liability and understandability of programs, although requiring a larger programming effort. (Müller et al. did not research the use of assertions for debugging, though.)

The EIFFEL language realized the idea of design by contract pioneered by Meyer (1997). EIFFEL software and voluminous training material are available at *http://www.eiffel.com/*.

To learn more about the *Z* specification language, I recommend *The Way of Z* by Jacky (1996). Few other books on formal methods come close in clarity, precision, and completeness.

JML was originally developed by Leavens et al. (1999). Since then, it has turned into a cooperative effort of dozens of researchers. Burdy et al. (2003) give an overview of JML tools and applications. Leavens and Cheon (2004) give a tutorial on how to use JML as a design-by-contract tool. These papers are available on the JML home page at *http://www.jmlspecs.org/*.

Once one has specified an invariant of a data structure, one can detect any violation. Demsky and Rinard (2003) go one step further and suggest *automatic repair* of data structures with given invariants.

The SPEC# language (spoken "spec sharp") by Barnett et al. (2004) is an extension of C#, providing non-`NULL` types (as in CYCLONE) as well as method contracts (as in JML). It is being used at Microsoft to improve safety. See the project home page at *http://research.microsoft.com/SpecSharp/*.

ESC/Java was developed at the Compaq Systems Research Center by a large group of researchers. To get started with extended static checking in general, see the project home page at *http://kind.ucd.ie/products/opensource/ESCJava2/*.

The VALGRIND tool was initially developed by Julian Seward, and later extended to a framework by Nicholas Nethercote. The paper of Nethercote and Seward (2003) gives several details about the framework.

The concept of having software fail fast is discussed by Shore (2004), using several code examples to illustrate the use of assertions and exception handlers.

Checking the integrity of all input is an important factor in building secure software. See Viega and McGraw (2001) for an introduction to the subject.

N-version programming is a lesser alternative to checking results, because there are serious doubts whether it works. For details, see Knight and Leveson (1986) and Leveson et al. (1990).

Bug Story 10, on the A320 accident, was compiled from "Report of the Main Commission Aircraft Accident Investigation Warsaw," and information from Peter Ladkin, posted to the *Risks Digest* (vol. 15, issue 30) in December 1993. The report is available, along with other accident reports, at *http://sunnyday.mit.edu/accidents/*.

EXERCISES

10.1 Why can an assertion such as `assert(sane())` be used at the beginning and end of public functions but not necessarily at other places?

10.2 What happens if `sane()` is called from `hour()`? How can you improve the `sane()` function?

10.3 Write assertions for the `bigbang` program shown in Figure 8.2:

(a) As invariants for the `Element` and `Container` classes.

(b) As pre- and postconditions for each public method, checking (among others) that the invariants still hold.

Do you catch the infection using these assertions? If so, how? If not, why not?

10.4 Consider the public interface of the JAVA `TreeMap` class. Design an aspect that adds `assert(sane())` to the beginning and end of each public function. Optimize the aspect such that postmethod checks are only issued for methods that can change the state.

10.5 Assume we had a function that could tell us whether the state is sane [say, `state_is_sane()`]. To search a defect, all we would have to do is insert assertions

```
assert(state_is_sane());
```

into the program to narrow down the infection site by a simple binary search, which could even be automated. However, if we had such a function we would not have to search the defect anyway. Why?

10.6 Assume the program state consists only of objects the sanity of which is guaranteed by (invariant) assertions. Can we assume that the entire state is sane?

10.7 Rather than writing an assertion such as

```
assert(0 <= h && h <= 23);
```

I can use GDB to check the following condition.

```
(gdb) break 'Time::set_hour(int)' if h < 0 || h > 23
Breakpoint 3 at 0x2dcf: file Time.C, line 45.
(gdb) _
```

Discuss the pros and cons of this technique.

10.8 Consider the `BinaryTree` class shown in Example 10.3. Write some aspects for the following tasks.

(a) A logging aspect that logs every entry and every exit of a method from the `BinaryTree` class. This aspect shall log which method is entered or left.

(b) A logging aspect that displays every setting of the left or the right child of a `BinaryTree`.

10.9 For the `BinaryTree` class shown in Example 10.3, write some JML invariants that hold for a tree node per the following:

- The key is a nonnegative number.

EXAMPLE 10.3: The `BinaryTree.java` program

```
 1  class BinaryTree {
 2      private int key;
 3      private Object value;
 4      private BinaryTree right;
 5      private BinaryTree left;
 6
 7      public BinaryTree(int _key,Object _value) {
 8          key   = _key;
 9          value = _value;
10          right = left = null;
11      }
12
13      // Lookup a node with a specific key
14      public Object lookup(int _key) {
15          BinaryTree descend;
16          if (_key == key)
17              return value;
18          if (_key < key)
19              descend = left;
20          else
21              descend = right;
22          if (descend == null)
23              return null;
24          return descend.lookup(_key);
25      }
26
27      // Insert a node with a certain key and value
28      public void insert(int _key, Object _value) {
29          if (_key <= key)
30              if (left == null)
31                  left = new BinaryTree(_key,_value);
32              else
33                  left.insert(_key,_value);
34          else
35              if (right == null)
36                  right = new BinaryTree(_key,_value);
37              else
38                  right.insert(_key,_value);
39      }
40
41      // Delete a node with a certain key
42      public boolean delete (int key) {
43          // ...
44          return true;
45      }
46  }
```

- The keys of all left children are less than or equal to the node's key. The keys of all right children are greater than or equal to the node's key.
- The data object is not empty.

10.10 Sketch JML assertions for the `insert` method of `BinaryTree` (Example 10.3) that guarantee the following conditions:
- The inserted object is not null.
- The key and the object are not altered during insertion.
- The children of this node contain one further instance of the inserted key/object pair after insertion.

10.11 Consider the following three pieces of code, which sum up the elements in an array `a[]`. First, a PYTHON version:

```
a = read_elems()
sum = 0
for elem in a.elems():
   sum = sum + elem
```

A C version:

```
read_elems(a);
sum = 0;
for (int i = 0; i < n; i++)
    sum += a[i];
```

And finally, a JAVA version:

```
a = read_elems();
sum = 0;
for (Iterator it = a.iterator(); it.hasNext(); )
    sum += it.next();
```

In all three versions, we have a variable `sum` that at the end of the loop holds the sum of all elements in `a[]`. Using relative debugging, which assertions can you set up?

10.12 What are the respective benefits of relative assertions versus standard assertions?

10.13 Some possible points for a program examination are as follows.
- Function `foo()` is reached.
- Variable `z[0]` is equal to 2.
- Function `foo()` is reached and variable `z[0]` is equal to 2.
- Function `foo()` is reached or variable `z[0]` is equal to 2.
- Function `foo()` is reached and at least one of `z[0]`, `z[1]`, …, `z[99]` is equal to 2.
- All of `z[0]`, `z[1]`, …, `z[99]` are equal to 2.

Assume that the processor has no special debugging support, except for changing individual machine instructions into interrupt instructions.

(a) Sort these queries according to the execution speed of the examined program from fastest to slowest.
(b) Sketch processor support that can make these queries more efficient.
(c) Sketch possible code instrumentation (e.g., adding new code at compilation time) to make these queries more efficient.

10.14 Use VALGRIND (or similar heap checker) to check `bigbang` (Example 8.3) for memory problems.

10.15 We observe a `union` in a C program—a data structure in which all members start at the same address and in which only one member is actually used.

```
union node_value {
  char c;   // 1 byte
  int  i;   // 4 bytes
  double d; // 8 bytes
}
```

Your goal is to disambiguate the union—that is, to decide which of the members `c`, `i`, or `d` is actually being used. Discuss the means of doing so at the moment of execution, using VALGRIND's bits. Which bits do you use for which purpose?

10.16 Design a global exception handler that catches failing assertions and gives user-friendly directions on how to handle the situation. (Hint: See Shore (2004) for a discussion.)

> One other obvious way to conserve programmer time is to teach machines how to do more of the low-level work of programming.
>
> – Eric S. Raymond
> *The Art of UNIX Programming* (1999)