Unit Tests

Unit tests verify that a focused, cohesive collection of code—a *unit*, such as a function or a class—behaves exactly as the programmer intended. Good unit tests isolate the unit being tested from its dependencies. Sometimes this can be hard to do: the unit might depend on other units. In such situations, you use mocks to stand in for these dependencies. *Mocks* are fake objects you use solely during testing to provide you with fine-grained control over how a unit's dependencies behave during the test. Mocks can also record how a unit interacted with them, so you can test whether a unit is interacting with its dependencies as expected. You can also use mocks to simulate rare events, such as a system running out of memory, by programming them to throw an exception.

Integration Tests

Testing a collection of units together is called an *integration test*. Integration tests can also refer to testing interactions between software and hardware, which system programmers deal with often. Integration tests are an important layer on top of unit tests, because they ensure that the software you've written works together as a system. These tests complement, but don't replace, unit tests.

Acceptance Tests

Acceptance tests ensure that your software meets all of your customers' requirements. High-performing software teams can use acceptance tests to guide development. Once all of the acceptance tests pass, your software is deliverable. Because these acceptance tests become part of the code base, there is built-in protection against refactoring or feature regression, where you break an existing feature in the process of adding a new one.

Performance Tests

Performance tests evaluate whether software meets effectiveness requirements, such as speed of execution or memory/power consumption. Optimizing code is a fundamentally empirical exercise. You can (and should) have ideas about which parts of your code are causing performance bottlenecks but can't know for sure unless you measure. Also, you cannot know whether the code changes you implement with the intent of optimizing are improving performance unless you measure again. You can use performance tests to instrument your code and provide relevant measures. *Instrumentation* is a technique for measuring product performance, detecting errors, and logging how a program executes. Sometimes customers have strict performance requirements (for example, computation cannot take more than 100 milliseconds or the system cannot allocate more than 1MB of memory). You can automate testing such requirements and make sure that future code changes don't violate them.

Code testing can be an abstract, dry subject. To avoid this, the next section introduces an extended example that lends context to the discussion.

An Extended Example: Taking a Brake

Suppose you're programming the software for an autonomous vehicle. Your team's software is very complicated and involves hundreds of thousands of code lines. The entire software solution is composed of several binaries. To deploy your software, you must upload the binaries into a car (using a relatively time-consuming process). Making a change to your code, compiling, uploading, and executing it in a live vehicle takes several hours per iteration.

The monumental task of writing all the vehicle's software is broken out into teams. Each team is responsible for a *service*, such as the steering wheel control, audio/video, or vehicle detection. Services interact with each other via a service bus, where each service publishes events. Other services subscribe to these events as needed. This design pattern is called a *service bus architecture*.

Your team is responsible for the autonomous braking service. The service must determine whether a collision is about to happen and, if so, tell the car to brake. Your service subscribes to two event types: the `SpeedUpdate` class, which tells you that the car's speed has changed, and the `CarDetected` class, which tells you that some other car has been detected in front of you. Your system is responsible for publishing a `BrakeCommand` to the service bus whenever an imminent collision is detected. These classes appear in Listing 10-1.

```
struct SpeedUpdate {
  double velocity_mps;
};

struct CarDetected {
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};
```

Listing 10-1: The POD classes that your service interacts with

You'll publish the `BrakeCommand` using a `ServiceBus` object that has a `publish` method:

```
struct ServiceBus {
  void publish(const BrakeCommand&);
  --snip--
};
```

The lead architect wants you to expose an `observe` method so you can subscribe to `SpeedUpdate` and `CarDetected` events on the service bus. You decide to build a class called `AutoBrake` that you'll initialize in the program's entry point. The `AutoBrake` class will keep a reference to the `publish` method of the service bus, and it will subscribe to `SpeedUpdate` and `CarDetected` events through its `observe` method, as in Listing 10-2.

```
template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish);
  void observe(const SpeedUpdate&);
  void observe(const CarDetected&);
private:
  const T& publish;
  --snip--
};
```

Listing 10-2: The `AutoBrake` class, which provides the automatic braking service

Figure 10-1 summarizes the relationship between the service bus `ServiceBus`, the automatic braking system `AutoBrake`, and other services.

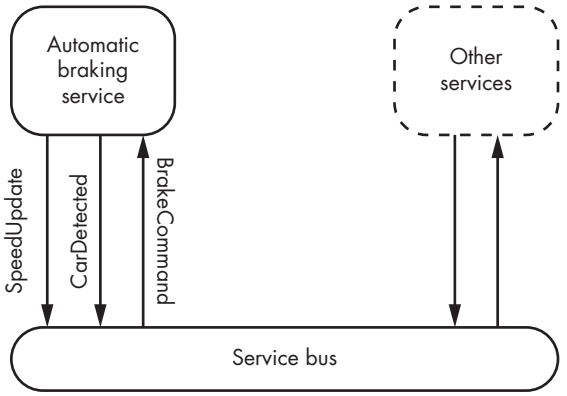

Figure 10-1: A high-level depiction of the interaction between services and the service bus

The service integrates into the car's software, yielding something like the code in Listing 10-3.

```
--snip--
int main() {
  ServiceBus bus;
  AutoBrake auto_brake{ [&bus❶] (const auto& cmd) {
                          bus.publish(cmd); ❷
                        }
  };
  while (true) {  // Service bus's event loop
    auto_brake.observe(SpeedUpdate{ 10L }); ❸
```

```
    auto_brake.observe(CarDetected{ 250L, 25L }); ❹
  }
}
```

Listing 10-3: A sample entry point using your `AutoBrake` service

You construct an `AutoBrake` with a lambda that captures a reference to a `ServiceBus` ❶. All the details of how `AutoBrake` decides when to brake are completely hidden from the other teams. The service bus mediates all interservice communication. You've simply passed any commands from the `AutoBrake` directly to the `ServiceBus` ❷. Within the event loop, a `ServiceBus` can pass `SpeedUpdate` ❸ and `CarDetected` objects ❹ to the `observe` method on your `auto_brake`.

Implementing AutoBrake

The conceptually simple way to implement `AutoBrake` is to iterate among writing some code, compiling the production binary, uploading it to a car, and testing functionality manually. This approach is likely to cause program (and car) crashes and to waste a whole lot of time. A better approach is to write code, compile a unit-test binary, and run it in your desktop development environment. You can iterate among these steps more quickly; once you're reasonably confident that the code you've written works as intended, you can do a manual test with a live car.

The *unit-test binary* will be a simple console application targeting the desktop operating system. In the unit-test binary, you'll run a suite of unit tests that pass specific inputs into an `AutoBrake` and assert that it produces the expected results.

After consulting with your management team, you've collected the following requirements:

- `AutoBrake` will consider the car's initial speed zero.
- `AutoBrake` should have a configurable sensitivity threshold based on how many seconds are forecast until a collision. The sensitivity must not be less than 1 second. The default sensitivity is 5 seconds.
- `AutoBrake` must save the car's speed in between `SpeedUpdate` observations.
- Each time `AutoBrake` observes a `CarDetected` event, it must publish a `BrakeCommand` if a collision is forecasted in less time than the configured sensitivity threshold.

Because you have such a pristine requirements list, the next step is to try implementing the automatic braking service using *test-driven development (TDD)*.

NOTE *Because this book is about C++ and not about physics, your `AutoBrake` only works when a car is directly in front of you.*

Test-Driven Development

At some point in the history of unit-testing adoption, some intrepid software engineers thought, "If I know I'm going to write a bunch of unit tests for this class, why not write the tests first?" This manner of writing software, known as TDD, underpins one of the great religious wars in the software engineering community. Vim or Emacs? Tabs or spaces? To use TDD or not to use TDD? This book humbly abstains from weighing in on these questions. But we'll use TDD because it fits so naturally into a unit-testing discussion.

Advantages of TDD

The process of writing a test that encodes a requirement *before* implementing the solution is the fundamental idea behind TDD. Proponents say that code written this way tends to be more modular, robust, clean, and well designed. Writing good tests is the best way to document your code for other developers. A good test suite is a fully working set of examples that never gets out of sync. It protects against regressions in functionality whenever you add new features.

Unit tests also serve as a fantastic way to submit bug reports by writing a unit test that fails. Once the bug is fixed, it will stay fixed because the unit test and the code that fixes the bug become part of the test suite.

Red-Green-Refactor

TDD practitioners have a mantra: *red, green, refactor*. Red is the first step, and it means to implement a failing test. This is done for several reasons, principal of which is to make sure you're actually testing something. You might be surprised how common it is to accidentally design a test that doesn't make any assertions. Next, you implement code that makes the test pass. No more, no less. This turns the test from red to green. Now that you have working code and a passing test, you can refactor your production code. To refactor means to restructure existing code without changing its functionality. For example, you might find a more elegant way to write the same code, replace your code with a third-party library, or rewrite your code to have better performance characteristics.

If you accidentally break something, you'll know immediately because your test suite will tell you. Then you continue to implement the remainder of the class using TDD. You can work on the collision threshold next.

Writing a Skeleton AutoBrake Class

Before you can write tests, you need to write a *skeleton class*, which implements an interface but provides no functionality. It's useful in TDD because you can't compile a test without a shell of the class you're testing.

Consider the skeleton `AutoBrake` class in Listing 10-4.

```
struct SpeedUpdate {
  double velocity_mps;
```

```
};

struct CarDetected {
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};

template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish❶) : publish{ publish } { }
  void observe(const SpeedUpdate& cd) { } ❷
  void observe(const CarDetected& cd) { } ❸
  void set_collision_threshold_s(double x) { ❹
    collision_threshold_s = x;
  }
  double get_collision_threshold_s() const { ❺
    return collision_threshold_s;
  }
  double get_speed_mps() const { ❻
    return speed_mps;
  }
private:
  double collision_threshold_s;
  double speed_mps;
  const T& publish;
};
```

Listing 10-4: A skeleton `AutoBrake` class

The `AutoBrake` class has a single constructor that takes the template parameter `publish` ❶, which you save off into a `const` member. One of the requirements states that you'll invoke `publish` with a `BrakeCommand`. Using the template parameter `T` allows you to program generically against any type that supports invocation with a `BrakeCommand`. You provide two different observe functions: one for each kind of event you want to subscribe to ❷❸. Because this is just a skeleton class, no instructions are in the body. You just need a class that exposes the appropriate methods and compiles without error. Because the methods return `void`, you don't even need a return statement.

You implement a setter ❹ and getter ❺. These methods mediate interaction with the private member variable `collision_threshold_s`. One of the requirements implies a class invariant about valid values for `collision_threshold_s`. Because this value can change after construction, you can't just use the constructor to establish a class invariant. You need a way to enforce this class invariant throughout the object's lifetime. You can use the setter to perform validation before the class sets a member's value. The getter allows you to read the value of `collision_threshold_s` without permitting modification. It enforces a kind of *external constness*.

Finally, you have a getter for `speed_mps` ❻ with no corresponding setter. This is similar to making `speed_mps` a public member, with the important difference that it would be possible to modify `speed_mps` from an external class if it were public.

Assertions: The Building Blocks of Unit Tests

A unit test's most essential component is the *assertion*, which checks that some condition is met. If the condition isn't met, the enclosing test fails.

Listing 10-5 implements an `assert_that` function that throws an exception with an error message whenever some Boolean `statement` is `false`.

```
#include <stdexcept>
constexpr void assert_that(bool statement, const char* message) {
  if (!statement❶) throw std::runtime_error{ message }; ❷
}

int main() {
  assert_that(1 + 2 > 2, "Something is profoundly wrong with the universe."); ❸
  assert_that(24 == 42, "This assertion will generate an exception."); ❹
}
--------------------------------------------------------------------------------
terminate called after throwing an instance of 'std::runtime_error'
  what():  This assertion will generate an exception. ❹
```

Listing 10-5: A program illustrating `assert_that` (Output is from a binary compiled by GCC v7.1.1.)

The `assert_that` function checks whether the `statement` ❶ parameter is `false`, in which case it throws an exception with the `message` parameter ❷. The first assertion checks that `1 + 2 > 2`, which passes ❸. The second assertion checks that `24 == 42`, which fails and throws an uncaught exception ❹.

Requirement: Initial Speed Is Zero

Consider the first requirement that the car's initial speed is zero. Before implementing this functionality in `AutoBrake`, you need to write a unit test that encodes this requirement. You'll implement the unit test as a function that creates an `AutoBrake`, exercises the class, and makes assertions about the results. Listing 10-6 contains a unit test that encodes the requirement that the initial speed is zero.

```
void initial_speed_is_zero() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} }; ❶
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0"); ❷
}
```

Listing 10-6: A unit test encoding the requirement that the initial speed be zero

You first construct an `AutoBrake` with an empty `BrakeCommand` `publish` function ❶. This unit test is only concerned with the initial value of `AutoBrake`

for car speed. Because this unit test is not concerned with how or when `AutoBrake` publishes a `BrakeCommand`, you give it the simplest argument that will still compile.

NOTE *A subtle but important feature of unit tests is that if you don't care about some dependency of the unit under test, you can just provide an empty implementation that performs some innocuous, default behavior. This empty implementation is sometimes called a* stub.

In `initial_speed_is_zero`, you only want to assert that the initial speed of the car is zero and nothing else ❷. You use the getter `get_speed_mps` and compare the return value to `0`. That's all you have to do; `assert` will throw an exception if the initial speed is zero.

Now you need a way to run the unit tests.

Test Harnesses

A *test harness* is code that executes unit tests. You can make a test harness that will invoke your unit test functions, like `initial_speed_is_zero`, and handle failed assertions gracefully. Consider the test harness `run_test` in Listing 10-7.

```
#include <exception>
--snip--
void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test(); ❶
    printf("[+] Test %s successful.\n", name); ❷
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what()); ❸
  }
}
```

Listing 10-7: A test harness

The `run_test` harness accepts a unit test as a function pointer named `unit_test` and invokes it within a `try-catch` statement ❶. As long as `unit_test` doesn't throw an exception, `run_test` will print a friendly message stating that the unit test passed before returning ❷. If any `exception` is thrown, the test fails and prints a disapproving message ❸.

To make a *unit-test program* that will run all of your unit tests, you place the `run_test` test harness inside the `main` function of a new program. All together, the unit-test program looks like Listing 10-8.

```
#include <stdexcept>

struct SpeedUpdate {
  double velocity_mps;
};

struct CarDetected {
```

```
  double distance_m;
  double velocity_mps;
};

struct BrakeCommand {
  double time_to_collision_s;
};

template <typename T>
struct AutoBrake {
  --snip--
};

constexpr void assert_that(bool statement, const char* message) {
  if (!statement) throw std::runtime_error{ message };
}

void initial_speed_is_zero() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test();
    printf("[+] Test %s successful.\n", name);
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what());
  }
}

int main() {
  run_test(initial_speed_is_zero, "initial speed is 0"); ❶
}
```

```
[-] Test failure in initial speed is 0. speed not equal 0. ❶
```

Listing 10-8: The unit-test program

When you compile and run this unit-test binary, you can see that the unit test `initial_speed_is_zero` fails with an informative message ❶.

NOTE *Because the `AutoBrake` member `speed_mps` is uninitialized in Listing 10-8, this program has undefined behavior. It's not actually certain that the test will fail. The solution, of course, is that you shouldn't write programs with undefined behavior.*

Getting the Test to Pass

To get `initial_speed_is_zero` to pass, all that's required is to initialize `speed_mps` to zero in the constructor of `AutoBrake`:

```
template <typename T>
struct AutoBrake {
```

```
  AutoBrake(const T& publish) : speed_mps{}❶, publish{ publish } { }
  --snip--
};
```

Simply add the initialization to zero ❶. Now, if you update, compile, and run the unit-test program in Listing 10-8, you're greeted with more pleasant output:

```
[+] Test initial speed is 0 successful.
```

Requirement: Default Collision Threshold Is Five

The default collision threshold needs to be 5. Consider the unit test in Listing 10-9.

```
void initial_sensitivity_is_five() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  assert_that(auto_brake.get_collision_threshold_s() == 5L,
              "sensitivity is not 5");
}
```

Listing 10-9: A unit test encoding the requirement that the initial speed be zero

You can insert this test into the test program, as shown in Listing 10-10.

```
--snip--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
}
--------------------------------------------------------------------------------
[+] Test initial speed is 0 successful.
[-] Test failure in initial sensitivity is 5. sensitivity is not 5.
```

Listing 10-10: Adding the `initial-sensitivity-is-5 test` to the test harness

As expected, Listing 10-10 reveals that `initial_speed_is_zero` still passes and the new test `initial_sensitivity_is_five` fails.

Now, make it pass. Add the appropriate member initializer to `AutoBrake`, as demonstrated in Listing 10-11.

```
template <typename T>
struct AutoBrake {
  AutoBrake(const T& publish)
    : collision_threshold_s{ 5 }, ❶
      speed_mps{},
      publish{ publish } { }
  --snip--
};
```

Listing 10-11: Updating `AutoBrake` to satisfy the collision threshold requirement

The new member initializer ❶ sets `collision_threshold_s` to 5. Recompiling the test program, you can see `initial_sensitivity_is_five` is now passing:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
```

Next, handle the class invariant that the sensitivity must be greater than 1.

Requirement: Sensitivity Must Always Be Greater Than One

To encode the sensitivity validation errors using exceptions, you can build a test that expects an exception to be thrown when `collision_threshold_s` is set to a value less than 1, as Listing 10-12 shows.

```
void sensitivity_greater_than_1() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} };
  try {
    auto_brake.set_collision_threshold_s(0.5L); ❶
  } catch (const std::exception&) {
    return; ❷
  }
  assert_that(false, "no exception thrown"); ❸
}
```

Listing 10-12: A test encoding the requirement that sensitivity is always greater than 1

You expect the `set_collision_threshold_s` method of `auto_brake` to throw an exception when called with a value of 0.5 ❶. If it does, you catch the exception and return immediately from the test ❷. If `set_collision_threshold_s` doesn't throw an exception, you fail an assertion with the message `no exception thrown` ❸.

Next, add `sensitivity_greater_than_1` to the test harness, as demonstrated in Listing 10-13.

```
--snip--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
  run_test(sensitivity_greater_than_1, "sensitivity greater than 1"); ❶
}
--------------------------------------------------------------------------------
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[-] Test failure in sensitivity greater than 1. no exception thrown. ❶
```

Listing 10-13: Adding `set_collision_threshold_s` to the test harness

As expected, the new unit test fails ❶.

You can implement validation that will make the test pass, as Listing 10-14 shows.

```
#include <exception>
--snip--
template <typename T>
struct AutoBrake {
  --snip--
  void set_collision_threshold_s(double x) {
    if (x < 1) throw std::exception{ "Collision less than 1." };
    collision_threshold_s = x;
  }
}
```

Listing 10-14: Updating the `set_collision_threshold` method of `AutoBrake` to validate its input

Recompiling and executing the unit-test suite turns the test green:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
```

Next, you want to make sure that an `AutoBrake` saves the car's speed in between each `SpeedUpdate`.

Requirement: Save the Car's Speed Between Updates

The unit test in Listing 10-15 encodes the requirement that an `AutoBrake` saves the car's speed.

```
void speed_is_saved() {
  AutoBrake auto_brake{ [](const BrakeCommand&) {} }; ❶
  auto_brake.observe(SpeedUpdate{ 100L }); ❷
  assert_that(100L == auto_brake.get_speed_mps(), "speed not saved to 100"); ❸
  auto_brake.observe(SpeedUpdate{ 50L });
  assert_that(50L == auto_brake.get_speed_mps(), "speed not saved to 50");
  auto_brake.observe(SpeedUpdate{ 0L });
  assert_that(0L == auto_brake.get_speed_mps(), "speed not saved to 0");
}
```

Listing 10-15: Encoding the requirement that an `AutoBrake` saves the car's speed

After constructing an `AutoBrake` ❶, you pass a `SpeedUpdate` with `velocity_mps` equal to 100 into its `observe` method ❷. Next, you get the speed back from `auto_brake` using the `get_speed_mps` method and expect it is equal to 100 ❸.

NOTE *As a general rule, you should have a single assertion per test. This test violates the strictest interpretation of this rule, but it's not violating its spirit. All of the assertions are examining the same, cohesive requirement, which is that the speed is saved whenever a `SpeedUpdate` is observed.*

You add the test in Listing 10-15 to the test harness in the usual way, as demonstrated in Listing 10-16.

```
--snip--
int main() {
  run_test(initial_speed_is_zero, "initial speed is 0");
  run_test(initial_sensitivity_is_five, "initial sensitivity is 5");
  run_test(sensitivity_greater_than_1, "sensitivity greater than 1");
  run_test(speed_is_saved, "speed is saved"); ❶
}
```

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[-] Test failure in speed is saved. speed not saved to 100. ❶
```

Listing 10-16: Adding the speed-saving unit test into the test harness

Unsurprisingly, the new test fails ❶. To make this test pass, you implement the appropriate `observe` function:

```
template <typename T>
struct AutoBrake {
  --snip--
  void observe(const SpeedUpdate& x) {
    speed_mps = x.velocity_mps; ❶
  }
};
```

You extract the `velocity_mps` from the `SpeedUpdate` and store it into the `speed_mps` member variable ❶. Recompiling the test binary shows that the unit test now passes:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
```

Finally, you require that `AutoBrake` can compute the correct time to collision and, if appropriate, publish a `BrakeCommand` using the `publish` function.

Requirement: AutoBrake Publishes a BrakeCommand When Collision Detected

The relevant equations for computing times to collision come directly from high school physics. First, you calculate your car's relative velocity to the detected car:

$$\text{Velocity}_{\text{Relative}} = \text{Velocity}_{\text{OurCar}} - \text{Velocity}_{\text{OtherCar}}$$

If your relative velocity is constant and positive, the cars will eventually collide. You can compute the time to such a collision as follows:

$$\text{Time}_{\text{Collision}} = \text{Distance} \,/\, \text{Velocity}_{\text{Relative}}$$

If $\text{Time}_{\text{Collision}}$ is greater than zero and less than or equal to `collision_threshold_s`, you invoke `publish` with a `BrakeCommand`. The unit test in Listing 10-17 sets the collision threshold to 10 seconds and then observes events that indicate a crash.

```
void alert_when_imminent() {
  int brake_commands_published{}; ❶
  AutoBrake auto_brake{
    [&brake_commands_published❷](const BrakeCommand&) {
      brake_commands_published++; ❸
  } };
  auto_brake.set_collision_threshold_s(10L); ❹
  auto_brake.observe(SpeedUpdate{ 100L }); ❺
  auto_brake.observe(CarDetected{ 100L, 0L }); ❻
  assert_that(brake_commands_published == 1, "brake commands published not
one"); ❼
}
```

Listing 10-17: Unit testing for brake events

Here, you initialize the local variable `brake_commands_published` to zero ❶. This will keep track of the number of times that the `publish` callback is invoked. You pass this local variable by reference into the lambda used to construct your `auto_brake` ❷. Notice that you increment `brake_commands_published` ❸. Because the lambda captures by reference, you can inspect the value of `brake_commands_published` later in the unit test. Next, you set `set_collision_threshold` to 10 ❹. You update the car's speed to 100 meters per second ❺, and then you detect a car 100 meters away traveling at 0 meters per second (it is stopped) ❻. The `AutoBrake` class should determine that a collision will occur in 1 second. This should trigger a callback, which will increment `brake_commands_published`. The assertion ❼ ensures that the callback happens exactly once.

After adding to `main`, compile and run to yield a new red test:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[-] Test failure in alert when imminent. brake commands published not one.
```

You can implement the code to make this test pass. Listing 10-18 provides all the code needed to issue brake commands.

```
template <typename T>
struct AutoBrake {
  --snip--
  void observe(const CarDetected& cd) {
    const auto relative_velocity_mps = speed_mps - cd.velocity_mps; ❶
    const auto time_to_collision_s = cd.distance_m / relative_velocity_mps; ❷
    if (time_to_collision_s > 0 &&  ❸
        time_to_collision_s <= collision_threshold_s ❹) {
```

```
      publish(BrakeCommand{ time_to_collision_s }); ❺
    }
  }
};
```

Listing 10-18: Code implementing the braking functionality

First, you calculate the relative velocity ❶. Next, you use this value to compute the time to collision ❷. If this value is positive ❸ and less than or equal to the collision threshold ❹, you publish a `BrakeCommand` ❺.

Recompiling and running the unit-test suite yields success:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test alert when imminent successful.
```

Finally, you need to check that the `AutoBrake` will not invoke `publish` with a `BrakeCommand` if a collision will occur later than `collision_threshold_s`. You can repurpose the `alert_when_imminent` unit test, as in Listing 10-19.

```
void no_alert_when_not_imminent() {
  int brake_commands_published{};
  AutoBrake auto_brake{
    [&brake_commands_published](const BrakeCommand&) {
      brake_commands_published++;
  } };
  auto_brake.set_collision_threshold_s(2L);
  auto_brake.observe(SpeedUpdate{ 100L });
  auto_brake.observe(CarDetected{ 1000L, 50L });
  assert_that(brake_commands_published == 0 ❶, "brake command published");
}
```

Listing 10-19: Testing that the car doesn't issue a `BrakeCommand` if a collision isn't anticipated within the collision threshold

This changes the setup. Your car's threshold is set to 2 seconds with a speed of 100 meters per second. A car is detected 1,000 meters away traveling 50 meters per second. The `AutoBrake` class should forecast a collision in 20 seconds, which is more than the 2-second threshold. You also change the assertion ❶.

After adding this test to `main` and running the unit-test suite, you have the following:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test alert when imminent successful.
[+] Test no alert when not imminent successful. ❶
```

For this test case, you already have all the code needed for this test to pass ❶. Not having a failing test at the outset bends the red, green, refactor mantra, but that's okay. This test case is closely related to `alert_when_imminent`. The point of TDD is not dogmatic adherence to strict rules. TDD is a set of reasonably loose guidelines that helps you write better software.

Adding a Service-Bus Interface

The `AutoBrake` class has a few dependencies: `CarDetected`, `SpeedUpdated`, and a generic dependency on some `publish` object callable with a single `BrakeCommand` parameter. The `CarDetected` and `SpeedUpdated` classes are plain-old-data types that are easy to use directly in your unit tests. The `publish` object is a little more complicated to initialize, but thanks to lambdas, it's really not bad.

Suppose you want to refactor the service bus. You want to accept a `std::function` to subscribe to each service, as in the new `IServiceBus` interface in Listing 10-20.

```
#include <functional>

using SpeedUpdateCallback = std::function<void(const SpeedUpdate&)>;
using CarDetectedCallback = std::function<void(const CarDetected&)>;

struct IServiceBus {
  virtual ~IServiceBus() = default;
  virtual void publish(const BrakeCommand&) = 0;
  virtual void subscribe(SpeedUpdateCallback) = 0;
  virtual void subscribe(CarDetectedCallback) = 0;
};
```

Listing 10-20: The `IServiceBus` interface

Because `IServiceBus` is an interface, you don't need to know the implementation details. It's a nice solution because it allows you to do your own wiring into the service bus. But there's a problem. How do you test `AutoBrake` in isolation? If you try to use the production bus, you're firmly in integration-test territory, and you want easy-to-configure, isolated unit tests.

Mocking Dependencies

Fortunately, you don't depend on the implementation: you depend on the interface. You can create a mock class that implements the `IServiceBus` interface and use this within `AutoBrake`. A mock is a special implementation that you generate for the express purpose of testing a class that depends on the mock.

Now when you exercise `AutoBrake` in your unit tests, `AutoBrake` interacts with the mock rather than the production service bus. Because you have complete control over the mock's implementation and the mock is a

unit-test-specific class, you have major flexibility in how you can test classes that depend on the interface:

- You can capture arbitrarily detailed information about how the mock gets called. This can include information about the parameters and the number of times the mock was called, for example.
- You can perform arbitrary computation in the mock.

In other words, you have complete control over the inputs and the outputs of the dependency of `AutoBrake`. How does `AutoBrake` handle the case where the service bus throws an out-of-memory exception inside of a `publish` invocation? You can unit test that. How many times did `AutoBrake` register a callback for `SpeedUpdates`? Again, you can unit test that.

Listing 10-21 presents a simple mock class you can use for your unit tests.

```
struct MockServiceBus : IServiceBus {
  void publish(const BrakeCommand& cmd) override {
    commands_published++; ❶
    last_command = cmd; ❷
  }
  void subscribe(SpeedUpdateCallback callback) override {
    speed_update_callback = callback; ❸
  }
  void subscribe(CarDetectedCallback callback) override {
    car_detected_callback = callback; ❹
  }
  BrakeCommand last_command{};
  int commands_published{};
  SpeedUpdateCallback speed_update_callback{};
  CarDetectedCallback car_detected_callback{};
};
```

Listing 10-21: A definition of `MockServiceBus`

The `publish` method records the number of times a `BrakeCommand` is published ❶ and the `last_command` that was published ❷. Each time `AutoBrake` publishes a command to the service bus, you'll see updates to the members of `MockServiceBus`. You'll see in a moment that this allows for some very powerful assertions about how `AutoBrake` behaved during a test. You save the callback functions used to subscribe to the service bus ❸❹. This allows you to simulate events by manually invoking these callbacks on the mock object.

Now, you can turn your attention to refactoring `AutoBrake`.

Refactoring AutoBrake

Listing 10-22 updates `AutoBrake` with the minimum changes necessary to get the unit-test binary compiling again (but not necessarily passing!).

```
#include <exception>
--snip--
struct AutoBrake { ❶
```

```
  AutoBrake(IServiceBus& bus) ❷
    : collision_threshold_s{ 5 },
      speed_mps{} {
  }
  void set_collision_threshold_s(double x) {
    if (x < 1) throw std::exception{ "Collision less than 1." };
    collision_threshold_s = x;
  }
  double get_collision_threshold_s() const {
    return collision_threshold_s;
  }
  double get_speed_mps() const {
    return speed_mps;
  }
private:
  double collision_threshold_s;
  double speed_mps;
};
```

Listing 10-22: A refactored `AutoBrake` skeleton taking an `IServiceBus` reference

Notice that all the `observe` functions have been removed. Additionally, `AutoBrake` is no longer a template ❶. Rather, it accepts an `IServiceBus` reference in its constructor ❷.

You'll also need to update your unit tests to get the test suite compiling again. One TDD-inspired approach is to comment out all the tests that are not compiling and update `AutoBrake` so all the failing unit tests pass. Then, one by one, uncomment each unit test. You reimplement each unit test using the new `IServiceBus` mock, then update `AutoBrake` so the tests pass.

Let's give it a try.

Refactoring the Unit Tests

Because you've changed the way to construct an `AutoBrake` object, you'll need to reimplement every test. The first three are easy: Listing 10-23 just plops the mock into the `AutoBrake` constructor.

```
void initial_speed_is_zero() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus }; ❷
  assert_that(auto_brake.get_speed_mps() == 0L, "speed not equal 0");
}

void initial_sensitivity_is_five() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus }; ❷
  assert_that(auto_brake.get_collision_threshold_s() == 5,
              "sensitivity is not 5");
}

void sensitivity_greater_than_1() {
  MockServiceBus bus{}; ❶
  AutoBrake auto_brake{ bus }; ❷
```

```
  try {
    auto_brake.set_collision_threshold_s(0.5L);
  } catch (const std::exception&) {
    return;
  }
  assert_that(false, "no exception thrown");
}
```

Listing 10-23: Reimplemented unit-test functions using the `MockServiceBus`

Because these three tests deal with functionality not related to the service bus, it's unsurprising that you didn't need to make any major changes to `AutoBrake`. All you need to do is create a `MockServiceBus` ❶ and pass it into the `AutoBrake` constructor ❷. Running the unit-test suite, you have the following:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
```

Next, look at the `speed_is_saved` test. The `AutoBrake` class no longer exposes an `observe` function, but because you've saved the `SpeedUpdateCallback` on the mock service bus, you can invoke the callback directly. If `AutoBrake` subscribed properly, this callback will update the car's speed, and you'll see the effects when you call the `get_speed_mps` method. Listing 10-24 contains the refactor.

```
void speed_is_saved() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };

  bus.speed_update_callback(SpeedUpdate{ 100L }); ❶
  assert_that(100L == auto_brake.get_speed_mps(), "speed not saved to 100"); ❷
  bus.speed_update_callback(SpeedUpdate{ 50L });
  assert_that(50L == auto_brake.get_speed_mps(), "speed not saved to 50");
  bus.speed_update_callback(SpeedUpdate{ 0L });
  assert_that(0L == auto_brake.get_speed_mps(), "speed not saved to 0");
}
```

Listing 10-24: Reimplemented `speed_is_saved` unit-test function using the `MockServiceBus`

The test didn't change too much from the previous implementation. You invoke the `speed_update_callback function` stored on the mock bus ❶. You make sure that the `AutoBrake` object updated the car's speed correctly ❷. Compiling and running the resulting unit-test suite results in the following output:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[-] Test failure in speed is saved. bad function call.
```

Recall that the `bad function call` message comes from the `std::bad_function_call` exception. This is expected: you still need to subscribe from `AutoBrake`, so `std::function` throws an exception when you invoke it.

Consider the approach in Listing 10-25.

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 },
    speed_mps{} {
    bus.subscribe([this](const SpeedUpdate& update) {
      speed_mps = update.velocity_mps;
    });
  }
  --snip--
}
```

Listing 10-25: Subscribing the `AutoBrake` to speed updates from the `IServiceBus`

Thanks to `std::function`, you can pass your callback into the subscribe method of `bus` as a lambda that captures `speed_mps`. (Notice that you don't need to save a copy of `bus`.) Recompiling and running the unit-test suite yields the following:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
```

Next, you have the first of the alert-related unit tests, `no_alert_when_not_imminent`. Listing 10-26 highlights one way to update this test with the new architecture.

```
void no_alert_when_not_imminent() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  auto_brake.set_collision_threshold_s(2L);
  bus.speed_update_callback(SpeedUpdate{ 100L }); ❶
  bus.car_detected_callback(CarDetected{ 1000L, 50L }); ❷
  assert_that(bus.commands_published == 0, "brake commands were published");
}
```

Listing 10-26: Updating the `no_alert_when_not_imminent` test with the `IServiceBus`

As in the `speed_is_saved` test, you invoke the callbacks on the `bus` mock to simulate events on the service bus ❶❷. Recompiling and running the unit-test suite results in an expected failure.

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[-] Test failure in no alert when not imminent. bad function call.
```

You need to subscribe with `CarDetectedCallback`. You can add this into the `AutoBus` constructor, as demonstrated in Listing 10-27.

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 },
    speed_mps{} {
    bus.subscribe([this](const SpeedUpdate& update) {
      speed_mps = update.velocity_mps;
    });
    bus.subscribe([this❶, &bus❷](const CarDetected& cd) {
      const auto relative_velocity_mps = speed_mps - cd.velocity_mps;
      const auto time_to_collision_s = cd.distance_m / relative_velocity_mps;
      if (time_to_collision_s > 0 &&
          time_to_collision_s <= collision_threshold_s) {
        bus.publish(BrakeCommand{ time_to_collision_s }); ❸
      }
    });
  }
  --snip--
}
```

Listing 10-27: An updated `AutoBrake` constructor that wires itself into the service bus

All you've done is port over the original `observe` method corresponding to `CarDetected` events. The lambda captures `this` ❶ and `bus` ❷ by reference in the callback. Capturing `this` allows you to compute collision times, whereas capturing `bus` allows you to publish a `BrakeCommand` ❸ if the conditions are satisfied. Now the unit-test binary outputs the following:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test no alert when not imminent successful.
```

Finally, turn on the last test, `alert_when_imminent`, as displayed in Listing 10-28.

```
void alert_when_imminent() {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  auto_brake.set_collision_threshold_s(10L);
  bus.speed_update_callback(SpeedUpdate{ 100L });
  bus.car_detected_callback(CarDetected{ 100L, 0L });
  assert_that(bus.commands_published == 1, "1 brake command was not published");
  assert_that(bus.last_command.time_to_collision_s == 1L,
              "time to collision not computed correctly."); ❶
}
```

Listing 10-28: Refactoring the `alert_when_imminent` unit test

In MockServiceBus, you actually saved the last BrakeCommand published to the bus into a member. In the test, you can use this member to verify that the time to collision was computed correctly. If a car is going 100 meters per second, it will take 1 second to hit a stationary car parked 100 meters away. You check that the BrakeCommand has the correct time to collision recorded by referring to the time_to_collision_s field on our mock bus ❶.

Recompiling and rerunning, you finally have the test suite fully green again:

```
[+] Test initial speed is 0 successful.
[+] Test initial sensitivity is 5 successful.
[+] Test sensitivity greater than 1 successful.
[+] Test speed is saved successful.
[+] Test no alert when not imminent successful.
[+] Test alert when imminent successful.
```

Refactoring is now complete.

Reevaluating the Unit-Testing Solution

Looking back at the unit-testing solution, you can identify several components that have nothing to do with AutoBrake. These are general purpose unit-testing components that you could reuse in future unit tests. Recall the two helper functions created in Listing 10-29.

```
#include <stdexcept>
#include <cstdio>

void assert_that(bool statement, const char* message) {
  if (!statement) throw std::runtime_error{ message };
}

void run_test(void(*unit_test)(), const char* name) {
  try {
    unit_test();
    printf("[+] Test %s successful.\n", name);
    return;
  } catch (const std::exception& e) {
    printf("[-] Test failure in %s. %s.\n", name, e.what());
  }
}
```

Listing 10-29: An austere unit-testing framework

These two functions reflect two fundamental aspects of unit testing: making assertions and running tests. Rolling your own simple assert_that function and run_test harness works, but this approach doesn't scale very well. You can do a lot better by leaning on a unit-testing framework.

Unit-Testing and Mocking Frameworks

Unit-testing frameworks provide commonly used functions and the scaffolding you need to tie your tests together into a user-friendly program. These frameworks provide a wealth of functionality that helps you create concise, expressive tests. This section offers a tour of several popular unit-testing and mocking frameworks.

The Catch Unit-Testing Framework

One of the most straightforward unit-testing frameworks, Catch by Phil Nash, is available at *https://github.com/catchorg/Catch2/*. Because it's a header-only library, you can set up Catch by downloading the single-header version and including it in each translation unit that contains unit-testing code.

NOTE *At press time, Catch's latest version is 2.9.1.*

Defining an Entry Point

Tell Catch to provide your test binary's entry point with `#define CATCH_CONFIG_MAIN`. Together, the Catch unit-test suite starts as follows:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

That's it. Within the `catch.hpp` header, it looks for the `CATCH_CONFIG_MAIN` preprocessor definition. When present, Catch will add in a `main` function so you don't have to. It will automatically grab all the unit tests you've defined and wrap them with a nice harness.

Defining Test Cases

Earlier, in "Unit Tests" on page 282, you defined a separate function for each unit test. Then you would pass a pointer to this function as the first parameter to `run_test`. You passed the name of the test as the second parameter, which is a bit redundant because you've already provided a descriptive name for the function pointed to by the first argument. Finally, you had to implement your own `assert` function. Catch handles all of this ceremony implicitly. For each unit test, you use the `TEST_CASE` macro, and Catch handles all the integration for you.

Listing 10-30 illustrates how to build a trivial Catch unit test program.

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

TEST_CASE("AutoBrake") { ❶
  // Unit test here
}
```

```
===============================================================================
test cases: 1 | 1 passed ❶
assertions: - none - ❷
```

Listing 10-30: A simple Catch unit-test program

The Catch entry point detects that you declared one test called `AutoBrake` ❶. It also provides a warning that you haven't made any assertions ❷.

Making Assertions

Catch comes with a built-in assertion that features two distinct families of assertion macros: `REQUIRE` and `CHECK`. The difference between them is that `REQUIRE` will fail a test immediately, whereas `CHECK` will allow the test to run to completion (but still cause a failure). `CHECK` can be useful sometimes when groups of related assertions that fail lead the programmer down the right path of debugging problems. Also included are `REQUIRE_FALSE` and `CHECK_FALSE`, which check that the contained statement evaluates to false rather than true. In some situations, you might find this a more natural way to represent a requirement.

All you need to do is wrap a Boolean expression with the `REQUIRE` macro. If the expression evaluates to false, the assertion fails. You provide an *assertion expression* that evaluates to true if the assertion passes and false if it fails:

```
REQUIRE(assertion-expression);
```

Let's look at how to combine `REQUIRE` with a `TEST_CASE` to build a unit test.

NOTE *Because it's by far the most common Catch assertion, we'll use `REQUIRE` here. Refer to the Catch documentation for more information.*

Refactoring the initial_speed_is_zero Test to Catch

Listing 10-31 shows the `initial_speed_is_zero` test refactored to use Catch.

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <functional>

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrake {
  --snip--
};
```

```
TEST_CASE❶("initial car speed is zero"❷) {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  REQUIRE(auto_brake.get_speed_mps() == 0); ❸
}
```

Listing 10-31: An `initial_speed_is_zero` *unit test refactored to use Catch*

You use the `TEST_CASE` macro to define a new unit test ❶. The test is described by its sole parameter ❷. Inside the body of the `TEST_CASE` macro, you proceed with the unit test. You also see the `REQUIRE` macro in action ❸. To see how Catch handles failed tests, comment out the `speed_mps` member initializer to cause a failing test and observe the program's output, as shown in Listing 10-32.

```
struct AutoBrake {
  AutoBrake(IServiceBus& bus)
    : collision_threshold_s{ 5 }/*,
    speed_mps{} */{ ❶
  --snip--
};
```

Listing 10-32: Intentionally commenting out the `speed_mps` *member initializer to cause test failures (using Catch)*

The appropriate member initializer ❶ is commented out, resulting in a test failure. Rerunning the Catch test suite in Listing 10-31 yields the output in Listing 10-33.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
catch_example.exe is a Catch v2.0.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
initial car speed is zero
-------------------------------------------------------------------------------
c:\users\jalospinoso\catch-test\main.cpp(82)
...............................................................................

c:\users\jalospinoso\catch-test\main.cpp(85):❶ FAILED:
  REQUIRE( auto_brake.get_speed_mps()L == 0 ) ❷
with expansion:
  -92559631349317830736831783200707727132248687965119994463780864.0 ❸
  ==
  0

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Listing 10-33: The output from running the test suite after implementing Listing 10-31

This is vastly superior output to what you had produced in the home-grown unit-test suite. Catch tells you the exact line where the unit test failed ❶ and then prints this line for you ❷. Next, it expands this line into the actual values encountered at runtime. You can see that the grotesque (uninitialized) value returned by `get_speed_mps()` is clearly not `0` ❸. Compare this output to the output of the home-grown unit test; I think you'll agree that there's immediate value to using Catch.

Assertions and Exceptions

Catch also provides a special assertion called `REQUIRE_THROWS`. This macro requires that the contained expression throw an exception. To achieve similar functionality in the home-grown unit-test framework, consider this multiline monstrosity:

```
try {
  auto_brake.set_collision_threshold_s(0.5L);
} catch (const std::exception&) {
  return;
}
assert_that(false, "no exception thrown");
```

Other exception-aware macros are available as well. You can require that some expression evaluation not throw an exception using the `REQUIRE_NOTHROW` and `CHECK_NOTHROW` macros. You can also be specific about the type of the exception you expect to be thrown by using the `REQUIRE_THROWS_AS` and `CHECK_THROWS_AS` macros. These expect a second parameter describing the expected type. Their usages are similar to `REQUIRE`; you simply provide some expression that must throw an exception for the assertion to pass:

```
REQUIRE_THROWS(expression-to-evaluate);
```

If the *`expression-to-evaluate`* doesn't throw an exception, the assertion fails.

Floating-Point Assertions

The `AutoBrake` class involves floating-point arithmetic, and we've been glossing over a potentially very serious problem with the assertions. Because floating-point numbers entail rounding errors, it's not a good idea to check for equality using `operator==`. The more robust approach is to test whether the difference between floating-point numbers is arbitrarily small. With Catch, you can handle these situations effortlessly using the `Approx` class, as shown in Listing 10-34.

```
TEST_CASE("AutoBrake") {
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
  REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5L));
}
```

Listing 10-34: A refactor of the "initializes sensitivity to five" test using the `Approx` class

The Approx class helps Catch perform tolerant comparisons of floating-point values. It can exist on either side of a comparison expression. It has sensible defaults for how tolerant it is, but you have fine-grained control over the specifics (see the Catch documentation on epsilon, margin, and scale).

Fail

You can cause a Catch test to fail using the FAIL() macro. This can sometimes be useful when combined with conditional statements, as in the following:

```
if (something-bad) FAIL("Something bad happened.")
```

Use a REQUIRE statement if a suitable one is available.

Test Cases and Sections

Catch supports the idea of test cases and sections, which make common setup and teardown in your unit tests far easier. Notice that each of the tests has some repeated ceremony each time you construct an AutoBrake:

```
  MockServiceBus bus{};
  AutoBrake auto_brake{ bus };
```

There's no need to repeat this code over and over again. Catch's solution to this common setup is to use nested SECTION macros. You can nest SECTION macros within a TEST_CASE in the basic usage pattern, as demonstrated in Listing 10-35.

```
TEST_CASE("MyTestGroup") {
  // Setup code goes here ❶
  SECTION("MyTestA") { ❷
    // Code for Test A
  }
  SECTION("MyTestB") { ❸
    // Code for Test B
  }
}
```

Listing 10-35: An example Catch setup with nested macros

You can perform all of the setup once at the beginning of a TEST_CASE ❶. When Catch sees SECTION macros nested within a TEST_CASE, it (conceptually) copies and pastes all the setup into each SECTION ❷❸. Each SECTION runs independently of the others, so generally any side effects on objects created in the TEST_CASE aren't observed across SECTION macros. Further, you can embed a SECTION macro within another SECTION macro. This might be useful if you have a lot of setup code for a suite of closely related tests (although it may just make sense to split this suite into its own TEST_CASE).

Let's look at how this approach simplifies the AutoBrake unit-test suite.

Refactoring the AutoBrake Unit Tests to Catch

Listing 10-36 refactors all the unit tests into a Catch style.

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include <functional>
#include <stdexcept>

struct IServiceBus {
  --snip--
};

struct MockServiceBus : IServiceBus {
  --snip--
};

struct AutoBrake {
  --snip--
};

TEST_CASE("AutoBrake"❶) {
  MockServiceBus bus{}; ❷
  AutoBrake auto_brake{ bus }; ❸

  SECTION❹("initializes speed to zero"❺) {
    REQUIRE(auto_brake.get_speed_mps() == Approx(0));
  }

  SECTION("initializes sensitivity to five") {
    REQUIRE(auto_brake.get_collision_threshold_s() == Approx(5));
  }

  SECTION("throws when sensitivity less than one") {
    REQUIRE_THROWS(auto_brake.set_collision_threshold_s(0.5L));
  }

  SECTION("saves speed after update") {
    bus.speed_update_callback(SpeedUpdate{ 100L });
    REQUIRE(100L == auto_brake.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 50L });
    REQUIRE(50L == auto_brake.get_speed_mps());
    bus.speed_update_callback(SpeedUpdate{ 0L });
    REQUIRE(0L == auto_brake.get_speed_mps());
  }

  SECTION("no alert when not imminent") {
    auto_brake.set_collision_threshold_s(2L);
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 1000L, 50L });
    REQUIRE(bus.commands_published == 0);
  }

  SECTION("alert when imminent") {
    auto_brake.set_collision_threshold_s(10L);
```

```
    bus.speed_update_callback(SpeedUpdate{ 100L });
    bus.car_detected_callback(CarDetected{ 100L, 0L });
    REQUIRE(bus.commands_published == 1);
    REQUIRE(bus.last_command.time_to_collision_s == Approx(1));
  }
}
-------------------------------------------------------------------------------
===============================================================================
All tests passed (9 assertions in 1 test case)
```

Listing 10-36: Using the Catch framework to implement the unit tests

Here, `TEST_CASE` is renamed to `AutoBrake` to reflect its more generic purpose ❶. Next, the body of the `TEST_CASE` begins with the common setup code that all the `AutoBrake` unit tests share ❷❸. Each of the unit tests has been converted into a `SECTION` macro ❹. You name each of the sections ❺ and then place the test-specific code within the `SECTION` body. Catch will do all the work of stitching together the setup code with each of the `SECTION` bodies. In other words, you get a fresh `AutoBrake` each time: the order of the `SECTIONS` doesn't matter here, and they're totally independent.

Google Test

Google Test is another extremely popular unit-testing framework. Google Test follows the xUnit unit-testing framework tradition, so if you're familiar with, for example, junit for Java or nunit for .NET, you'll feel right at home using Google Test. One nice feature when you're using Google Test is that the mocking framework Google Mocks was merged in some time ago.

Configuring Google Test

Google Test takes some time to get up and running. Unlike Catch, Google Test is not a header-only library. You must download it from *https://github.com/google/googletest/*, compile it into a set of libraries, and link those libraries into your test project as appropriate. If you use a popular desktop build system, such as GNU Make, Mac Xcode, or Visual Studio, some templates are available that you can use to start building the relevant libraries.

For more information about getting Google Test up and running, refer to the Primer available in the repository's `docs` directory.

NOTE *At press time, Google Test's latest version is 1.8.1. See this book's companion source, available at* https://ccc.codes*, for one method of integrating Google Test into a Cmake build.*

Within your unit-test project, you must perform two operations to set up Google Test. First, you must ensure that the included directory of your Google Test installation is in the header search path of your unit-test project. This allows you to use `#include "gtest/gtest.h"` within your tests. Second, you must instruct your linker to include `gtest` and `gtest_main` static libraries from your Google Test installation. Make sure that you link in the correct architecture and configuration settings for your computer.