Try writing a few shell scripts to automate the process. Do you always click on the same sequence of icons repeatedly? Can you create a macro to do all that for you?

- How much of your project paperwork can be automated? Given the high expense of programming staff, determine how much of the project’s budget is being wasted on administrative procedures. Can you justify the amount of time it would take to craft an automated solution based on the overall cost savings it would achieve?

Ruthless Testing

Most developers hate testing. They tend to test gently, subconsciously knowing where the code will break and avoiding the weak spots. Pragmatic Programmers are different. We are driven to find our bugs now, so we don’t have to endure the shame of others finding our bugs later.

Finding bugs is somewhat like fishing with a net. We use fine, small nets (unit tests) to catch the minnows, and big, coarse nets (integration tests) to catch the killer sharks. Sometimes the fish manage to escape, so we patch any holes that we find, in hopes of catching more and more slippery defects that are swimming about in our project pool.

Tip 62
Test Early. Test Often. Test Automatically.

We want to start testing as soon as we have code. Those tiny minnows have a nasty habit of becoming giant, man-eating sharks pretty fast, and catching a shark is quite a bit harder. But we don’t want to have to do all that testing by hand.

---

5. For estimating purposes, you can figure an industry average of about US$100,000 per head—that’s salary plus benefits, training, office space and overhead, and so on.
Many teams develop elaborate test plans for their projects. Sometimes they will even use them. But we've found that teams that use automated tests have a much better chance of success. Tests that run with every build are much more effective than test plans that sit on a shelf.

The earlier a bug is found, the cheaper it is to remedy. "Code a little, test a little" is a popular saying in the Smalltalk world,⁶ and we can adopt that mantra as our own by writing test code at the same time (or even before) we write the production code.

In fact, a good project may well have more test code than production code. The time it takes to produce this test code is worth the effort. It ends up being much cheaper in the long run, and you actually stand a chance of producing a product with close to zero defects.

Additionally, knowing that you've passed the test gives you a high degree of confidence that a piece of code is "done."

![Tip 63](image)

**Tip 63**

Coding Ain't Done 'Til All the Tests Run

Just because you have finished hacking out a piece of code doesn't mean you can go tell your boss or your client that it's done. It's not. First of all, code is never really done. More importantly, you can't claim that it is usable by anyone until it passes all of the available tests.

We need to look at three main aspects of project-wide testing: what to test, how to test, and when to test.

**What to Test**

There are several major types of software testing that you need to perform:

- Unit testing
- Integration testing
- Validation and verification

---

⁶ eXtreme Programming [URL 45] calls this concept "continuous integration, relentless testing."
- Resource exhaustion, errors, and recovery
- Performance testing
- Usability testing

This list is by no means complete, and some specialized projects will require various other types of testing as well. But it gives us a good starting point.

**Unit Testing**

A unit test is code that exercises a module. We covered this topic by itself in *Code That's Easy to Test*, page 189. Unit testing is the foundation of all the other forms of testing that we'll discuss in this section. If the parts don't work by themselves, they probably won't work well together. All of the modules you are using must pass their own unit tests before you can proceed.

Once all of the pertinent modules have passed their individual tests, you're ready for the next stage. You need to test how all the modules use and interact with each other throughout the system.

**Integration Testing**

Integration testing shows that the major subsystems that make up the project work and play well with each other. With good contracts in place and well tested, any integration issues can be detected easily. Otherwise, integration becomes a fertile breeding ground for bugs. In fact, it is often the single largest source of bugs in the system.

Integration testing is really just an extension of the unit testing we've described—only now you're testing how entire subsystems honor their contracts.

**Validation and Verification**

As soon as you have an executable user interface or prototype, you need to answer an all-important question: the users told you what they wanted, but is it what they need?

Does it meet the functional requirements of the system? This, too, needs to be tested. A bug-free system that answers the wrong question isn't very useful. Be conscious of end-user access patterns and
how they differ from developer test data (for an example, see the story about brush strokes on page 92).

**Resource Exhaustion, Errors, and Recovery**

Now that you have a pretty good idea that the system will behave correctly under ideal conditions, you need to discover how it will behave under **real-world** conditions. In the real world, your programs don't have limitless resources; they run out of things. A few limits your code may encounter include:

- Memory
- Disk space
- CPU bandwidth
- Wall-clock time
- Disk bandwidth
- Network bandwidth
- Color palette
- Video resolution

You might actually check for disk space or memory allocation failures, but how often do you test for the others? Will your application fit on a 640 × 480 screen with 256 colors? Will it run on a 1600 × 1280 screen with 24-bit color without looking like a postage stamp? Will the batch job finish before the archive starts?

You can detect environmental limitations, such as the video specifications, and adapt as appropriate. Not all failures are recoverable, however. If your code detects that memory has been exhausted, your options are limited: you may not have enough resources left to do anything except fail.

When the system does fail,⁷ will it fail gracefully? Will it try, as best it can, to save its state and prevent loss of work? Or will it "GPF" or "core-dump" in the user's face?

---

⁷. Our copy editor wanted us to change this sentence to "If the system does fail...." We resisted.
Performance Testing

Performance testing, stress testing, or testing under load may be an important aspect of the project as well.

Ask yourself if the software meets the performance requirements under real-world conditions— with the expected number of users, or connections, or transactions per second. Is it scalable?

For some applications, you may need specialized testing hardware or software to simulate the load realistically.

Usability Testing

Usability testing is different from the types of testing discussed so far. It is performed with real users, under real environmental conditions.

Look at usability in terms of human factors. Were there any misunderstandings during requirements analysis that need to be addressed? Does the software fit the user like an extension of the hand? (Not only do we want our own tools to fit our hands, but we want the tools we create for users to fit their hands as well.)

As with validation and verification, you need to perform usability testing as early as you can, while there is still time to make corrections. For larger projects, you may want to bring in human factors specialists. (If nothing else, it’s fun to play with the one-way mirrors.)

Failure to meet usability criteria is just as big a bug as dividing by zero.

How to Test

We’ve looked at what to test. Now we’ll turn our attention to how to test, including:

- Regression testing
- Test data
- Exercising GUI systems
- Testing the tests
- Testing thoroughly
Design/Methodology Testing

Can you test the design of the code itself and the methodology you used to build the software? After a fashion, yes you can. You do this by analyzing metrics—measurements of various aspects of the code. The simplest metric (and often the least interesting) is lines of code—how big is the code itself?

There are a wide variety of other metrics you can use to examine code, including:

- McCabe Cyclomatic Complexity Metric (measures complexity of decision structures)
- Inheritance fan-in (number of base classes) and fan-out (number of derived modules using this one as a parent)
- Response set (see Decoupling and the Law of Demeter, page 138)
- Class coupling ratios (see [URL 48])

Some metrics are designed to give you a “passing grade,” while others are useful only by comparison. That is, you calculate these metrics for every module in the system and see how a particular module relates to its brethren. Standard statistical techniques (such as mean and standard deviation) are usually used here.

If you find a module whose metrics are markedly different from all the rest, you need to ask yourself if that is appropriate. For some modules, it may be okay to “blow the curve.” But for those that don’t have a good excuse, it can indicate potential problems.

Regression Testing

A regression test compares the output of the current test with previous (or known) values. We can ensure that bugs we fixed today didn’t break things that were working yesterday. This is an important safety net, and it cuts down on unpleasant surprises.

All of the tests we’ve mentioned so far can be run as regression tests, ensuring that we haven’t lost any ground as we develop new code. We can run regressions to verify performance, contracts, validity, and so on.
Test Data
Where do we get the data to run all these tests? There are only two kinds of data: real-world data and synthetic data. We actually need to use both, because the different natures of these kinds of data will expose different bugs in our software.

Real-world data comes from some actual source. Possibly it has been collected from an existing system, a competitor’s system, or a prototype of some sort. It represents typical user data. The big surprises come as you discover what typical means. This is most likely to reveal defects and misunderstandings in requirements analysis.

Synthetic data is artificially generated, perhaps under certain statistical constraints. You may need to use synthetic data for any of the following reasons.

- You need a lot of data, possibly more than any real-world sample could provide. You might be able to use the real-world data as a seed to generate a larger sample set, and tweak certain fields that need to be unique.
- You need data to stress the boundary conditions. This data may be completely synthetic: date fields containing February 29, 1999, huge record sizes, or addresses with foreign postal codes.
- You need data that exhibits certain statistical properties. Want to see what happens if every third transaction fails? Remember the sort algorithm that slows to a crawl when handed presorted data? You can present data in random or sorted order to expose this kind of weakness.

Exercising GUI Systems
Testing GUI-intensive systems often requires specialized testing tools. These tools may be based on a simple event capture/playback model, or they may require specially written scripts to drive the GUI. Some systems combine elements of both.

Less sophisticated tools enforce a high degree of coupling between the version of software being tested and the test script itself: if you move a dialog box or make a button smaller, the test may not be able to
find it, and may fail. Most modern GUI testing tools use a number of different techniques to get around this problem, and try to adjust to minor layout differences.

However, you can’t automate everything. Andy worked on a graphics system that allowed the user to create and display nondeterministic visual effects which simulated various natural phenomena. Unfortunately, during testing you couldn’t just grab a bitmap and compare the output with a previous run, because it was designed to be different every time. For situations such as this one, you may have no choice but to rely on manual interpretation of test results.

One of the many advantages of writing decoupled code (see Decoupling and the Law of Demeter, page 138) is more modular testing. For instance, for data processing applications that have a GUI front end, your design should be decoupled enough so that you can test the application logic without having a GUI present. This idea is similar to testing your subcomponents first. Once the application logic has been validated, it becomes easier to locate bugs that show up with the user interface in place (it’s likely that the bugs were created by the user-interface code).

Testing the Tests
Because we can’t write perfect software, it follows that we can’t write perfect test software either. We need to test the tests.

Think of our set of test suites as an elaborate security system, designed to sound the alarm when a bug shows up. How better to test a security system than to try to break in?

After you have written a test to detect a particular bug, cause the bug deliberately and make sure the test complains. This ensures that the test will catch the bug if it happens for real.

Tip 64
Use Saboteurs to Test Your Testing

If you are really serious about testing, you might want to appoint a project saboteur. The saboteur’s role is to take a separate copy of the
source tree, introduce bugs on purpose, and verify that the tests will catch them.

When writing tests, make sure that alarms sound when they should.

**Testing Thoroughly**

Once you are confident that your tests are correct, and are finding bugs you create, how do you know if you have tested the code base thoroughly enough?

The short answer is “you don’t,” and you never will. But there are products on the market that can help. These *coverage analysis* tools watch your code during testing and keep track of which lines of code have been executed and which haven’t. These tools help give you a general feel for how comprehensive your testing is, but don’t expect to see 100% coverage.

Even if you do happen to hit every line of code, that’s not the whole picture. What is important is the number of states that your program may have. States are not equivalent to lines of code. For instance, suppose you have a function that takes two integers, each of which can be a number from 0 to 999.

```c
int test(int a, int b) {
    return a / (a + b);
}
```

In theory, this three-line function has 1,000,000 logical states, 999,999 of which will work correctly and one that will not (when a + b equals zero). Simply knowing that you executed this line of code doesn’t tell you that—you would need to identify all possible states of the program. Unfortunately, in general this is a *really hard* problem. Hard as in, “The sun will be a cold hard lump before you can solve it.”

---

**Tip 65**

**Test State Coverage, Not Code Coverage**

Even with good code coverage, the data you use for testing still has a huge impact, and, more importantly, the *order* in which you traverse code may have the largest impact of all.
When to Test

Many projects tend to leave testing to the last minute—right where it will be cut against the sharp edge of a deadline. We need to start much sooner than that. As soon as any production code exists, it needs to be tested.

Most testing should be done automatically. It's important to note that by "automatically" we mean that the test results are interpreted automatically as well. See Ubiquitous Automation, page 230, for more on this subject.

We like to test as frequently as we can, and always before we check code into the source repository. Some source code control systems, such as Aegis, can do this automatically. Otherwise, we just type

```
% make test
```

Usually, it isn’t a problem to run regressions on all of the individual unit tests and integration tests as often as needed.

But some tests may not be easily run on a such a frequent basis. Stress tests, for instance, may require special setup or equipment, and some hand holding. These tests may be run less often—weekly or monthly, perhaps. But it is important that they be run on a regular, scheduled basis. If it can’t be done automatically, then make sure it appears on the schedule, with all the necessary resources allocated to the task.

Tightening the Net

Finally, we'd like to reveal the single most important concept in testing. It is an obvious one, and virtually every textbook says to do it this way. But for some reason, most projects still do not.

If a bug slips through the net of existing tests, you need to add a new test to trap it next time.

---

8. **dead-line** (self-affirming) n (1864) a line drawn within or around a prison that a prisoner passes at the risk of being shot—Webster's Collegiate Dictionary.
Find Bugs Once

Once a human tester finds a bug, it should be the last time a human tester finds that bug. The automated tests should be modified to check for that particular bug from then on, every time, with no exceptions, no matter how trivial, and no matter how much the developer complains and says, "Oh, that will never happen again."

Because it will happen again. And we just don't have the time to go chasing after bugs that the automated tests could have found for us. We have to spend our time writing new code—and new bugs.

Related sections include:
- The Cat Ate My Source Code, page 2
- Debugging, page 90
- Decoupling and the Law of Demeter, page 138
- Refactoring, page 184
- Code That's Easy to Test, page 189
- Ubiquitous Automation, page 230

Challenges
- Can you automatically test your project? Many teams are forced to answer "no." Why? Is it too hard to define the acceptable results? Won't this make it hard to prove to the sponsors that the project is "done"?

Is it too hard to test the application logic independent of the GUI? What does this say about the GUI? About coupling?