Outline of Part 1

- ...
- Common test design errors
- Testing changes

è Test implementation
  - why automation
  - the basic tools
  - dangers of automation
  - problems implementing tests
  - testpoints

2

Why Automated Testing?

- **Traditionally, developer tests are manual**
  - **this is cheap now, expensive later**
- **First release well tested**
  - **good developer tests**
  - **system tests to fill in the gaps**
  - **testing through use**
- **Second release poorly tested**
  - **you're not retesting old code**
  - **system testing isn't retesting old code much**
  - **users are retesting old code**

Most developers execute their tests through the debugger. That's the easiest way to test. It's short-term thinking. Here's what invariably happens:

Release 1 is reasonably well tested. Developers test all the new code they're writing. System testers write tests to find the types of defects that developer testing doesn't find. Alpha and beta sites back up system testing. And of course, the pioneers who buy release 1.0 do a little testing too.

In release 2.0, all the developers are busy adding new features and new faults (sometimes in the form of bugfixes). They test the new code they write. They don't retest the old code - who has time? Unfortunately, there are defects in the interaction between old and new code.

The system testers don't have tests that fill the role of those unexecuted developer tests. Their system tests exercise the old code some, but not enough to find all the new faults. Unless you have a much stronger beta testing program, the people who find the new defects in old code are the users. They're the ones who exercise the old code most thoroughly.

3

The Basic Tools

- **The test driver**
  - **sets up the test**
  - **feeds input to the code**
  - **collects results**
  - **judges correctness**
- **The test manager (or suite driver)**
  - **runs all or part of the test suite**
  - **tells you how this run differs from the last**

For automated testing, you need two basic tools. Details are given in the following reference slides. Different systems will require different tools, but the tools all perform the same basic function and fulfill the same requirements.

The test driver is a tool that sets up the environment for a test, provides the inputs to the code, collects the results of the test, and judges whether the code passed the test (often by comparing the collected actual results to stored reference results).

The test manager (also called the test suite driver or the test suite manager) runs more than one test. It can run the entire test suite. Sometimes it can run selected subsets. In addition to running tests and telling you what tests failed, it should also be able to compare this run to a previous run. What tests used to pass, but now fail? What tests used to fail, but now pass?

Reference

4

Selecting Test Implementation Tools

Expect This from Test Drivers

S - Setup

E - Execution

A - Analysis

R - Reporting

C - Cleanup

H - Help

Think of yourself as a customer shopping for a test driver. You will have certain requirements. This acronym will remind you of them. It comes from:

Mark Bergman and Keith Stobie, 'How to Automate Testing, the Big Picture', 9th International Conference on Software Testing (USPDI), also in 1992 Quality Week Proceedings (Software Research).

Reference 6

Setup, Execution, and Analysis

- **All test setup**
- **Provides inputs**
- **Collects results**
- **Judges correctness**
  - **reference files**
  - **(beware of immortalizing failures)**
  - **with a program**

A driver must do the following:

•It must set up the environment before the test runs. Ideally, everything about the environment should be explicitly initialized by the driver. If not, you'll run into problems like tests that only run when you invoke them. (Someone else will eventually want to run your tests.) At the least, the test should check whether the environment is correctly initialized.

•It must provide all inputs to the product.

•It must collect all results: terminal output, network traffic, files changed, and so on.

•It must judge the correctness of the results. With many types of output, such as messages to the user, this is most easily done by capturing the output in a file and then comparing that **log file** to a predefined **reference file** that contains the correct output. (Beware, though: people sometimes get in the habit of creating the reference file by scanning the log file, seeing that it seems correct, and saving it. Often, the result is a failure "immortalized" as the reference result.) Other times, the results cannot be exactly predetermined, so a program (or script) that checks correctness has to be written. For example, a test might require that a file contain particular records, but can't say what their order should be. Since the order may change, a test with a reference file would require constant updating. Better to write a correctness checking program that first sorts the records, then compares them to a reference file.

Reference 7

Reporting, Cleanup, and Help

- **Reporting**
  - **of results**
  - **as desired**
  - **other useful information**
- **Cleanup**
  - **unless product fails**
- **Help**

The driver should report the results of the analysis. The report may be to varying levels of detail, ranging from a simple “Test 3 found a failure” to a complete description of everything relevant about the failure. (The same driver might produce both types of reports.) The report can also contain other useful information, such as how long the test took to run.

After execution, the test should clean up after itself by removing temporary files and so on. Clean up only if the program passes the test. In the case of failure, the temporary files may help during debugging.

If the driver is too hard to learn, you won’t use it. You need help: user documentation, on-line help, perhaps an expert for consultation, etc.

Reference 8

Your Requirements for a Test Suite Manager

- Execute the entire suite
- Execute a portion of the suite
- Report on what's changed
- SEARCH also applies here

There's another tool you will want to use: the test suite manager. It should at least be able to run the entire test suite. It's better if it can run selected portions (those that test some changed feature, for example).

The test suite manager should tell you what's changed since a previous run of the suite. The next page gives more details.

Everything that SEARCH stands for applies at the test suite manager level as well. For example, the manager should be able to produce aggregate reports (percentage successes, total elapsed time, perhaps total coverage achieved, and so on).

Reference 9

Test Suite Manager Reports

- "Product now fails this test"
  - defect caused by fix?
- "Product now passes this test"
  - defect successfully fixed?
  - defect has "gone away" for no apparent reason?
- "Product fails differently"
  - new defect?

Test suite managers should report at least this information.

•New failures mean some change has caused a fault.

•New successes mean a fault has been fixed - you hope. If there's no reason for the new success, it more likely means that some change now obscures the fault, so that the test no longer makes it visible. That's bad, because you no longer have a way of tracking it down.

•The product can also fail in a new way. This could be a new manifestation of an old fault, which is probably not particularly important, or it could be a completely new fault, which definitely is. If the test suite manager only reported that a failure had happened, you might not discover the new fault.

Reference

10

Your Own Drivers and Managers

- SEARCH applies in principle
- In practice, can simplify
  - fewer users (less Help needed)
  - specialized application (full generality and support unneeded)

If existing testing tools won't save you work, you'll need to build your own. In principle, your tools should do everything that SEARCH stands for. In practice, your tools probably won't be as polished. Your tools can be simpler because they'll be used by fewer people. That means less help is needed - more can be done by word of mouth, by consulting the local expert. Since the tools will be used in a narrower domain, they can be more specialized and don't have to be as flexible.

But keep the SEARCH requirements in mind. Leave things out of your tools because you've decided they're unnecessary, not because you overlooked them.

11

Test Implementation Lab

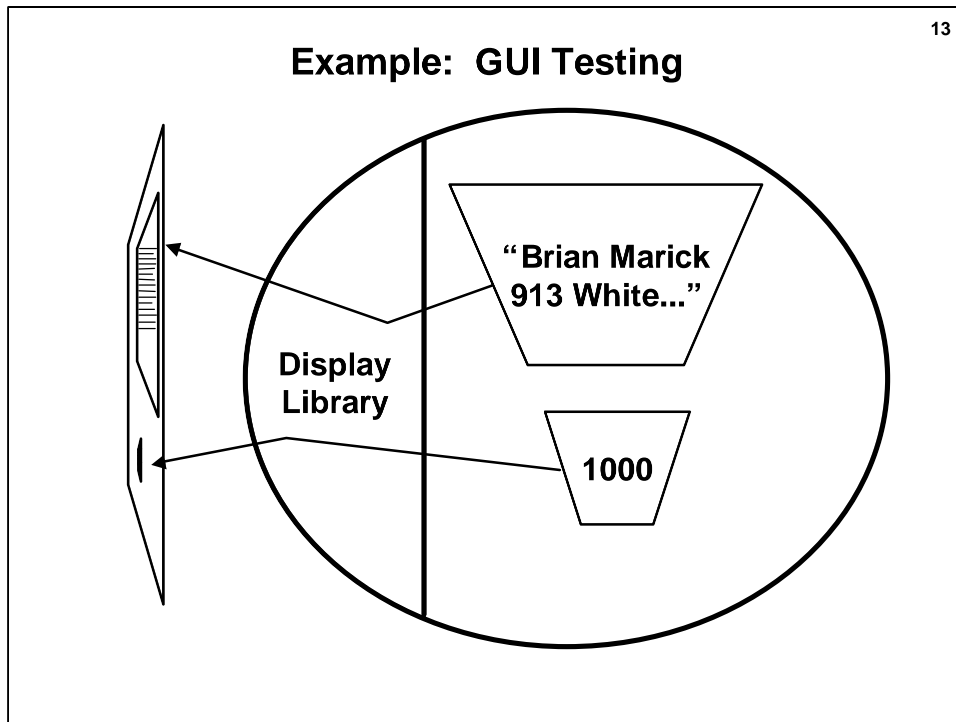
12

The Major Danger: Test Suite Maintenance

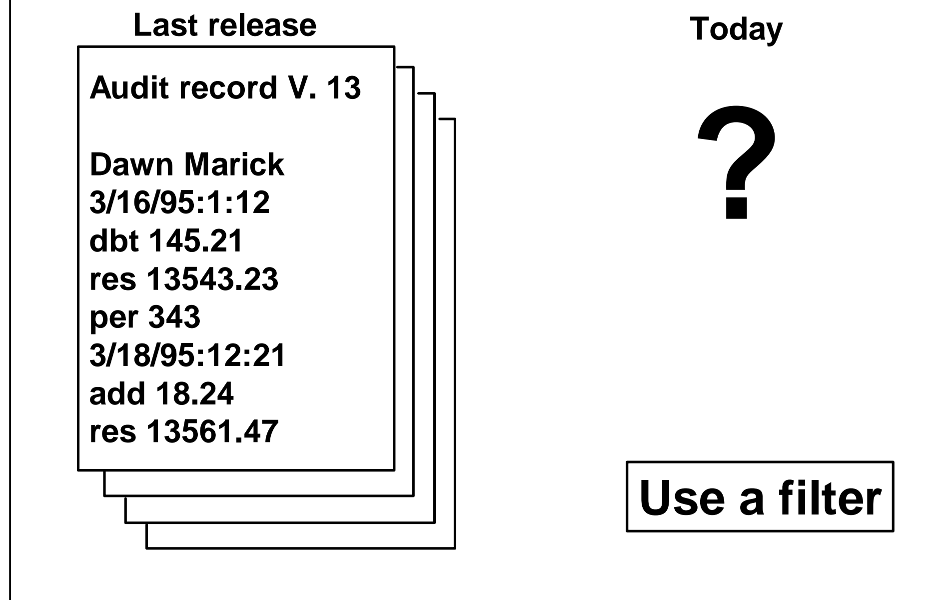
- **The product will change**
- **Your test suite will have to change with it**
- **That can be easy or hard**
  - **it depends on how much thought you put into test implementation**
  - **you must predict likely changes**

The whole point of automated testing is that the product will change and you will have to rerun tests. However, the effort of automating may be wasted if you don't plan for change. If you don't, changes in the new version will invalidate many old tests. You'll spend so much time updating old automated tests that you'll have no time to design new ones. (Or, more likely, you'll get disgusted and throw out the old tests.)

Generally, updating old tests is reasonably easy if you plan for it. Planning means predicting likely changes and designing/implementing your tests so that they're less affected by them.



As an example, suppose your product uses a graphical user interface (GUI). The obvious way to check actual results of a test run is to take a snapshot of the screen image. The problem is that the snapshot contains a lot of irrelevant information. You care that the Account Balance text field contains $1,000. The snapshot contains that, but also the color of the bits, the position of the text field on the screen, and so on. It's these latter things that are most subject to change. When they change, all the old snapshots are invalidated and have to be recreated and rechecked by hand. For this reason, modern GUI test drivers capture not snapshots but the underlying data structure that corresponds to the screen image. They capture the "$1,000" and leave the rest alone. Changes to colors and screen location are ignored because they're irrelevant. (Of course, someone does have to test that the underlying data structures turn into the right pretty pictures on the screen - but that's a specific testing task, not part of everyone's job.)



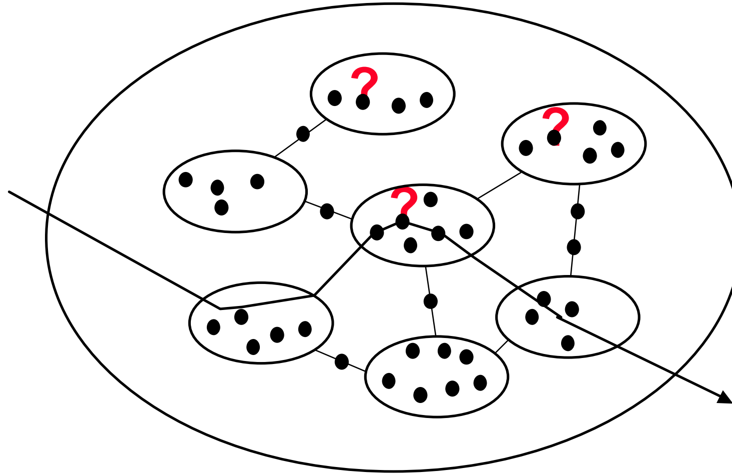
As another example, suppose you're testing a banking system. Such a system might have an audit trail. You might decide that the audit trail contains the information you need to check correctness of results. You create your test suite by running the tests, carefully validating the outputs, then storing those validated audit records as reference files.

The problem is that audit records often change from release to release. In the next release, every test will fail. You have to check whether the failure was a real failure or because of an audit record change.

The way to avoid this problem is to write a release-specific filter. That filter extracts exactly and only the information you care about. That extracted information is stored in the reference file. In the next release, only the single filter has to be updated, not every reference file.

15

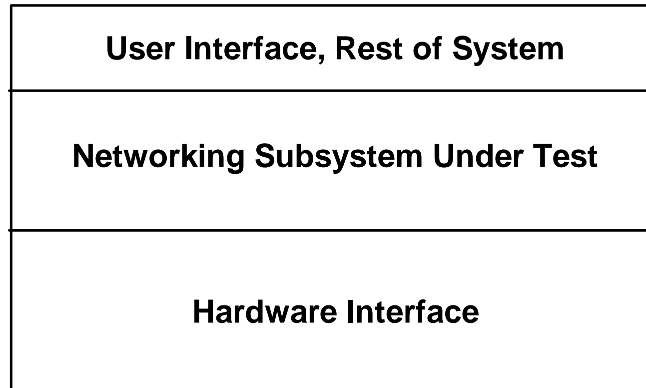
Problems Testing Through the System Interface (1)



Up to now, the course has assumed you'll test individual routines through the system interface. There are problems testing though the system interface. They'll be described in the following slides.

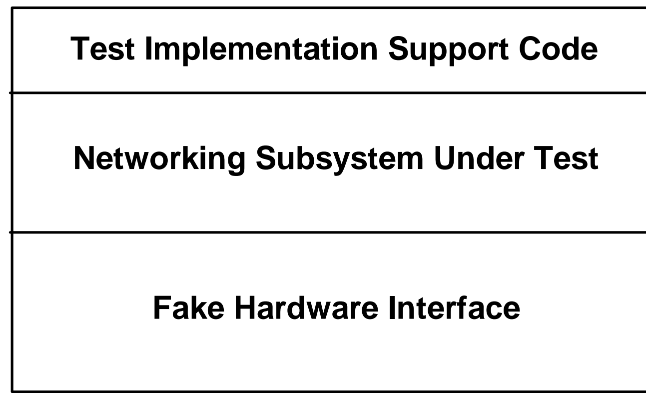
First problem: you may have trouble figuring out how to force a particular value to be delivered to some routine buried deep within the system. Test requirements easy to satisfy at the routine interface may be hard to satisfy from the system interface.

Dealing With the Problem (1): Test Subsystems in Isolation



One solution is to test the subsystem in isolation. As an example, consider a simple networking package embedded in some larger system. The networking subsystem accepts commands from the larger system. In response, it communicates with the hardware through some hardware interface code (such as a device driver).

One job of the networking subsystem is to accept packets of data from the hardware interface. Correct packets are the easy case. The tricky part of networking is dealing with packets that get lost (and have to be retransmitted by the other end), duplicate packets (after the other end retransmits the packet, the original packet arrives, followed by the retransmit), and corrupted packets. Real instances of such packets are hard to generate.



On this slide, the subsystem is tested in isolation. The hardware interface is replaced by a set of subroutines that return the kinds of packets needed by the tests you've designed. The test implementation support code calls the networking code in whatever way your tests need. This support code is either all or part of the test driver (the distinction is explained in the following slides).

18

Tests Might Be Linked Into the Support Code

```
test14()
{
/* Set up what the fake hardware interface will return */
/* Test support routines driven by test specifications */
fake_packet("some data", NORMAL);
fake_packet("some more data", RETRY);
fake_packet("even more data", CORRUPT_CHECKSUM);

/* Call the subsystem interface routine */
data = fetch_data(123);

/* Check correctness of subsystem result */
expect_strings(3, data, "some data", "some_more_data",
                        "even_more_data");
}
```

Here's an example of a test. It is code compiled with the test implementation support code. It first invokes some utility routines in the fake hardware interface. These routines tell that interface what to return when the networking subsystem calls it.

Then the test invokes the subsystem just as the normal system would.

Finally, it checks that the actual results are the same as the expected results.

In this case, the test implementation support code performs all the functions of a test driver. The advantage of this approach is that it can be easiest: you write only the support code absolutely essential for your tests.

Or the Support Code Might Provide a System Interface

```
COMTEST <test1 >test1.res

Input file contains:
    test packet "some data" normal
    test packet "some more data" retry
    test packet "even more data" corrupt
    transfer 3 host=dummyhost
```

- **Now can use system test drivers and managers**

The problem with the previous test is that you have to write code. Writing the code to set up the test and (especially) check the results can take more time than you'd like. If you could set up the test using data files, and check the results by comparing output files to stored reference files, you'd save time.

What sometimes happens is that internal test drivers "grow" an external interface. After a while, you get tired of writing a lot of code, so you decide to spend the time to make writing tests easier. In this slide's example, the test support code has a command-line interface. Test setup commands all begin with the "test" keyword. The three test commands describe the packets to be used in the test. The next command invokes the subsystem in the way the normal system would. The output is stored in a file, which is then compared to a reference file.

This test does the same thing as the one on the previous page, but it's faster to create and easier to use because it can use normal command-line-level tools like file comparison programs, test drivers, and test managers.

Which approach works best depends on the details of your system. The idea is to create the cheapest tests that are still maintainable.

Dealing With the Problem (2)

- **Generally unwise to test subsystems smaller than the "natural unit"**
  - **writing the drivers now is expensive**
  - **maintenance is horrendous**
- **Best solution: add testpoints to force test requirements**
  - **often a little code goes a long way**
  - **applies to system interface as well**
  - **(reference material later)**

Once you start dividing a system into subsystems, the question is where you stop. In the extreme, you could have test support code for every subroutine. (This is sometimes called "unit testing".) The problem is that writing those drivers is expensive. Moreover, changes to the subsystem tend to break many of the drivers, making maintenance terribly expensive. As a result, such unit tests tend to be abandoned in later releases, which misses the whole point of automation.

The type of unit testing most likely to work is one where tests for every source file are contained in that file. When the file is conditionally compiled for testing, it becomes an executable that you run to test the source. When the file is compiled normally, the test code is omitted. This is still rather expensive, but it's more likely to be maintained. It has some effectiveness problems that will be covered more fully in Part 2. Briefly, it makes it unlikely that you'll discover problems due to the interaction between this code and other code.

On balance, you should probably not break a "natural" subsystem into independently tested pieces. What's a natural subsystem? It's the code you would normally expect one person to develop or maintain because splitting it in two would require those two people to spend too much time talking to each other. In testing, such a split would mean too much support code that emulated the other half of the split.

If internal test requirements are still hard to satisfy, add internal test support code (here called "testpoints") to the subsystem. Testpoints are controlled from the subsystem interface and allow you to force particular paths through the subsystem, or to change particular variables in the subsystem. You will often find that it doesn't take much code to force the requirements you need. Testpoints will be discussed in more detail shortly.

21

Dealing With the Problem (3)

- **Weaken testing**
  - **–test manually (via debugger)**
  - **–do not expect these tests to be repeatable**
- **Rely only on inspections**

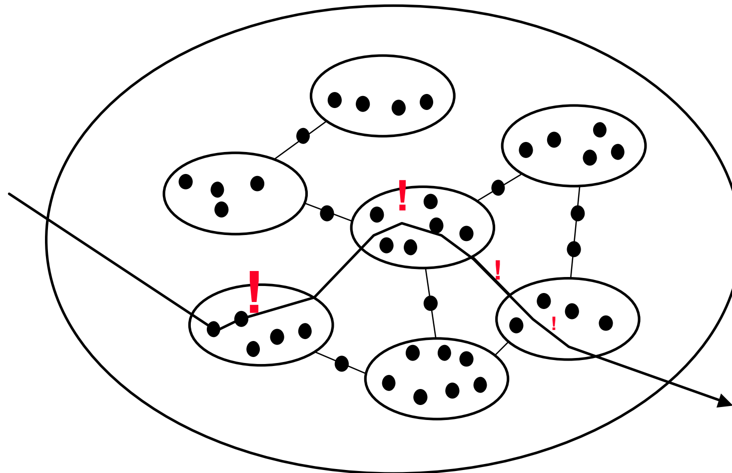
If inserting testpoints would still be too expensive or unmaintainable, you can test manually: start the debugger, give inputs to the code under test, and manually check the results. It’s unlikely you’ll repeat the tests in later releases (even if you intend to now), but you’ll get some immediate benefit from them.

Another alternative is to rely only on inspections. Do this when you decide the risk of an undiscovered defect is small enough that you’re willing to live without testing. Use your test requirement checklist as an inspection checklist giving you useful questions to ask.

Even if you do implement tests, the test requirement checklist can also be used in inspections.

22

Problem: Failure Propagation



Another problem you'll run into is failure propagation. If a failure occurs in some routine, will it likely be "damped out" before it reaches the system interface? For example, the entire system might produce a single boolean value, but internally it creates large and complex data structures. One of those data structures might be wrong in a way that happens to produce the right boolean value.

Dealing With the Problem

- **Test subsystem in isolation**
- **Add testpoints to reveal internal state**
- **Weaken or abandon testing**

If the large and complex data structure is visible from a natural subsystem interface, perhaps you should be implementing your tests from that interface.

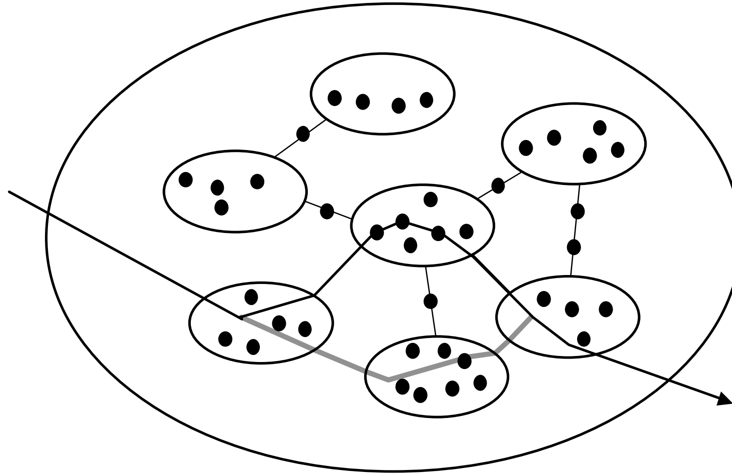
A less extreme solution is to print out that internal data structure so that you can check it directly. Often, it's not hard to identify data structures whose incorrectness might be masked, and not terribly difficult to print them out. This is another type of testpoint.

An alternative is to use the debugger to inspect the internal state. That makes the test easier, but less repeatable.

Simplest is to not test a requirement if you think any defect it detected wouldn't turn into a visible failure. In that case, you try to find the underlying fault through an inspection.

24

What If You Didn't Notice the Problem?



You won't always notice the problem. You may think you've exercised a particular routine in a particular way, but the test might actually take a completely different path. What then?

Dealing With the Problem

- **Coverage (testing the tests) will warn of many missed requirements**
  - **but not all**
- **Damped failures: you're out of luck**
  - **a risk tradeoff**
  - **missed failures due to large subsystem**

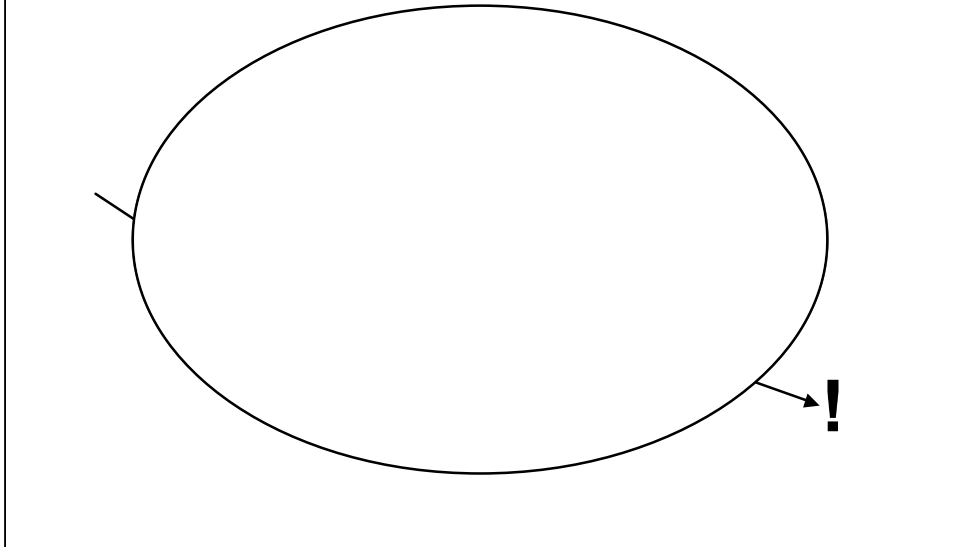
    vs.
  - **missed failures due to lack of time**

There is a step after test implementation where you test your tests by using code coverage tools to point to unexercised code. Unexercised code surely means unexercised requirements. (This use of coverage is discussed in Part 2.)

However, it won't discover all such problems. Some missed test requirements won't be detected by coverage. That's a risk you take.

A larger risk, especially in some subsystems, is of failures that don't propagate. Coverage won't help you there. Essentially, you have a risk tradeoff. There's a risk of missed failures because of large subsystems. But breaking the subsystem up into more-easily-tested pieces means you have more test support work to do. That inevitably means fewer tests - which means failures that are missed.

You choose your test implementation strategy to minimize risk. Since you can't quantify the risk, you do the best you can, which probably means testing from the system interface, or - if that clearly won't work - from a natural subsystem interface.



Larger subsystems cause another problem. Suppose the code does fail the test. Where's the fault? If you are testing a routine in isolation, you know the fault is in that routine. If you're testing a subsystem in isolation, you only know the fault's somewhere in the subsystem. If you're testing an entire system, it might be anywhere in the system.

Dealing With the Problem

- **Testpoints to reveal internal state (written for failure propagation) may help**
- **But this is really just an unavoidable consequence**
  - **beats debugging over the telephone**

Adding testpoints that print internal state may make the location of the fault more obvious. But, in most cases, this problem is just something you have to live with. Debugging a subsystem is harder than debugging a routine, but it beats debugging over the telephone because you couldn't write all the tests that were needed.

28

Reference Materials For Testpoints

This section shows some detailed examples of testpoints.

Reference 29

Testpoint Type 1: Control

```
dir1 = dir_sort(name1);
if (dir1.length == -1)...
```

```
dir_sort(char *name)
{
    extern int TP_dir_sort_unreadable;
    if (TP_dir_sort_unreadable)
    {
        dir.length = -1;
        return dir;
    }
    /* The real dir_sort code... */
```

Suppose that it were difficult to create an unreadable directory and make `dir_sort` return -1. One way to test `diff_dirs` would be to insert code within `dir_sort`. That code will check the value of a global variable. If that variable was set before the test was invoked, `dir_sort` will "fake" discovering an unreadable directory. If the variable is not set, `dir_sort` behaves normallly.

Reference

Avoiding Inefficiency

```
dir_sort(char *name)
{
#ifdef TESTING
    extern int TP_dir_sort_unreadable;
    if (TP_dir_sort_unreadable)
    {
        dir.length = -1;
        return dir;
    }
#endif TESTING
    ...
```

An objection to this scheme is that the variable's value is checked even when there's no testing going on. That can slow down the subsystem. The usual solution is to conditionally compile the testpoint. In C, that's done through the C preprocessor. "TESTING" is a compile time option. If set, the code within the #ifdef/#endif pair is compiled; otherwise, it's ignored.

You would "compile in" the testing code during testing, but leave it out during normal use.

This sometimes makes people nervous - you're not testing the exact executable you're shipping. It's not a problem if you have automated tests.

•Earlier in development, you test the conditionally compiled code.

•When development is finished, you rerun all the tests that don't require testpoints, just to make sure the code still works. It's quite rare for there to be a problem.

Reference 31

Another Example of Control

```
/* Packet driver interface functions */
bool pd_initialize_interface();
packet *pd_fetch();
bool pd_send(packet *pack);

#ifdef TESTING
pd_delay_next_packet(time_t amount);
pd_corrupt_next_packet_data();
#endif TESTING
```

Here's another example. It shows a functional interface instead of a global variable interface.

The top set of routines are the subsystem's normal interface routines. They've been augmented by some special-purpose testing routines.

The subsystem is used to fetch packets from and send packets to a network. The testpoint routines are used to simulate error conditions on the network. The first causes an incoming packet to be held up for some time. The second corrupts the packet in order to test the subsystem's error handling. (Both of them probably set variables that are read by the routine `pd_fetch`.)

Suppose that a test requirement were "packet delayed beyond timeout interval". A test of that requirement would call `pd_delay_next_packet` before calling `pd_fetch`.

If you were testing via the system interface, you would extend the system's user interface to include commands that called the testpoint routines.

Reference

Control: Summary

- **Can't satisfy a test requirement from the chosen interface?**
- **Add code to allow it to be satisfied**
- **Control that code from the chosen interface**

Testpoints are driven by test requirements. If you can't test a requirement from your chosen interface because it's too hard to drive internal code down the right path, add control testpoints to push the subsystem in the right direction. The control testpoints are themselves controlled from the chosen interface. Now you can do all your testing from that interface.

Reference 33

Testpoint Type 2: Visibility

```
bool
hash_table_insert(char *name, hash_table *ht)
{
    ...
    DEBUG_PRINT_HASH_TABLE(ht);
    return result;
}
```

- **Conditionally compiled code**
- **Turned on by runtime switch**

A visibility testpoint makes the internals of the subsystem more visible. In this example, a hash table insertion routine can be told to print out the value of the hash table. Your test can then check whether the element has been inserted in the right place, and that other entries in the hash table have not been corrupted.

As with the previous examples, the hash table is only printed if

•the code has been conditionally compiled to allow printing, and

•an appropriate runtime switch has been set.

What does the printing code look like?

Reference

Implementation

```
#ifdef DEBUG
    debug_print_hash_table(hash_table *ht)
    {
        /* A lot of print statements */
    }
#   define DEBUG_PRINT_HASH_TABLE(ht) \
       {if (ht_debugging) debug_print_hash_table(ht);}
#else
#   define DEBUG_PRINT_HASH_TABLE()
#endif
```

If the DEBUG flag is given to the compiler at compile time, the debugging routine is compiled in. Further, the macro DEBUG_PRINT_HASH_TABLE is defined as an IF that checks the runtime switch and calls the debugging routine.

If the DEBUG flag was not given, DEBUG_PRINT_HASH_TABLE turns into nothing, so there is no runtime penalty.

Notice that the compile-time switch is called DEBUG. That's because the same data-structure-printing code that makes debugging easier also makes testing easier. You might as well reuse it.

Reference

Another Example: Consistency Checking Code

```
bool
hash_table_insert(char *name, hash_table *ht)
{
    ...
    assert(hash_table_lookup(name, ht) == true);
    return result;
}
```

In the previous example, the hash table was printed so that expected results could be checked. Another option is to have the testpoint check the expected results itself. That's called an assertion.

In this example, the hash table insertion routine calls an assertion that checks that the name is in the hash table. Assertions, like other testpoints, can be conditionally compiled in. Except in the most time-critical code, it's a good idea to compile them into even the version you ship.

The decision between an assertion and a state-printing routine is one of efficiency: do whichever is simplest but still lets you check all the expected results you should check.

Reference

36

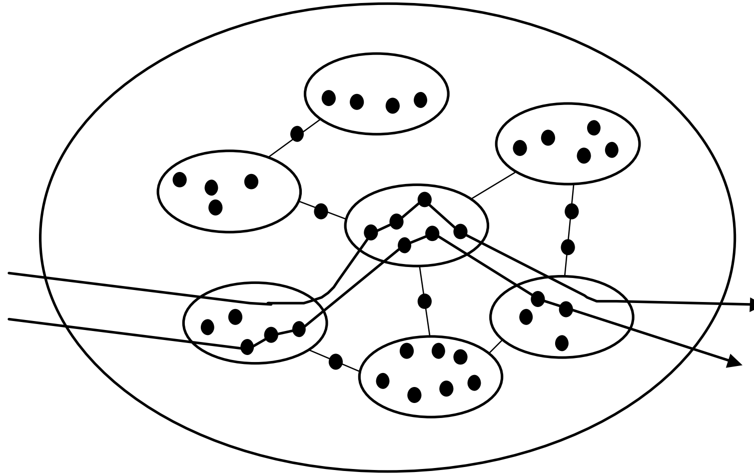
Visibility: Summary

- **Can't reliably check all expected results from the chosen interface?**
- **Add code to allow checking**
- **Control that code from the chosen interface**

Use visibility testpoints when there are internal results that you think might not reliably propagate to the chosen interface. Add code that reveals those results (or, in the case of an assertion, signals a failure). From the chosen interface, you can now control whether internals are visible there.

37

Final Note: Multiple Routines



You may have already noticed some inefficiency in test design. Each test was designed to satisfy one or more requirements of a single routine. Because the test is implemented through a larger interface, it will necessarily exercise other routines. Why not have it satisfy some of their test requirements along the way?

That will, in fact, be recommended in Part 2. When you're starting out, though, it's probably simpler to stick to one routine at a time. When you're comfortable with the process, start thinking about routines together.

38

Summary: Implementation Is All About Efficiency

- **Test implementation should make your life easier**
  - **you will have to rerun tests**
  - **manual vs. automated tests**
- **Automation will mean**
  - **+ faults discovered earlier after changes**
  - **– test implementation takes more time**
  - **– debugging is harder**
  - **this release's schedule <u>may</u> be hurt**
- **If you automate, do it right**

When you change the product and break it, you want to find the problem tomorrow at the latest. Tracking down a fault is much easier the day after it's created than a month after. To find faults early, you must rerun tests often. You can only do that if they're automated.

The downside is that you have to create the automated tests. That will take longer than manual testing. That means some failures will be discovered later in this release than with manual testing. Moreover, there's some extra cost now in debugging - it's often harder to debug an automated test. (Designing the test driver to aid debugging can help a lot.)

When there are many changes yet to go into the product before the next release, starting automation now may save you time before that release. But it can also delay that release (or result in less testing being done before the release date). Different people have had different experiences.

One thing is clear, though: if you automate and do it wrong - make unmaintainable automated tests - you'll pay all the costs and get few of the benefits.

39

What to Do Now

- Plan testing around time available
- Use multicondition and boundary testing on high-risk parts of your subsystem
- Automate if at all reasonable
  - another risk tradeoff
  - more testing now vs. more testing later
- Use the cookbook

With what you've learned so far, you should be able to:

• decide which routines to concentrate on,

• design tests for those routines, and

• automate those tests, after deciding how much automation is reasonable.

Go and do that. Don't forget to use the cookbook as a quick reference. After you're skilled at the Part 1 materials, adopt the extensions of part 2 of the course.