Extracted from:

Practices of an Agile Developer

Working in the Real World

The Pragmatic Programmers

Practices of an Agile Developer

Venkat Subramaniam
Andy Hunt

Pragmatic Bookshelf

ISBN 0-9745140-8-X

Printed on acid-free paper.

Fifth printing, January 2009

Version: 2009-3-13

31 Tell, Don't Ask

"Don't trust other objects. After all, they were written by other people, or even by you last month when you weren't as smart. Get the information you need from others, and then do your own calculations and make your own decisions. Don't give up control to others!"

"Procedural code gets information and then makes decisions. Object-oriented code tells objects to do things." Alec Sharp [Sha97] hit the nail on the head with that observation. But it's not limited to the object-oriented paradigm; any agile code should follow this same path.

As the caller, you should *not* make decisions based on the state of the called object and then change the state of that object. The logic you are implementing should be the called object's responsibility, not yours. For you to make decisions outside the object violates its encapsulation and provides a fertile breeding ground for bugs.

David Bock illustrates this well with the tale of the paperboy and the wallet.[9] Suppose the paperboy comes to your door, requesting his payment for the week. You turn around and let the paperboy pull your wallet out of your back pocket, take the two bucks (you hope), and put the wallet back. The paper boy then drives off in his shiny new Jaguar.

The paperboy, as the "caller" in this transaction, should simply tell the customer to pay $2. There's no inquiry into the customer's financial state, or the condition of the wallet, and no decision on the paperboy's part. All of that is the customer's responsibility, not the paperboy's. Agile code should work the same way.

Keep commands separate from queries

A helpful side technique related to *Tell, Don't Ask* is known as *command-query separation* [Mey97]. The idea is to categorize each of your functions and methods as either a command or a query and document them as such in the source code (it helps if all the commands are grouped together and all the queries are grouped together).

A routine acting as a command will likely change the state of the object and might also return some useful value as a convenience. A query just

9. http://www.javaguy.org/papers/demeter.pdf

Beware of Side Effects

Have you ever heard someone say, "Oh—we're just calling that method because of its side effects." That's pretty much on par with defending an odd bit of architecture by saying, "Well, it's like that because it used to...."

Statements such as these are clear warning signs of a fragile, not agile, design.

Relying on side effects or living with an increasingly twisted design that just doesn't match reality are urgent indications you need to redesign and refactor the code.

gives you information about the state of the object and does not modify the externally visible state of the object.

That is, queries should be side effect free as seen from the outside world (you may want to do some pre-calculation or caching behind the scenes as needed, but fetching the value of *X* in the object should not change the value of *Y*).

Mentally framing methods as *commands* helps reinforce the idea of *Tell, Don't Ask*. Additionally, keeping queries as side effect free is just good practice anyway, because you can use them freely in unit tests, call them from assertions, or from the debugger, all without changing the state of the application.

Explicitly considering queries separately from commands also gives you the opportunity to ask yourself *why* you're exposing a particular piece of data. Do you really need to do so? What would a caller do with it? Perhaps there should be a related command instead.

Tell, don't ask. *Don't take on another object's or component's job. Tell it what to do, and stick to your own job.*

What It Feels Like

Smalltalk uses the concept of "message passing" instead of method calls. *Tell, Don't Ask* feels like you're sending messages, not calling functions.

Keeping Your Balance

- Objects that are just giant data holders are suspect. Sometimes you need such things, but maybe not as often as you think.
- It's OK for a command to return data as a convenience (it'd be nice to be able to retrieve that data separately, too, if that's needed).
- It's not OK for an innocent-looking query to change the state of an object.

32

Substitute by Contract

"Deep inheritance hierarchies are great. If you need functionality from some other class, just inherit from it! And don't worry if your new class breaks things; your callers can just change their code. It's their problem, not yours."

A key way to keep systems flexible is by letting new code take the place of existing code without the existing code knowing the difference. For instance, you might need to add a new type of encryption to a communications infrastructure or implement a better search algorithm using the same interface. As long as the interface remains the same, you are free to change the implementation without changing any other code. That's easier said than done, however, so we need a little bit of guidance to do it correctly. For that, we'll turn to Barbara Liskov.

Liskov's Substitution principle [Lis88] tells us that "Any derived class object must be substitutable wherever a base class object is used, without the need for the user to know the difference." In other words, code that uses methods in base classes must be able to use objects of derived classes without modification.

What does that mean exactly? Suppose you have a simple method in a class that sorts a list of strings and returns a new list. You might invoke it like this:

```
utils = new BasicUtils();
...
sortedList = utils.sort(aList);
```

Now suppose you subclass the BasicUtils class and make a new sort() method that uses a much better, faster sort algorithm:

```
utils = new FasterUtils();
...
sortedList = utils.sort(aList);
```

Note the call to sort() is the same; a FasterUtils object is perfectly substitutable for a BasicUtils object. The code that calls utils.sort() could be handed a utils of either type, and it would work fine.

But if you made a subclass of BasicUtils that changed the meaning of sort—returning a list that sorted in reverse order, perhaps—then you've grossly violated the Substitution principle.

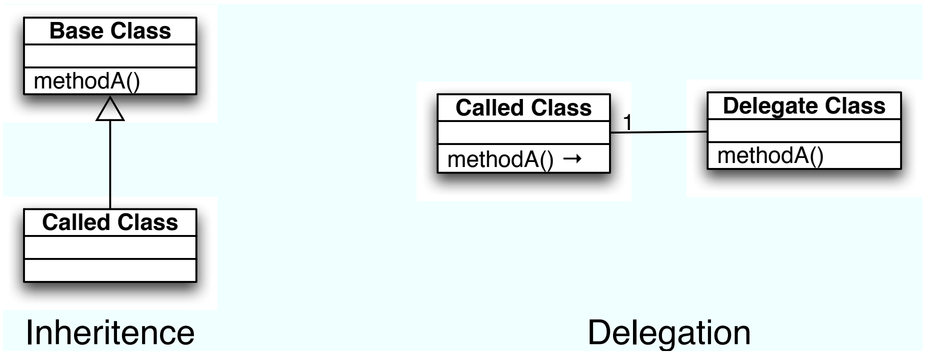


Figure 6.3: Delegation versus inheritance

To comply with the Substitution principle, your derived class services (methods) should *require no more, and promise no less*, than the corresponding methods of the base class; it needs to be freely substitutable. This is an important consideration when designing class inheritance hierarchies.

Inheritance is one of the most abused concepts in OO modeling and programming. If you violate the Substitution principle, your inheritance hierarchy may still provide code reusability but will not help with extensibility. The user of your class hierarchy may now have to examine the type of the object it is given in order to know how to handle it. As new classes are introduced, that code has to constantly be reevaluated and revised. That's not an agile approach.

But help is available. Your compiler may help you enforce the LSP, at least to some extent. For example, consider method access modifiers. In Java, the overriding method's access modifier must be the same or more lenient than the modifier of the overridden method. That is, if the base method is protected, the derived overriding method must be protected or public. In C# and VB .NET, the access protection of the overridden method and the overriding method are required to be the same.

Consider a class `Base` with a method `findLargest()` that throws an `IndexOutOfRangeException`. Based on the documentation, a user of this class will prepare to catch that exception if thrown. Now, assume you inherit the class `Derived` from `Base`, override the method `findLargest()`, and in the new method throw a different exception. Now, if an instance of Derived is used by code expecting an object of class Base, that code may receive an unexpected exception. Your Derived class is not substitutable wherever Base is used. Java avoids this problem by not allowing you to throw any new kind of checked exceptions from the overriding methods, unless the exception itself derives from one of the exception classes thrown from the overridden method (of course, for unchecked exceptions such as RuntimeException, the compiler won't help you).

Unfortunately, Java violates the Substitution principle as well. The java.util.Stack class derives from the java.util.Vector class. If you (inadvertently) send an object of Stack to a method that expects an instance of Vector, the elements in the Stack can be inserted or removed in an order inconsistent with its intended behavior.

Use inheritance for *is-a*; use delegation for *has-a* or *uses-a*

When using inheritance, ask yourself whether your derived class is substitutable in place of the base class. If the answer is no, then ask yourself why you are using inheritance. If the answer is to reuse code in the base class when developing your new class, then you should probably use composition instead. *Composition* is where an object of your class contains and uses an object of another class, delegating responsibilities to the contained object (this technique is also known as *delegation*).

Figure 6.3, on the previous page shows the difference. Here, a caller invoking methodA() in Called Class will get it automatically from Base Class via inheritance. In the delegation model, the Called Class has to explicitly forward the method call to the contained delegate.

When should you use inheritance versus delegation?

- If your new class can be used in place of the existing class and the relationship between them can be described as *is-a*, then use inheritance.
- If your new class needs to simply use the existing class and the relationship can be described as *has-a* or *uses-a*, then use delegation.

You may argue that in the case of delegation you have to write lots of tiny methods that route method calls to the contained object. In inheritance, you don't need these, because the public methods of the base class are readily available in the derived class. By itself, that's not a good enough reason to use inheritance.

You can write a good script or a nice IDE macro to help you write these few lines of code or use a better language/environment that supports a more automatic form of delegation (Ruby does this nicely, for instance).

Extend systems by substituting code. *Add and enhance features by substituting classes that honor the interface contract. Delegation is almost always preferable to inheritance.*

What It Feels Like

It feels sneaky; you can sneak a replacement component into the code base without any of the rest of the code knowing about it to achieve new and improved functionality.

Keeping Your Balance

- Delegation is usually more flexible and adaptable than inheritance.
- Inheritance isn't evil, just misunderstood.
- If you aren't sure what an interface really promises or requires, it will be hard to provide an implementation that honors it.

You might get the impression that experienced woodworkers never make mistakes. I can assure you that isn't true. Pros simply know how to salvage their goofs.

► Jeff Miller, furniture maker and author

Chapter 7

Agile Debugging

Even on the most talented agile projects, things will go wrong. Bugs, errors, defects, mistakes—whatever you want to call them, they will happen.

The real problem with debugging is that it is not amenable to a time box. You can time box a design meeting and decide to go with the best idea at the end of some fixed time. But with a debugging session, an hour, a day, or a week may come and go and find you no closer to finding and fixing the problem.

You really can't afford that sort of open-ended exposure on a project. So, we have some techniques that might help, from keeping track of previous solutions to providing more helpful clues in the event of a problem.

To reuse your knowledge and effort better, it can help to *Keep a Solutions Log*, and we'll see how on the following page. When the compiler warns you that something is amiss, you need to assume that *Warnings Are Really Errors* and address them right away (that's on page 132).

It can be very hard—even impossible—to track down problems in the middle of an entire system. You have a much better chance at finding the problem when you *Attack Problems in Isolation*, as we'll see on page 136. When something does go wrong, don't hide the truth. Unlike some government cover-up, you'll want to *Report All Exceptions*, as described on page 139. Finally, when you do report that something has gone awry, you have to be considerate of users, and *Provide Useful Error Messages*. We'll see why on page 141.

33 Keep a Solutions Log

"Do you often get that déjà vu feeling during development? Do you often get that déjà vu feeling during development? That's OK. You figured it out once. You can figure it out again."

Facing problems (and solving them) is a way of life for developers. When a problem arises, you want to solve it quickly. If a similar problem occurs again, you want to remember what you did the first time and fix it more quickly the next time. Unfortunately, sometimes you'll see a problem that looks the same as something you've seen before but can't remember the fix. This happens to us all the time.

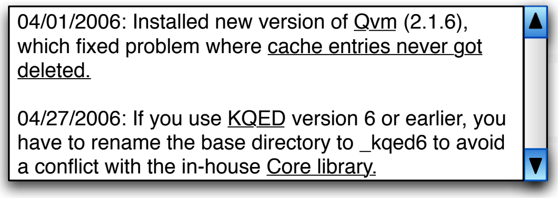
Can't you just search the Web for an answer? After all, the Internet has grown to be this incredible resource, and you might as well put that to good use. Certainly searching the Web for an answer is better than wasting time in isolated efforts. However, it can be *very* time-consuming. Sometimes you find the answers you're looking for; other times, you end up reading a lot of opinions and ideas instead of real solutions. It might be comforting to see how many other developers have had the same problem, but what you need is a solution.

Don't get burned twice

To be more productive than that, maintain a log of problems faced and solutions found. When a problem appears, instead of saying, "Man, I've seen this before, but I have no clue how I fixed it," you can quickly look up the solution you've used in the past. Engineers have done this for years: they call them *daylogs*.

You can choose any format that suits your needs. Here are some items that you might want to include in your entries:

- Date of the problem
- Short description of the problem or issue
- Detailed description of the solution
- References to articles, and URLs, that have more details or related information
- Any code segments, settings, and snapshots of dialogs that may be part of the solution or help you further understand the details



04/01/2006: Installed new version of Qvm (2.1.6), which fixed problem where cache entries never got deleted.

04/27/2006: If you use KQED version 6 or earlier, you have to rename the base directory to _kqed6 to avoid a conflict with the in-house Core library.

Figure 7.1: Example of a solutions log entry, with hyperlinks

Keep the log in a computer-searchable format. That way you can perform a keyword search to look up the details quickly. Figure 7.1 shows a simple example, with hyperlinks to more information.

When you face a problem and you can't find the solution in your log, remember to update your log with the new details as soon as you do figure out a solution.

Even better than maintaining a log is sharing it with others. Make it part of your shared network drive so others can use it. Or create a Wiki, and encourage other developers to use it and update it.

Maintain a log of problems and their solutions. *Part of fixing a problem is retaining details of the solution so you can find and apply it later.*

What It Feels Like

Your solutions log feels like part of your brain. You can find details on particular issues and also get guidance on similar but different issues.

Keeping Your Balance

- You still need to spend more time solving problems than documenting them. Keep it light and simple; it doesn't have to be publication quality.

- Finding previous solutions is critical; use plenty of keywords that will help you find an entry when needed.
- If a web search doesn't find *anyone* else with the same problem, perhaps you're using something incorrectly.
- Keep track of the specific version of the application, framework or platform where the problem occurred. The same problem can manifest itself differently on different platforms/versions.
- Record *why* the team made an important decision. That's the sort of detail that's hard to remember six to nine months later, when the decision needs to be revisited and recriminations fill the air.

34 Warnings Are Really Errors

“Compiler warnings are just for the overly cautious and pedantic. They’re just warnings after all. If they were serious, they’d be errors, and you couldn’t compile. So just ignore them, and let ’er rip.”

When your program has a compilation error, the compiler or build tool refuses to produce an executable. You don’t have a choice—you have to fix the error before moving on.

Warnings, unfortunately, are not like that. You can run the program that generates compiler warnings if you want. What happens if you ignore warnings and continue to develop your code? You’re sitting on a ticking time bomb, one that will probably go off at the worst possible moment.

Some warnings are benign by-products of a fussy compiler (or interpreter), but others are not. For instance, a warning about a variable not being used in the code is probably benign but may also allude to the use of some other incorrect variable.

At a recent client site, Venkat found more than 300 warnings in an application in production. One of the warnings that was being ignored by the developers said this:

```
Assignment in conditional expression is always constant;
did you mean to use == instead of = ?
```

The offending code was something like this:

```
if (theTextBox.Visible = true)
...
```

In other words, that **if** will always evaluate as true, regardless of the hapless theTextBox variable. It’s scary to see genuine errors such as this slip through as warnings and be ignored.

Consider the following C# code:

```
public class Base
{
  public virtual void foo()
  {
    Console.WriteLine("Base.foo");
  }
}
```

```
public class Derived : Base
{
  public virtual void foo()
  {
    Console.WriteLine("Derived.foo");
  }
}

class Test
{
  static void Main(string[] args)
  {
    Derived d = new Derived();
    Base b = d;
    d.foo();
    b.foo();
  }
}
```

When you compile this code using the default Visual Studio 2003 project settings, you'll see the message "Build: 1 succeeded, 0 failed, 0 skipped" at the bottom of the Output window. When you run the program, you'll get this output:

```
Derived.foo
Base.foo
```

But this isn't what you'd expect. You should see both the calls to foo() end up in the Derived class. What went wrong? If you examine the Output window closely, you'll find a warning message:

```
Warning. Derived.foo hides inherited member Base.foo
To make the current member override that implementation,
add the override keyword. Otherwise, you'd add the new keyword.
```

This was clearly an *error*—the code should use **override** instead of **virtual** in the Derived class's foo() method.[1] Imagine systematically ignoring warnings like this in your code. The behavior of your code becomes unpredictable, and its quality plummets.

You might argue that good unit tests will find these problems. Yes, they will help (and you should certainly use good unit tests). But if the compiler can detect this kind of problem, why not let it? It'll save you both some time and some headaches.

1. And this is an insidious trap for former C++ programmers; the program would work as expected in C++.

Find a way to tell your compiler to treat warnings as errors. If your compiler allows you to fine-tune warning reporting levels, turn that knob all the way up so no warnings are ignored. GCC compilers support the -Werror flag, for example, and in Visual Studio, you can change the project settings to treat warnings as errors.

That is the least you should do on a project. Unfortunately, if you go that route, you will have to do it on each project you create. It'd be nice to enable that more or less globally.

In Visual Studio, for instance, you can modify the project templates (see *.NET Gotchas* [Sub05] for details) so any project you create on your machine will have the option set, and in the current version of Eclipse, you can change these settings under Window → Preferences → Java → Compiler → Errors/Warnings. If you're using other languages or IDEs, take time to find how you can treat warnings as errors in them.

While you're modifying settings, set those same flags in the continuous integration tool that you use on your build machine. (For details on continuous integration, see Practice 21, *Different Makes a Difference*, on page 87.) This small change can have a huge impact on the quality of the code that your team is checking into the source control system.

You want to get all of this set up right as you start the project; suddenly turning warnings on partway through a project may be too overwhelming to handle.

Just because your compiler treats warnings lightly doesn't mean you should.

Treat warnings as errors. *Checking in code with warnings is just as bad as checking in code with errors or code that fails its tests. No checked-in code should produce any warnings from the build tools.*

What It Feels Like

Warnings feel like...well, warnings. They are warning you about something, and that gets your attention.

Keeping Your Balance

- Although we've been talking about compiled languages here, interpreted languages usually have a flag that enables run-time warnings. Use that flag, and capture the output so you can identify—and eliminate—the warnings.
- Some warnings can't be stopped because of compiler bugs or problems with third-party tools or code. If it can't be helped, don't waste further time on it. But this shouldn't happen very often.
- You can usually instruct the compiler to specifically suppress unavoidable warnings so you don't have to wade through them to find genuine warnings and errors.
- Deprecated methods have been deprecated for a reason. Stop using them. At a minimum, schedule an iteration where they (and their attendant warning messages) can be removed.
- If you mark methods you've written as deprecated, document what current users should do instead and when the deprecated methods will be removed altogether.

35 Attack Problems in Isolation

“Stepping line by line through a massive code base is pretty scary. But the only way to debug a significant problem is to look at the entire system. All at once. After all, you don’t know where the problem may be, and that’s the only way to find it.”

One of the positive side effects of unit testing (Chapter 5, *Agile Feedback*, on page 76) is that it forces you to layer your code. To make your code testable, you have to untangle it from its surroundings. If your code depends on other modules, you’ll use mock objects to isolate it from those other modules. In addition to making your code robust, it makes it easier to locate problems as they arise.

Otherwise, you may have problems figuring out where to even start. You might start by using a debugger, stepping through the code and trying to isolate the problem. You may have to go through a few forms or dialogs before you can get to the interesting part, and that makes it hard to reach the problem area. You may find yourself struggling with the entire system at this point, and that just increases stress and reduces productivity.

Large systems are complicated—many factors are involved in the way they execute. While working with the entire system, it’s hard to separate the details that have an effect on your particular problem from the ones that don’t.

The answer is clear: don’t try to work with the whole system at once. Separate the component or module you’re having problems with from the rest of the code base for serious debugging. If you have unit tests, you’re there already. Otherwise, you’ll have to get creative.

For instance, in the middle of a time-critical project (aren’t they all?), Fred and George found themselves facing a major data corruption problem. It took a lot of work to find what was wrong, because their team didn’t separate the database-related code from the rest of the application. They had no way to report the problem to the vendor—they certainly couldn’t email the entire source code base to them!

So, they developed a small prototype that exhibited similar symptoms. They sent this to the vendor as an example and asked for their expert opinion. Working with the prototype helped them understand the issues more clearly.

Plus, if they *weren't* able to reproduce the problem in the prototype, it would have shown them examples of code that actually worked and would have helped them isolate the problem.

Prototype to isolate

The first step in identifying complex problems is to isolate them. You wouldn't try to fix an airplane engine in midair, so why would you diagnose a hard problem in a part or component of your application while it's working inside the entire application? It's easier to fix engines when they're out of the aircraft and on the workbench. Similarly, it's easier to fix problems in code if you can isolate the module causing the problem.

But many applications are written in a way that makes isolation difficult. Application components or parts may be intertwined with each other; try to extract one, and all the rest come along too.[2] In these cases, you may be better off spending some time ripping out the code that is of concern and creating a test bed on which to work.

Attacking a problem in isolation has a number of advantages: by isolating the problem from the rest of the application, you are able to focus directly on just the issues that are relevant to the problem. You can change as much as you need to get to the bottom of the problem—you aren't dealing with the live application. You get to the problem quicker because you're working with the minimal amount of relevant code.

Isolating problems is not just something you do after the application ships. Isolation can help us when prototyping, debugging, and testing.

Attack problems in isolation. *Separate a problem area from its surroundings when working on it, especially in a large application.*

What It Feels Like

When faced with a problem that you have to isolate, it feels like searching for a needle in a tea cup, not a needle in a haystack.

2. This is affectionately known as the "Big Ball of Mud" design antipattern.

Keeping Your Balance

- If you separate code from its environment and the problem goes away, you've helped to isolate the problem.
- On the other hand, if you separate code from its environment and the problem *doesn't* go away, you've still helped to isolate the problem.
- It can be useful to *binary chop* through a problem. That is, divide the problem space in half, and see which half contains the problem. Then divide that half in half again, and repeat.
- Before attacking your problem, consult your log (see Practice 33, *Keep a Solutions Log*, on page 129).

36

Report All Exceptions

"Protect your caller from weird exceptions. It's your job to handle it. Wrap everything you call, and send your own exception up instead—or just swallow it."

Part of any programming job is to think through how things should work. But it's much more profitable to think about what happens when things *don't* work—when things don't go as planned.

Perhaps you're calling some code that might throw an exception; in your own code you can try to handle and recover from that failure. It's great if you can recover and continue with the processing without your user being aware of any problem. If you can't recover, it's great to let the user of your code know exactly what went wrong.

But that doesn't always happen. Venkat found himself quite frustrated with a popular open-source library (which will remain unnamed here). When he invoked a method that was supposed to create an object, he received a null reference instead. The code was small, isolated, and simple enough, so not a whole lot could've been messed up at the code level. Still, he had no clue what went wrong.

Fortunately it was open source, so he downloaded the source code and examined the method in question. It in turn called another method, and that method determined that some necessary components were missing on his system. This low-level method threw an exception containing information to that effect. Unfortunately, the top-level method quietly suppressed that exception with an empty **catch** block and returned a null instead. The code Venkat had written had no way of knowing what had happened; only by reading the library code could he understand the problem and finally get the missing component installed.

Checked exceptions, such as those in Java, force you to catch or propagate exceptions. Unfortunately, some developers, maybe temporarily, catch and ignore exceptions just to keep the compiler from complaining. This is dangerous—temporary fixes are often forgotten and end up in production code. You must handle all exceptions and recover from the failures if you can. If you can't handle it yourself, propagate it to your method's caller so it can take a stab at handling it (or gracefully communicate the information about the problem to users; see Practice 37, *Provide Useful Error Messages*, on the next page).

Sounds pretty obvious, doesn't it? Well, maybe it's not as obvious as you think. A story in the news not long ago talked about a major failure of a large airline reservations system. The system crashed, grounding airplanes, stranding thousands of passengers, and snarling the entire air transportation system for days. The cause? *A single unchecked SQL exception in an application server.*

Maybe you'd enjoy the fame of being mentioned on CNN, but probably not like that.

Handle or propagate all exceptions. *Don't suppress them, even temporarily. Write your code with the expectation that things will fail.*

What It Feels Like

You feel you can rely on getting an exception when something bad happens. There are no empty exception handlers.

Keeping Your Balance

- Determining who is responsible for handling an exception is part of design.
- Not all situations are exceptional.
- Report an exception that has meaning in the context of this code. A NullPointerException is pretty but just as useless as the **null** object described earlier.
- If the code writes a running debug log, issue a log message when an exception is caught or thrown; this will make tracking them down much easier.
- Checked exceptions can be onerous to work with. No one wants to call a method that throws thirty-one different checked exceptions. That's a design error: fix it, don't patch over it.
- Propagate what you can't handle.

37

Provide Useful Error Messages

"Don't scare the users, or even other programmers. Give them a nice, sanitized error message. Use something comforting like 'User Error. Replace, and Continue.'"

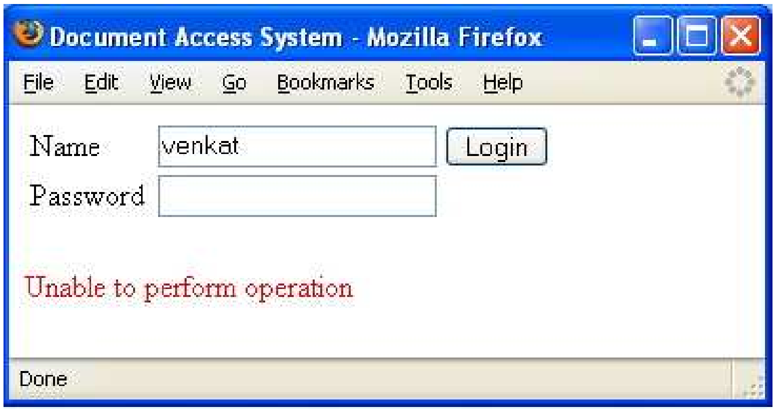
As applications are deployed and put into use in the real world, things will fail from time to time. A computation module may fail, for instance, or the connection to a database server may be lost. When you can't honor a user's request, you want to handle it gracefully.

When such an error occurs, is it enough to pop up a graceful, apologetic message to the user? Sure, a general message that informs the user about a failure is better than the application misbehaving or disappearing because of a crash (which leaves the user confused and wondering what happened). However, a message along the lines of "something went wrong" doesn't help your team diagnose the problem. When users call your support team to report the problem, you'd like them to report a lot of good information so you can identify the problem quickly. Unfortunately, with just a general error message, they won't be able to tell you much.

The most common solution to this issue is logging: when something goes wrong, have the application log details of the error. In the most rudimentary approach, the log is maintained as a text file. But you might instead publish to a systemwide event log. You can use tools to browse through the logs, generate an RSS feed of all logged messages, and so on.

While logging is useful, it is not sufficient: it might give you, the developer, information if you dig for it, but it doesn't help the hapless user. If you show them something like the message in Figure 7.2, on the following page, they are left clueless—they don't know what they did wrong, what they might do to work around it, or even what to report when they call tech support.

If you pay attention, you may find early warning signs of this problem during development. As a developer, you'll often pretend to be a user in order to test new functionality. If error messages are hard for you to understand or are not helpful to locate problems, imagine how hard it will be for your real users and your support team.


Document Access System - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Name
venkat
Login
Password
Done

Unable to perform operation

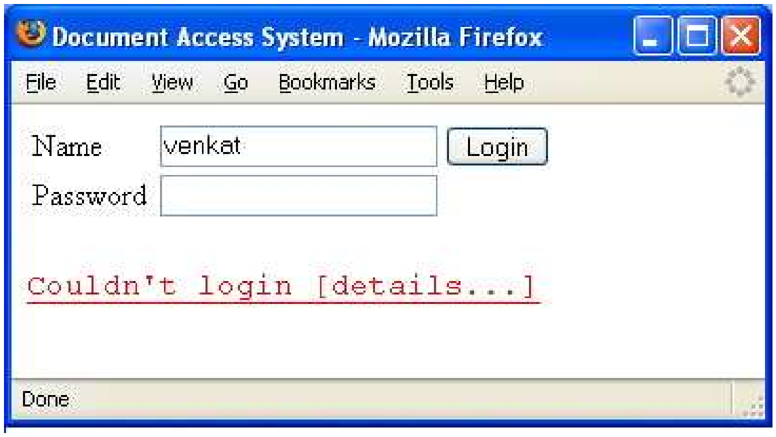
Figure 7.2: Exception message that doesn't help

For example, suppose the logon UI calls the middle tier of your application, which makes a request to its database tier. The database tier throws an exception because it couldn't connect to a database. The middle tier then wraps that exception into its own exception and passes that up. What should your UI tier do? It should at least let the user know there was a system error, and it's not due to any user input.

So the user calls up and tells you that he can't log on. How can you locate the actual problem? The log file may have hundreds of entries, and it's going to be hard to find the relevant details.

Instead, provide more details right in the message you give the user. Imagine being able to see exactly which SQL query or stored procedure messed up: this can make the difference between finding the problem and moving ahead versus wasting hours trying to find the problem blindly. On the other hand, providing the specific details about what went wrong during database connectivity doesn't help the users once the application is in production. It may well scare the living daylights out of some users.

On one hand, you want to provide users with a clean, high-level explanation of what went wrong so that they can understand the problem and perhaps pursue a workaround. On the other hand, you want to give them all the low-level, nitty-gritty details of the error so that you can identify the real problem in the code.


Document Access System - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Name
venkat
Login
Password
Done

Couldn't login [details...]

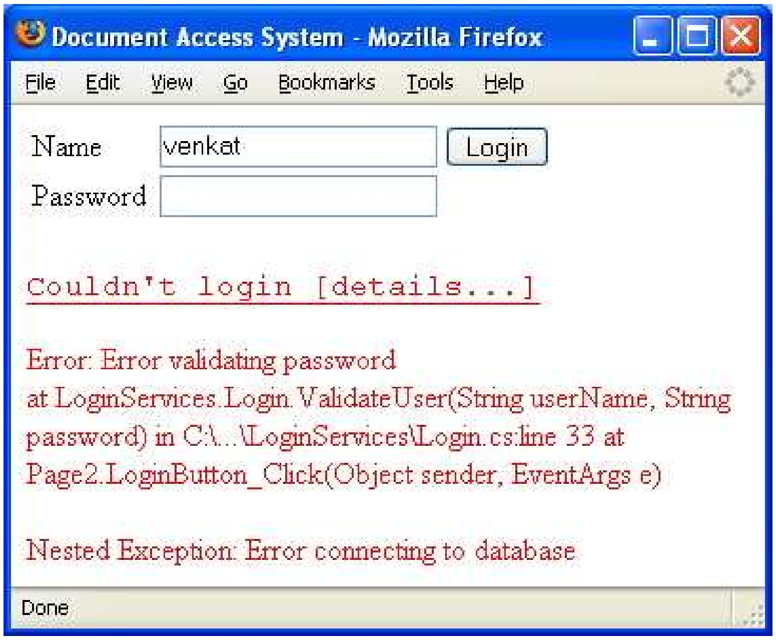
Figure 7.3: An exception message with link for more details

Here's one way to reconcile those disparate goals: Figure 7.3 shows a high-level message that appears when something goes wrong. This error message, instead of being just simple text, contains a hyperlink. The user, the developers, or the testers can then follow this link to get more information, as shown in Figure 7.4, on the following page.

When you follow the link, you'll see details about the exception (and all the nested exceptions). During development, you may want to simply display these details by default. When the application goes into production, however, you'll probably want to modify this so that instead of displaying these gory details directly to the users, you provide a link or some sort of handle or entry into your error log. Your support team can ask the user to click the error message and read the handle so they can quickly find the specific details in the log. In the case of a stand-alone system, clicking the link might email the details of what went wrong directly to your support department.

The information you've logged may contain not only the details about what went wrong but also a snapshot of the state of the system as well (the session state in a web application, for example).[3]

3. Some security-sensitive information should not be revealed or even logged; this includes items such as passwords, account numbers, etc.


Document Access System - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Name
venkat
Login
Password
Done

Couldn't login [details...]

Error: Error validating password
at LoginServices.Login.ValidateUser(String userName, String password) in C:\...\LoginServices\Login.cs:line 33 at Page2.LoginButton_Click(Object sender, EventArgs e)

Nested Exception: Error connecting to database

Figure 7.4: Complete details displayed for debugging

Using these details, your support group can re-create the situation that caused the problem, which will really help efforts to find and fix the issue.

Error reporting has a big impact on developer productivity as well as your eventual support costs. If finding and fixing problems during development is frustrating, take it as an early sign that you need a more proactive approach to error reporting. Debugging information is precious and hard to come by. Don't throw it away.

Present useful error messages. *Provide an easy way to find the details of errors. Present as much supporting detail as you can about a problem when it occurs, but don't bury the user with it.*

Distinguishing Types of Errors

Program defects. These are genuine bugs, such as NullPointerException, missing key values, etc. There's nothing the user or system administrators can do.

Environmental problems. This category includes failure to connect to a database or a remote web service, a full disk, insufficient permissions, and that sort of thing. The programmer can't do anything about it, but the user might be able to get around it, and the system administrator certainly should be able to fix it, if you give them sufficiently detailed information.

User error. No need to bother the programmer or the system administrators about this; the user just needs to try again, after you tell them what they did wrong.

By keeping track of what kind of error you are reporting, you can provide more appropriate advice to your audience.

What It Feels Like

Error messages feel useful and helpful. When a problem arises, you can hone in on the precise details of what went wrong, where.

Keeping Your Balance

- An error message that says "File Not Found" is not helpful by itself. "Can't open `/andy/project/main.yaml` for reading" is much more informative.
- You don't have to wait for an exception to tell you something went wrong. Use assertions at key points in the code to make sure everything is correct. When an assertion fails, provide the same good level of detail you would for exception reporting.
- Providing more information should not compromise security, privacy, trade secrets, or any other sensitive information (this is especially true for web-based applications).
- The information you provide the user might include a key to help you find the relevant section in a log file or audit trail.