Cleaner, more elegant, and wrong

devblogs.microsoft.com/oldnewthing/20040422-00

April 22, 2004

Raymond Chen

Just because you can't see the error path doesn't mean it doesn't exist.

Here's a snippet from a book on C# programming, taken from the chapter on how great exceptions are.

```
try {
  AccessDatabase accessDb = new AccessDatabase();
  accessDb.GenerateDatabase();
} catch (Exception e) {
  // Inspect caught exception
}
public void GenerateDatabase()
{
  CreatePhysicalDatabase();
  CreateTables();
  CreateIndexes();
}
```

> Notice how much cleaner and more elegant [this] solution is.

Cleaner, more elegant, and wrong.

Suppose an exception is thrown during CreateIndexes(). The GenerateDatabase() function doesn't catch it, so the error is thrown back out to the caller, where it is caught.

But when the exception left GenerateDatabase(), important information was lost: The state of the database creation. The code that catches the exception doesn't know which step in database creation failed. Does it need to delete the indexes? Does it need to delete the tables? Does it need to delete the physical database? It doesn't know.

So if there is a problem creating CreateIndexes(), you leak a physical database file and a table forever. (Since these are presumably files on disk, they hang around indefinitely.)

Writing correct code in the exception-throwing model is in a sense *harder* than in an error-code model, since *anything* can fail, and you have to be ready for it. In an error-code model, it's obvious when you have to check for errors: When you get an error code. In an exception

model, you just have to know that errors can occur anywhere.

In other words, in an error-code model, it is obvious when somebody failed to handle an error: They didn't check the error code. But in an exception-throwing model, it is not obvious from looking at the code whether somebody handled the error, since the error is not explicit.

Consider the following:

```
Guy AddNewGuy(string name)
{
 Guy guy = new Guy(name);
 AddToLeague(guy);
 guy.Team = ChooseRandomTeam();
 return guy;
}
```

This function creates a new Guy, adds him to the league, and assigns him to a team randomly. How can this be simpler?

Remember: Every line is a possible error.

What if an exception is thrown by "new Guy(name)"?
Well, fortunately, we haven't yet started doing anything, so no harm done.

What if an exception is thrown by "AddToLeague(guy)"?
The "guy" we created will be abandoned, but the GC will clean that up.

What if an exception is thrown by "guy.Team = ChooseRandomTeam()"?
Uh-oh, now we're in trouble. We already added the guy to the league. If somebody catches this exception, they're going to find a guy in the league who doesn't belong to any team. If there's some code that walks through all the members of the league and uses the guy.Team member, they're going to take a NullReferenceException since guy.Team isn't initialized yet.

When you're writing code, do you think about what the consequences of an exception would be if it were raised by each line of code? **You have to do this if you intend to write correct code**.

Okay, so how to fix this? Reorder the operations.

```
Guy AddNewGuy(string name)
{
 Guy guy = new Guy(name);
 guy.Team = ChooseRandomTeam();
 AddToLeague(guy);
 return guy;
}
```

This seemingly insignificant change has a big effect on error recovery. By delaying the commitment of the data (adding the guy to the league), any exceptions taken during the construction of the guy do not have any lasting effect. All that happens is that a partly-constructed guy gets abandoned and eventually gets cleaned up by GC.

General design principle: Don't commit data until they are ready.

Of course, this example was rather simple since the steps in setting up the guy had no side-effects. If something went wrong during set-up, we could just abandon the guy and let the GC handle the cleanup.

In the real world, things are a lot messier. Consider the following:

```
Guy AddNewGuy(string name)
{
 Guy guy = new Guy(name);
 guy.Team = ChooseRandomTeam();
 guy.Team.Add(guy);
 AddToLeague(guy);
 return guy;
}
```

This does the same thing as our corrected function, except that somebody decided that it would be more efficient if each team kept a list of members, so you have to add yourself to the team you intend to join. What consequences does this have on the function's correctness?

Raymond Chen

Follow