

Design/Pattern guidance to refactor my current design for unit testing

Source:

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.csharp/2007-08/msg02964.html>

- *From:* "sklett" <sklett@xxxxxxxxxxxxx>
 - *Date:* Tue, 21 Aug 2007 22:05:06 -0700
-

Hello all-

I would like to start unit testing my projects, I'm the sole developer in a startup with stacks of projects that needed to be done yesterday (nice sob story, I know) – point is I keep breaking things! Moving too fast to get a new feature out and I'm breaking stuff left and right. My first step to remedy this problem is to implement unit tests for my projects. I'm learning that is easier said than done.

The first project I decided to attempt is an authentication and authorization module (class library) that is consumed by 5+ different WinForms apps. When I originally designed the module, ease of implementation was high on the list considering I had over 5 projects that needed to use it. The result design is a single class called AuthMgr that implements an IAuthMgr interface. Here is the interface:

```
<code>
public interface IAuthMgr
{
    LoginResponse Login();
    LoginResponse AuthenticateUser(string username, string rawPassword, out
    UserAccount account);
    LoginResponse AuthenticateUser(string username, string rawPassword,
    short requiredRoleID, out UserAccount account);
    PasswordChangeResponse ChangePassword();
    void ResetPassword(UserAccount account);
    UserAccountCollection GetAccounts();
    UserRoleCollection GetRoles();
    UserRoleCollection GetAccountRoles(int accountID);
    void ManageAccounts();
    bool ValidateUserPassword(string rawPassword);
    bool ValidateUserPassword(UserAccount account, string rawPassword);
    bool MonitorSessionTimeout { get; set; }
    int SessionTimeoutDuration { get; set; }
    IPrincipal Principle { get; }
    IIdentity Identity { get; }
}

```

Design/Pattern guidance to refactor my current design for unit testing

</code>

I haven't gone through a cleanup pass yet, there is some unused stuff in there.

I will explain an aspect of the module; Login()

**** Login ****

- When the application consuming AuthMgr starts up, I instantiate a new instance of AuthMgr and make a call to Login().
- Login() displays a Form (LoginDlg) to collection Username, password, location, etc.
- Because I want to present any errors or warnings to the user in the context of the Login Dialog, I have LoginDlg handle the validation of the data entered (required fields) and also handle business rules (Password expired = show ChangePasswordDlg, Account disabled = show message box alerting user account is disabled, etc)
- LoginDlg takes a reference to the AuthMgr instance in it's constructor. It uses this reference to make calls to AuthMgr.AuthenticateUser(), etc.

Maybe some code would explain things better...

<code>

```
// AuthMgr
public LoginResponse Login()
{
    // null out the current authenticated user
    _authenticatedUser = null;

    // Show the login form
    Views.LoginDlg loginView = new Views.LoginDlg(this);
    loginView.StartPosition = FormStartPosition.CenterScreen;

    if(loginView.ShowDialog() == DialogResult.Cancel)
    {
        _exitHandler();

        return LoginResponse.UnspecifiedFailure;
    }

    _authenticatedUser = loginView.Account;
    Guard.ReferenceNotNull(_authenticatedUser,
        "successful login should yield valid UserAccount reference");

    // Add the login to the audit trail
    AddLoginToAuditTrail(_authenticatedUser.Id.Value,
        Environment.MachineName);

    // Start the activity monitor
    if(_monitorSessionActivity)
    {
        #if (!SKIP_GLOBAL_HOOKS)
```

Design/Pattern guidance to refactor my current design for unit testing

```
// Global hooks cause PAIN when breaking into the debugger,
// unless you REALLY need to debug the hook code (and related)
// functionality I would leave this option off while
// debugging.
StartMonitoringSessionActivity();
#endif
}

return LoginResponse.Success;
}

// LoginDlg
private void button1_Click(object sender, EventArgs e)
{
if(string.IsNullOrEmpty(textBox_Username.Text))
{
// Show error message
base.ShowMissingDataMessage("Username");
}

if(string.IsNullOrEmpty(textBox_Password.Text))
{
// Show error message
base.ShowMissingDataMessage("Password");
}

LoginResponse authenticationResponse = _authMgr.AuthenticateUser(
textBox_Username.Text,
textBox_Password.Text,
out _account);

// Check if the account password has expired
if(_account != null && _account.PasswordExpirationDate < DateTime.Now)
{
base.ShowGeneralErrorMessage("Your password has expired, you will be
prompted to create a new one");
PasswordChangeResponse changeResponse =
_authMgr.ChangePassword(_account);
return;
}

switch(authenticationResponse)
{
case LoginResponse.InvalidPassword:

textBox_Password.Clear();
textBox_Password.Focus();

// If this user hasn't been added to the failed attempts
// dictionary, add them now
if(!_failedAttempts.ContainsKey(textBox_Username.Text) == false)
```

Design/Pattern guidance to refactor my current design for unit testing

```
{
_failedAttempts.Add(textBox_Username.Text, 0);
}

// increment the failed attempts count for this username
_failedAttempts[textBox_Username.Text]++;

// Show the invalid password message
base.ShowInvalidPasswordMessage(_failedAttempts[textBox_Username.Text]);

// If the user has exceeded the number of failed attempts,
// lock their account and show the message
if (_failedAttempts[textBox_Username.Text] >=
Settings.Default.MaxFailedLoginAttempts)
{
_authMgr.LockoutAccount(_account);
base.ShowAccountLockedOutMessage(_account);
}
break;
case LoginResponse.PasswordExpired:
{
base.ShowGeneralErrorMessage("Your password has expired, you
will be prompted to create a new one");

PasswordChangeResponse changeResponse =
_authMgr.ChangePassword(_account);
if (changeResponse == PasswordChangeResponse.Success)
{
this.DialogResult = DialogResult.OK;
}
}
break;
}
case LoginResponse.Success:
this.DialogResult = DialogResult.OK;
break;

// Additional cases removed to keep things reasonably small...
}
}
</code>
```

Hopefully that will give you a general idea of how I've structured things. I'm starting to think that I might have a bad design on my hands, I will let you all judge that ;0)

Now, finally to my problem with Unit Testing. As you can imagine when I'm showing these dialogs it will break my unit tests because the modal dialogs will block indefinitely. After speaking with a friend that has experience with unit testing (I gave him a much simpler example) he suggested that I separate my concerns. For example, I would have something like this:

Design/Pattern guidance to refactor my current design for unit testing

```
<pseudo code>
interface ICredentialsCollector
{
string Username {get;}
string Password {get;}
bool GetCredentials();
}

LoginDialog : ICredentialsCollector
{
// IGetCredentials members
}

class Authenticate
{
private ICredentialsCollector _collector = null;

public Authenticate(ICredentialsCollector collector){ _collector =
collector};

bool AuthenticateUser()
{
if(_collector.GetCredentials())
{
// do stuff with the username and password
return true;
}

return false;
}
}

// In my application I would do this
ICredentialsCollector collector = new LoginDialog();
Authenticate auth = new Authenticate(collector);
auth.AuthenticateUser();

// This would show the dialog, etc, etc

// in my unit test I would do something like this
class MockLoginDialog : ICredentialsCollector
{
public bool GetCredentials()
{
_username = "username to test with";
_password = "password to test with";
return true;
}
}

ICredentialsCollector collector = new MockLoginDialog();
```

Design/Pattern guidance to refactor my current design for unit testing

```
Authenticate auth = new Authenticate(collector);  
auth.AuthenticateUser();  
<pseudo code>
```

What the above example achieves is the ability to either have my Authenticate class show a dialog or use a mock object for testing (or some other concrete class to obtain user credentials)

It also is completely incompatible with my current design. Furthermore, if I were to extrapolate the above "mini-design" to my entire module I would need to instantiate all of my dialogs (or mocks) up front, supply the AuthMgr class with a reference to each, etc. This would result in my original 2 lines of code being needed on the consuming application to many lines of code and it wouldn't make sense to anyone. The comment would be something like:

```
// This stuff is all here so I can unit test the authentication system
```

His suggestion makes sense to me, I can already imagine many other areas where this pattern would make a lot of my code cleaner. I should have given him the FULL requirement like I did here, I just didn't anticipate all the details.

I'm not exactly proud of my original design either, so I am very open to finding some middle ground.

What I want is a module with a very limited public interface that will implement and manage complex operations involving multiple Forms, MessageBoxes, etc that can also be unit tested following the execution path of the public interface as closely as possible.

I can imagine a design where AuthMgr is really just a container for many separate classes, each responsible for different operations and each taking dialog interfaces in their constructors. AuthMgr would instantiate these components and execute them in the correct order.. sort of a "macro". My unit test would test these individual components on there own, building mock dialog objects, etc. So while this design would not allow me to unit test AuthMgr.Login(), it would allow me to test all the pieces that AuthMgr.Login() is composed of. This is still not very good though, AuthMgr.Login() would require logic to handle error conditions and business rules, this logic would not be under test with this design.

Wow, this is a long thread. Apologies. I really hope someone out there understands what I'm after and picks this thread up, I would really appreciate the help.

Thanks for reading,
Steve

.