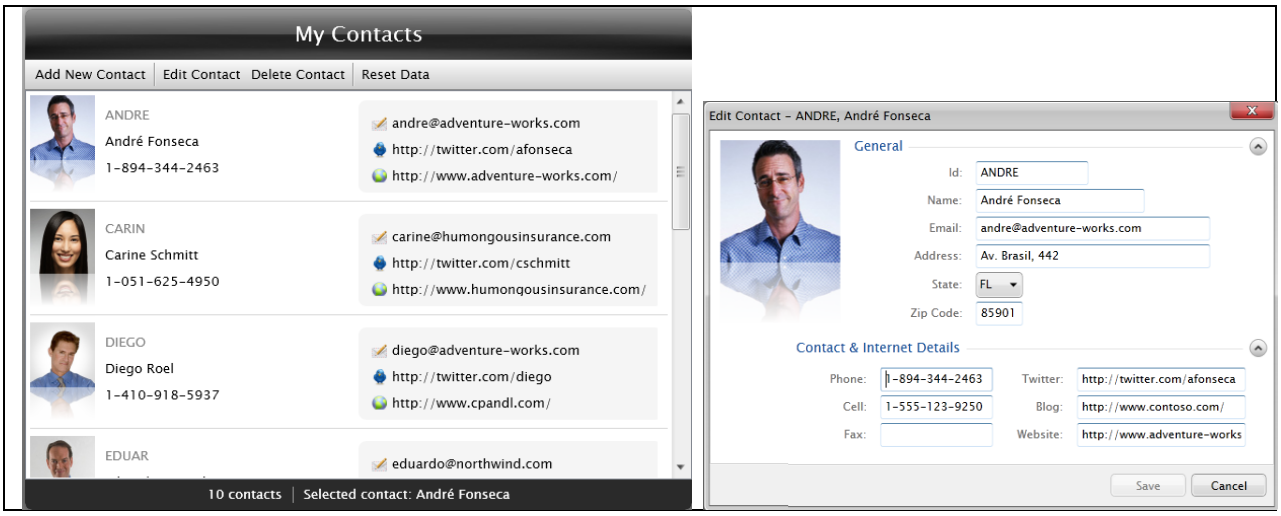
Add a Sandboxed Contact Editor to the Contacts Application

The first walkthrough described how to enable Intersoft's Contacts sample application with access to remote data using IdeaBlade's DevForce cross-platform data services product.

> This is the second in an evolving series of joint ClientUI/DevForce samples. Please visit links.ideablade.com/drc-ClientUI.DevForce to learn more about the series.

The sample application displays and edits customer contacts. A "contact" is a person with an address. The UI consists of two views: one lists all Contacts; the second is an overlay child window for editing the currently selected contact and address.



ContactsView **ContactEditDialogBox**

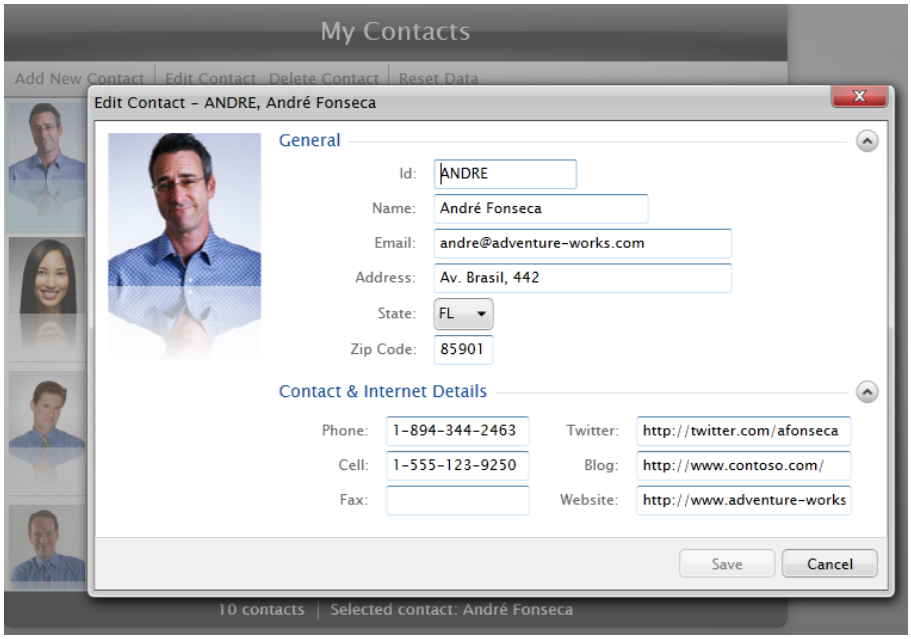
This "Sandbox Contact Editor" walkthrough extends that example by replacing the original Contact Editor with a "sandboxed" editor. The document discusses

- how the ContactEditor evolved,
- the potential problem posed by binding to the same physical Contact in both the main ContactsView and the ContactEditor,
- how a "sandboxed" ContactEditor relieves that problem
- the step-by-step changes to creating and using a "sandboxed" ContactEditor.

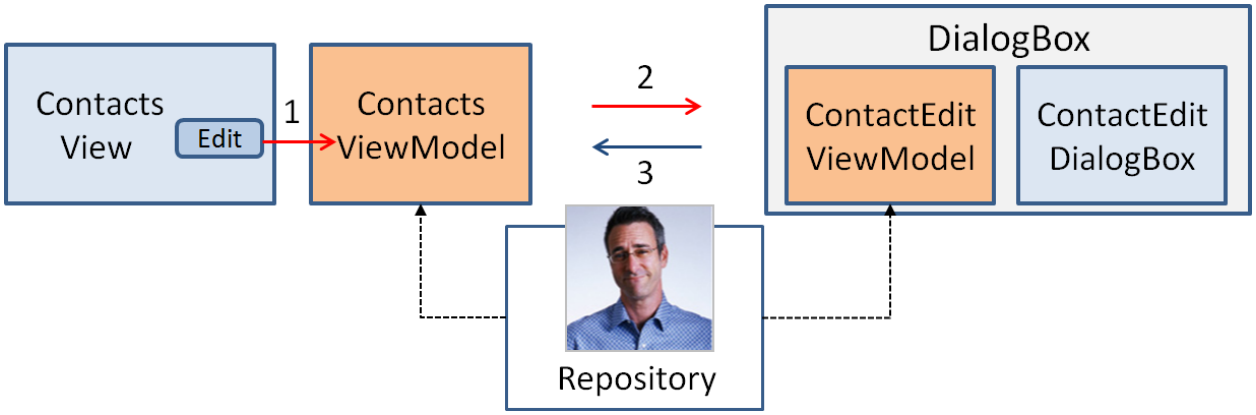
Evolution of the ContactEditor

Why the ContactsViewModel should not manage Contact editing

The ContactsView cannot edit a Contact on its own. Contact editing is done in a separate view, the ContactEditDialogBox which is hosted within a DialogBox modal window that overlays the ContactsView.



The developer must choose how to launch the DialogBox window and configure it to show the editing view. One possible approach is shown in the next figure.



1. The user clicks “edit” in the ContactsView; the view alerts the ContactsViewModel
2. The ContactsViewModel
    a. creates the window,
    b. creates the editing View,
    c. creates the editing ViewModel with the main repository
    d. puts the View and ViewModel inside the window,
    e. shows the window, and waits for the user to finish editing.

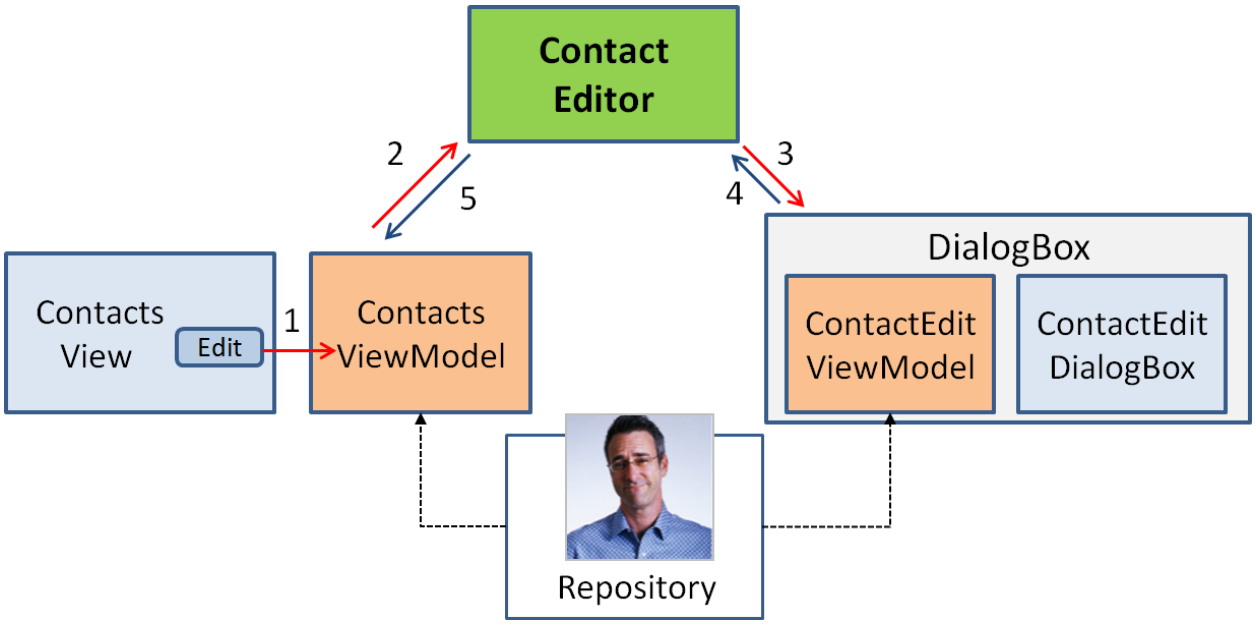
3. When the user finishes, having clicked either Save or Cancel, the DialogBox window reports that choice back to the ContactsViewModel. If the user cancelled, the ContactsViewModel must rollback the changes to the edited Contact.

A great deal of work is packed into steps #2 and #3. More troubling, it means that the ContactsViewModel is deeply involved in the details of managing the Contact editing process. Contact editing should not be a concern of the ContactsViewModel. Yet it is vulnerable to future changes in editing requirements and implementation details.

The tight coupling with the DialogBox makes the ContactsViewModel more difficult to test. A test of the ViewModel's response to an "edit" command would have to create a Silverlight control (the DialogBox) inside the test hosting framework. A simple question ("does the ViewModel respond to edit requests") turns into a complex, asynchronous test method, something to avoid.

Introducing the ContactEditor

The ClientUI sample takes a better approach. It delegates the details of DialogBox setup – everything in step #2 – to a helper class, a **ContactEditor**, as shown in the following diagram.



1. The user clicks "edit" in the ContactsView; the view alerts the ContactsViewModel
2. The ContactsViewModel calls the ContactEditor
3. The ContactEditor
    a. creates the window,
    b. creates the editing View,
    c. creates the editing ViewModel providing it with the repository
    d. puts the View and ViewModel inside the window,
    e. shows the window, and waits for the user to finish editing.

4. When the user closes the window the ContactEditor, if the user canceled, the ContactEditor rolls-back the changes to the Contact.
5. The ContactEditor cleans up and tells the ContactsViewModel what happened.

The ContactsViewModel only deals with a ContactEditor. More precisely, it deals with an implementation of the **IContactEditor** interface which has two asynchronous methods:

```
void EditExistingContact(Contact contact, Action<Contact> onOk, Action onCancel);
void AddNewContact(Action<Contact> onOk, Action onCancel);
```

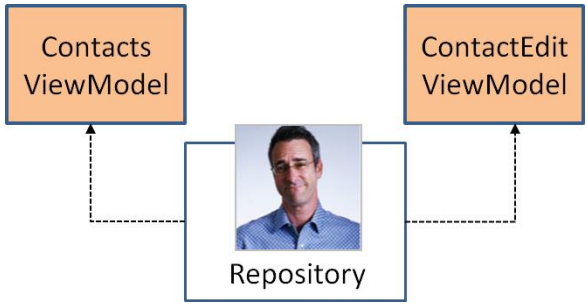
The EditExistingContact method is the case at hand but the interaction with the ContactEditor is much the same when editing a new Contact.

The ContactsViewModel constructor creates an instance of ContactEditor. That re-couples the ViewModel to a dependency that would be difficult to test. In a more sophisticated example, the IContactEditor would be injected into the ContactsViewModel rather than be created by it.

> Fortunately, the ContactEditor property of the ContactsViewModel is public. A test method could replace the ContactEditor with a mock or fake before it tested the edit command; the test would never try to create or show a real UI component.

ContactSandboxEditor

Notice in the previous diagram that both the ContactsViewModel and the ContactEditViewModel shared the same Contact entity and the same repository in which that Contact resides.



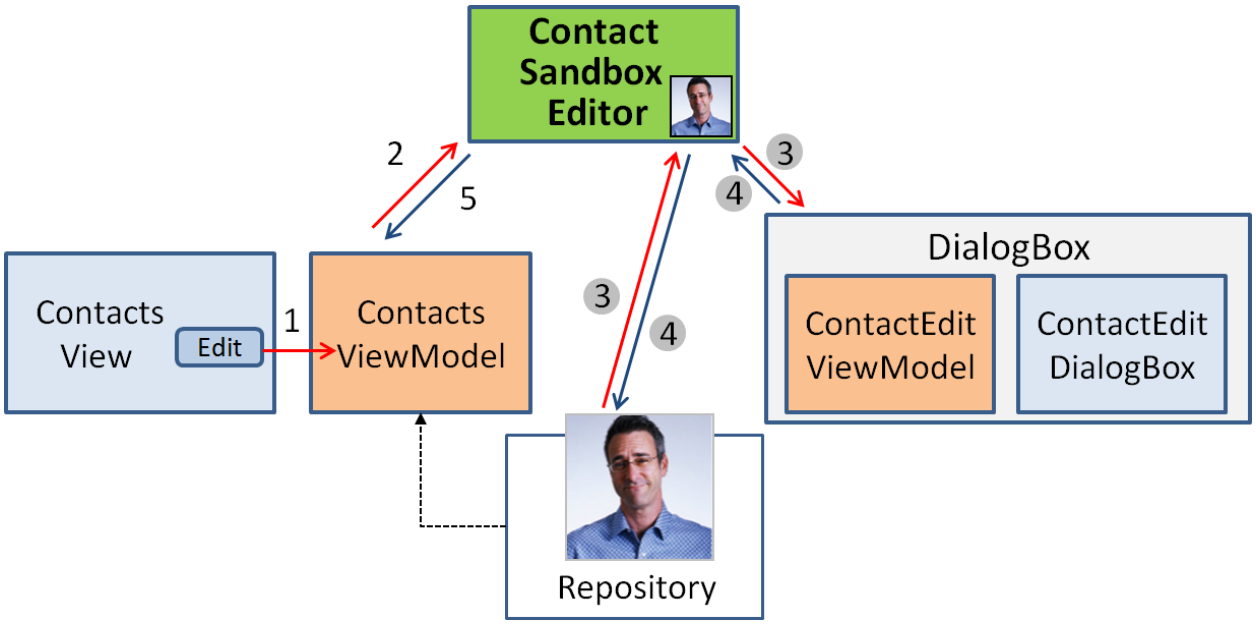
Such “entity sharing” is convenient but it also comes with implications. Because both views are bound to the same physical Contact instance, as the user makes changes in the ContactEditDialogBox view, those changes appear simultaneously in the ContactsView. In the following screenshot you can see that the name of the Contact is changing from “Andre” to “Bob” in both views.

Some users like this kind of instant value propagation. Other users find it confusing. They prefer that changes made inside the editor are confined to the editor. The ContactsView should only be updated after the user saves them.

The ContactEditViewModel will requires its own copy of the Contact (and the Contacts PostalAddress) during the edit process. If the user makes and save changes, the application updates the original Contact in the parent ContactsView.

> Adding a new Contact works the same way. The ContactEditViewModel works on a new Contact that is private to the editing session. If the user saves the new Contact, the application adds the new Contact to the parent ContactsView.

An editor that works with its own copy of data is said to be “sandboxed”, hence the name of the alternative editor described in this walkthrough document, the **ContactSandboxEditor**. The following diagram illustrates the key change that sets the sandboxed editor apart:



1. The user clicks “edit” in the ContactsView; the view alerts the ContactsViewModel
2. The ContactsViewModel calls the ContactEditor
3. The ContactEditor
    a. makes a copy of the Contact and imports it into its ***own private*** repository
    b. creates the window,
    c. creates the editing View,
    d. creates the editing ViewModel using the ***private*** repository
    e. puts the View and ViewModel inside the window,
    f. shows the window, and waits for the user to finish editing.
4. When the user closes the window the ContactEditor
    a. if the user saved, it copies the saved Contact back into the main Repository
    b. If the user canceled, the changes to the Contact copy are abandoned.
5. The ContactEditor cleans up and tells the ContactsViewModel what happened.

When the application uses the sandboxed editor, an unsaved change to the Contact name from “Andre” to “Bob” is no longer propagated to the parent ContactsView:

Once this edit is saved, the “ANDRE” Contact displayed in the parent ContactsView will become “Bob”.

This walkthrough describes the few application changes that are needed to turn the ContactEditor into a ContactSandboxEditor.

The Modification Plan

Adding the sandbox editor takes four steps:

1. Derive a new **ContactSandboxEditor** from the existing ContactEditor
2. Extend the **IContactsRepository** with two new methods
3. Implement those methods for the existing **ContactsRepository**
4. Tell the **ContactsViewModel** to use the new ContactSandboxEditor

Creating the ContactSandboxEditor

The sample code described by this walkthrough includes both the original ContactEditor and the new ContactSandboxEditor. The two editors differ in a few small details so deriving the sandbox editor from the original editor is an appropriate use of object inheritance.

> In a production application you would choose one or the other ; you wouldn’t have both and you wouldn’t use inheritance. The base ContactEditor has protected and virtual methods only to support the derived ContactSandboxEditor; your editor should make them private and non-virtual.

The entire ContactSandboxEditor code is as follows:

```
public class ContactSandboxEditor : ContactEditor
{
    public ContactSandboxEditor(IContactsRepository callerRepository)
        : base(ContactsRepository.CreateRepository())
    {
        CallerRepository = callerRepository;
    }

    private IContactsRepository CallerRepository { get; set;  }

    public override void AddNewContact(Action<Contact> onOk, Action onCancel)
```

```
    {
        Repository.ClearAllContacts();
        var contact = Repository.CreateContact();
        ShowContactDialogBox(contact, onOk, onCancel);
    }

    public override void EditExistingContact(
        Contact sourceContact, Action<Contact> onOk, Action onCancel)
    {
        Repository.ClearAllContacts();
        var contact = Repository.ImportContact(sourceContact);
        ShowContactDialogBox(contact, onOk, onCancel);
    }

    protected override Contact GetSavedContact(ContactEditViewModel viewModel)
    {
        return CallerRepository.ImportContact(viewModel.Contact);
    }

    // no need to rollback the sandbox repository
    protected override void RollbackChanges() {}

}
```

Observations on some key points follow:

- The sandbox editor maintains references to two repositories instead of one: a CallerRepository and a private repository.
- The CallerRepository is the main application repository used by the Contacts view.
- The Repository property used within the edit session by the ContactEditViewModel is the private repository, initialized in the base constructor via this statement: `base(ContactsRepository.CreateRepository())`
- The CallerRepository is the main application repository used by the Contacts view.
- When the editor is launched it always clears the private repository's memory of prior edited contacts, whether it will be adding a new contact or editing an existing contact; see the statement: `Repository.ClearAllContacts();`
- In the EditExistingContact method, the source contact cannot be used directly by the editor; otherwise contact changes would appear immediately in the ContactsView as they did before.

  The method imports the source Contact into the private repository instead with the line: `Repository.ImportContact(sourceContact);`

  The import returns a clone of the source Contact (and its related PostalAddress entity). The clone belongs to the private Repository and the editor's ContactEditViewModel works with this clone rather than the source Contact.

- The AddNewContact method acquires a new Contact from the Repository as it did in the original editor. The key difference: it is creating the contact with the private repository, not the main repository.

- When the user clicks the save button, the editor gets the contact from the ContactEditViewModel and returns it to the caller (the ContactsView). In the original, when the ContactsView and the editor shared the same physical Contact instance, the code simply returned the edited Contact entity. The edited Contact is a clone; it has to be merged back into the main repository (the CallerRepository) before the ContactsView can see the values.

  The GetSavedContact method use the ImportContacts again, this time in the opposite direction, back into the CallerRepository. When editing an existing contact, the import merges the clone values into that existing contact. If the edited contact is new, it is cloned into the CallerRepository as if it had been retrieved from the database.

- When the user cancels, there is no need to send a Contact back to the ContactsView. There is no need to rollback changes in the private repository either. The original editor had to rollback changes because it was modifying a Contact that was visible to the ContactsView; as this is no longer the case, the base Rollback method is overridden with a “do nothing” implementation.

Extend the IContactsRepository interface

The new ContactSandboxEditor requires two new repository features as we just saw:

```
/// <summary>
/// Import a given Contact and its graph of dependent related entities
/// into the repository, returning the corresponding "clone" of that Contact
/// from the repository.
/// </summary>
Contact ImportContact(Contact contact);

/// <summary>
/// Remove all Contacts and their dependent related entities from the repository.
/// </summary>
void ClearAllContacts();
```

Implement the ContactsRepository extensions

The methods are small:

```
public Contact ImportContact(Contact contact)
{
    if (contact == null || contact.EntityAspect.IsNullOrPendingEntity)
    {
        throw new ArgumentException("Imported Contact is null or nullo");
    }
    // Don't bother if already attached to repository's manager
    if (contact.EntityAspect.EntityManager == Manager) return contact;

    ImportEntities(new object[] { contact, contact.Address });
    return (Contact) Manager.FindEntity(contact.EntityAspect.EntityKey);
```

```
}


private void ImportEntities(IEnumerable entities)
{
    EnsureEntitiesAreUnmodified(entities);
    Manager.ImportEntities(entities, MergeStrategy.OverwriteChanges);
}

public void ClearAllContacts()
{
    Manager.RemoveEntities(typeof(PostalAddress), EntityState.AllButDetached);
    Manager.RemoveEntities(typeof(Contact), EntityState.AllButDetached);
}

private static void EnsureEntitiesAreUnmodified(IEnumerable entities)
{
    if (entities.Cast<EntityWrapper>()
        .Any(e => e.EntityAspect.EntityState != EntityState.Unchanged))
    {
        throw new InvalidOperationException(
            "Cannot import entities with pending changes");
    }
}
```

ImportContact begins with guard logic to catch a caller who provides a null Contact or the special "NullEntity" version of the Contact (the "Nullo") that stands in for a Contact in "No Contact" scenarios. There is also no point to a repository importing an entity that it already holds. The application will not deliver such bad Contacts in practice but it pays to be sure.

Having assured itself that the source Contact is worth importing, the method builds an array of the entities that represent the full Contact aggregate. The Contact aggregate consists of the root Contact entity and its related PostalAddress. Yyou can imagine a more complex aggregate with many related entities, an Order aggregate with an order header, order line items, and shipping addresses for example.

ImportEntities delegates to the DevForce EntityManager (aka, "Context") which has entity cloning and importing built-in. You have to decide what to do when (a) importing an entity that already exists in the target entity cache and (b) there are pending (unsaved) changes for that existing entity.

There can't be any pending changes in either repository in this application as it stands now but you never know what might happen in future. Overwriting pending changes seems like the right thing to do.

ImportEntities also checks (via **EnsureEntitiesAreUnmodified** to make sure that the entities it will import are unmodified. A DevForce EntityManager can import an entity that is unmodified, changed, or added. It's the application that expects only unmodified imports so it's the application that imposes this validation.

Switch to the Sandboxed Editor

The ContactsViewModel has a ContactEditor property that implements IContactEditor. The ViewModel constructor initializes the property with the SetDefaultContactEditor method as follows:

```
private void SetDefaultContactEditor()
{
    this.ContactEditor = new ContactEditor(Repository);
}
```

The ContactsViewModel uses the new ContactsSandboxViewModel after changing that line:

```
private void SetDefaultContactEditor()
{
    //this.ContactEditor = new ContactEditor(Repository);
    this.ContactEditor = new ContactSandboxEditor(Repository);
}
```

SetDefaultContactEditor Explained

This one line could have appeared in the constructor itself. It was extracted to its own method for three reasons:

1. To draw attention to the editor exchange
2. To facilitate future testing of the ViewModel
3. To hint at future dependency injection.

The first reason is pedagogical: the switch from one editor to the other is easier to see when performed in a method dedicated to the editor.

Testing is the second reason. The application has no automated tests at this stage. Testing is important; the code should be test-friendly even in the absence of tests . A test of the ContactsViewModel's response to an edit command should not launch a real ContactEditor; that would require the test to display the editor and wait for a user response which is infeasible in an automated test.

The ContactEditor property has a public setter. An "edit" test would set that property to a mock object before exercising the ViewModel's edit command. Meanwhile, the ContactEditor property is initialized to the real ContactEditor by default so that the application behaves as expected in production.

The third reason is also forward looking. In principle, the ViewModel should never ***create*** an external dependency such as the ContactEditor. The ContactEditor should be given to the ViewModel preferably as a parameter to the constructor . However, this ViewModel is created by the ContactsViewModel.xaml:

```
<Intersoft:UXPage.DataContext>
    <local:ContactsViewModel />
</Intersoft:UXPage.DataContext>
```

Objects created in Xaml must have parameterless constructors.

There's an architectural conflict between the limitations of xaml and the proper way to de-couple the ViewModel from its dependencies.

> The same thinking applies to the other dependencies: ContactsRepository, MessagePresenter, and DataSourceResetter.

This version of a tutorial application is too simple to warrant dependency injection techniques. What you see is a compromise that shapes the ViewMode codel in anticipation of such techniques without actually using them.

Summary

In this walkthrough, you considered three designs for a ContactEditor:

1. constructing the editor within the ViewModel,
2. delegating to an external ContactEditor class,
3. a sandboxed editor that shields the main view from Contact changes until the user saves them.

The first design was rejected. The original ContactEditor implemented the second design. This walkthrough shows how to implement the third design based on the second.

For more information about Intersoft's ClientUI, please visit www.clientui.com.
For more information about IdeaBlade's DevForce, please visit www.ideablade.com.

For questions and feedback on the presentation controls (View) and ViewModels, please visit Intersoft ClientUI Forum. For questions and feedback on model, services, and data programmability, please visit IdeaBlade DevForce Forum.