

This demo shows the simplicity and power of using Perst in a Silverlight application. You may need to have offline access with a data-driven application. While you can save data as XML files in Silverlight storage, you won't be able to have quick search, ease of use, integrity protection through transactions, and other benefits of proper database management.

Perst is a powerful object-oriented database system written entirely in C#. It's fast and simple to use. In that demo we illustrate how to use Perst in a Silverlight application that implements a simple CRM (Contacts – Leads – Activities) application. It has following functionality:

- Create, Read, Update, Delete 3 types of objects – Contact, Lead, Activity
- Quickly filter Leads by Contacts, and Activities by Leads
- Instant full-text search of objects
- Suggestions to complete typed words in the search box
- Quick generation of demo data
- Work with simple business objects
- LINQ queries over storage
- Multi-edit of selected objects
- Installation for offline use

Define Business Objects (Classes.cs)

// We are using additional base class to do automatic update to full-text index and be able to store in memory temporary objects(used for multi-edit). It's inherited from Perst Persistent class

```
public class Base : Persistent
{
    [NonSerialized]
    private bool isTemp;
    public bool IsTemp
    {
        get { return isTemp; }
        set { isTemp = value; }
    }
    protected static Database Database
    {
        get { return ((App)Application.Current).Database; }
    }
    public override void Deallocate()
    {
        Database.DeleteRecord(this);
    }
    public void Save()
    {
        Store();
        // Manually updating index for all fields marked with
        // [FullTextIndexable] attribute
        Database.UpdateFullTextIndex(this);
    }
}
// Example of declaring a class for which the objects will be persisted in the Perst database.
public class BusinessObject : Base
{
    [Indexable] // Create index to make fast search
    [FullTextIndexable] // Create full-text index over string fields
    [NonSerialized] // This is used to mark fields that shouldn't be stored at all
    public bool field;
    public bool Field
    {
        get { return field; }
        set { field = value; }
    }
}
```

See the source code for the complete class declarations in this application.

App Class: the entry point in the Silverlight application (App.xaml.cs)

```
public partial class App
{
    public App()
    {
        Startup += ApplicationStartup; // subscribe to start event
        Exit += ApplicationExit; // subscribe to exit event
        UnhandledException += ApplicationUnhandledException; // subscribe to unhandled exception
        InitializeComponent();
    }

    public Database Database { get; internal set; }

    // here we start
    private void ApplicationStartup(object sender, StartupEventArgs e)
    {
        // take storage instance for application
        using (var stor = IsolatedStorageFile.GetUserStoreForApplication())
        {
            // check that Perst DB file exists and initialize it if necessary
            if (stor.FileExists(DataGenerator.StorageName))
            {
                InitializePerstStorage();
            }
        }
        // initializing root visual object for application
        RootVisual = new MainPage { VerticalAlignment = VerticalAlignment.Stretch };
    }
    // finish application and close the database
    private void ApplicationExit(object sender, EventArgs e)
    {
        if (Database != null && Database.Storage != null)
            Database.Storage.Close();
    }
    // global exception handler
    private static void ApplicationUnhandledException(object sender,
        ApplicationUnhandledExceptionEventArgs e)
    {
        // Exit in case we are in debugger
        if (Debugger.IsAttached) return;
        // mark that this exception is handled
        e.Handled = true;
        // call error showing method in main GUI thread
        Deployment.Current.Dispatcher.BeginInvoke(() => ReportErrorToDom(e));
    }
    // initialize Perst storage
    internal void InitializePerstStorage()
    {
        // Create instance of Perst Storage
        var storage = StorageFactory.Instance.CreateStorage();
        // Initial database size set to 512KB, to fit in Silverlight Isolated Storage
        storage.SetProperty("perst.file.extension.quantum", 512 * 1024);
        // Step of storage extension 256KB to have less fragmentation on disk
        storage.SetProperty("perst.extension.quantum", 256 * 1024);
        // Open Storage
        storage.Open(DataGenerator.StorageName, 0);

        // Now we create a high-level wrapper on top of the storage to use the
        // Relational access mode of Perst
        //(Perst's lower level interface can be used to implement your own scheme of
        // database access, e.g. navigate the objects and their relationships directly).
```

```

//Create Database wrapper over Perst Storage
//Here, false means not multithreaded and true means automatic creation of tables
Database = new Database(storage, false, true, new FullTextSearchHelper(storage));

//Turn off auto-index creation (defined manually)
Database.EnableAutoIndices = false;
}
// Show error in browser window
private static void ReportErrorToDom(ApplicationUnhandledExceptionEventArgs e)
{
// Create error message text
var errorMsg = e.ExceptionObject.Message
    + e.ExceptionObject.StackTrace.Replace("'", "'').Replace("\r\n", @"\n");
// Show error by JavaScript window
HtmlPage.Window.Eval("throw new Error(\"Unhandled Error in Silverlight Application \"
    + errorMsg + "\");");
}
}

```

Main Control (MainPage.xaml.cs)

```

public partial class MainPage
{
// Cache of detail panels
private readonly Dictionary<DataGrid, DetailPanel> cacheDetailPanels
    = new Dictionary<DataGrid, DetailPanel>();
// create object to delay showing data in detail panel
private readonly Delayer detailDelayer;
private object currentDataGrid; // reference to currently selected grid

public MainPage()
{
InitializeComponent();
SizeChanged += OnThisSizeChanged; // subscribe to resize event

// subscribe to MouseEnter event of all grids
gridContact.MouseEnter += GridOnMouseEnter;
gridLead.MouseEnter += GridOnMouseEnter;
gridActivity.MouseEnter += GridOnMouseEnter;

// subscribe to GotFocus event of all grids
gridContact.GotFocus += DataGridGotFocus;
gridLead.GotFocus += DataGridGotFocus;
gridActivity.GotFocus += DataGridGotFocus;

//We need to show generated data in grids

//During application start and after search we need to show contacts in grid.
//Fill in Contacts grid with all contacts we have in the database

// if Database is initialized, load data in contacts grid
if (Database != null)
    gridContact.ItemsSource = Database.GetTable<Contact>().ToObservableCollection();

// tune delayer time for all grids
Delayer.DelayMilliseconds = 300;
// create delayer object for contacts grid
var contactDelayer = new Delayer();
// subscribe to event of contact selection change
gridContact.SelectionChanged += (sender, e) => contactDelayer.Action = RefreshLeads;
// create delayer object for leads grid
var leadDelayer = new Delayer();
// subscribe to event of lead selection change
gridLead.SelectionChanged += (sender, e) => leadDelayer.Action = RefreshActivities;
}
}

```

```

        // check buttons state on selection changes in grids
        gridContact.SelectionChanged += (sender, e) => CheckButtons();
        gridLead.SelectionChanged += (sender, e) => CheckButtons();
        CheckButtons();

        detailDelayer = new Delayer();
    }
    private static Database Database
    {
        get { return ((App)Application.Current).Database; }
    }
    private bool IsSearchableState
    {
        get { return !tbSearch.IsEmpty; }
    }
    private static Storage Storage
    {
        get { return Database.Storage; }
    }
}

// Add database record in DB, grid and details
private void AddItem(IPersistent obj, DataGrid dataGrid)
{
    Database.AddRecord(obj); // Adding new record to Database
    // Add object in grid
    ((IList)dataGrid.ItemsSource).Insert(0, obj);
    // select newly added object
    dataGrid.SelectedItem = obj;
    // show new object in details panel
    ShowDetailPanel(dataGrid);
    // focus first input in details
    ((DetailPanel)swDetail.Content).FocusFirstTextBox();
}

// clear the search box
private void bClearSearch_Click(object sender, RoutedEventArgs e)
{
    tbSearch.Clear();
}

// create and add new Activity
private void bNewActivity_Click(object sender, RoutedEventArgs e)
{
    // get currently selected lead
    var currentLead = (Lead)gridLead.SelectedItem;
    // create and add (in db and grid) new Activity instance
    AddItem(new Activity { Lead = currentLead }, gridActivity);
}

// create and add new Contact object
private void bNewContact_Click(object sender, RoutedEventArgs e)
{
    AddItem(new Contact(), gridContact);
}

// create and add new Lead object
private void bNewLead_Click(object sender, RoutedEventArgs e)
{
    // get current selected contact
    var currentContact = (Contact)gridContact.SelectedItem;
    // create and add (in db and grid) new Lead instance
    AddItem(new Lead { Contact = currentContact }, gridLead);
}

```

```

}

// check buttons state
private void CheckButtons()
{
    // turn off «Clear», «New Contact» buttons if Database is not initialized
    bClearDB.IsEnabled = Database != null;
    bNewContact.IsEnabled = Database != null;

    // turn off «New Lead» if no contacts are selected
    bNewLead.IsEnabled = gridContact.SelectedItem != null;
    // turn off «New Activity» if no Leads are selected
    bNewActivity.IsEnabled = gridLead.SelectedItem != null;
}

// clear the database
private void ClearDBClick(object sender, RoutedEventArgs e)
{
    // create cleanup dialog
    var clearPopup = new ClearPopup();
    // subscribe to dialog close event
    clearPopup.Closed += (sender1, e1) =>
    {
        // if OK is clicked, reload the contacts grid (others will follow automatically)
        if (clearPopup.DialogResult == true)
            RefreshContact();
        // refresh buttons state
        CheckButtons();
    };
    // show cleanup dialog
    clearPopup.Show();
}

// implement grid focus logic
private void DataGridGotFocus(object sender, RoutedEventArgs e)
{
    // load details panel for selected item.
    // Necessary here to overcome Silverlight 3 known bug related to duplicate event
    if (currentDataGrid == sender)
        ShowDetailPanel((DataGrid)sender);
}

// logic for changes in grid rows
private void DataGrid_RowEditEnded(object sender, DataGridRowEditEndedEventArgs e)
{
    // if the edit finished without cancelling, call commit method
    if (e.EditAction != DataGridEditAction.Commit) return;

    // get a link to the persistent object
    var persistent = e.Row.DataContext as Base;
    if (persistent == null) return;

    // Save Item to Storage
    persistent.Save();
    // Commit changes
    Storage.Commit();
}

// Load data in details panel with delay
private void DataGrid_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    // delegate for delayed load
    detailDelayer.Action = () =>
    {

```

```

        // check that cache has required grid
        if (!cacheDetailPanels.ContainsKey((DataGrid)sender)) return;
        // set a target to display
        cacheDetailPanels[(DataGrid)sender].Target =
            ((DataGrid)sender).SelectedItem;
    };
}

// generate demo data
private void GenerateDBClick(object sender, RoutedEventArgs e)
{
    // create dialog object
    var generatorPopup = new GeneratorPopup();
    // do actions after dialog close
    generatorPopup.Closed += (sender1, e1) =>
    {
        // On OK refresh contacts grid (others will do the same automatically)
        if (generatorPopup.DialogResult == true)
            RefreshContact();
        CheckButtons();
    };
    // show dialog
    generatorPopup.Show();
}

// set current active grid on mouse enter
private void GridOnMouseEnter(object sender, MouseEventArgs e)
{
    currentDataGrid = sender;
}

// refresh list of activities
private void RefreshActivities()
{
    // don't refresh list in case something typed in search box
    if (IsSearchableState) return;
    // reset current list of activities
    gridActivity.ItemsSource = null;
    IEnumerable<Activity> res = null;
    // if database exists, get data
    if (Database != null)
    {
        // Find and fill in activities where Activity.Lead is from the list of selected ones
        // or where Activity.Lead.Contact from selected contacts

        if (gridLead.SelectedItem != null)
        {
            var leads = gridLead.SelectedItems.Cast<Lead>();

            // The following is similar to forming an SQL SELECT query
            // Note the 'from' and 'where'

            res = (from activity in Database.GetTable<Activity>()
                // Load activities
                where leads.Contains(activity.Lead)
                // by selected lead
                select activity);
        }

        // On event of changed contacts selection, we need to filter out Leads
        // and Activities.
        // Find and fill in Leads that have Contacts selected in grid
        if (res == null && gridContact.SelectedItem != null)
        {
            var contacts = gridContact.SelectedItems.Cast<Contact>();
            res = (from activity in Database.GetTable<Activity>()

```

```

        // Load activities
        where activity.Lead != null &&
            contacts.Contains(activity.Lead.Contact)
        // by selected Contact
        select activity);
    }
}
if (res != null)
{
    // convert results to proper collection type
    var result = res.ToObservableCollection();
    // set activity data source
    gridActivity.ItemsSource = result;
}
else
{
    gridActivity.ItemsSource = null;
}
}

//That's it - we have shown our data from storage based on user selection.

// refresh contacts list
private void RefreshContact()
{
    // don't refresh list in case something typed in search box
    if (IsSearchableState) return;
    var contacts = Database != null ?
        Database.GetTable<Contact>().ToObservableCollection() : // Reload all contacts
        null;
    gridContact.ItemsSource = contacts;
    // refresh Leads
    RefreshLeads();
}

// refresh Leads
private void RefreshLeads()
{
    if (IsSearchableState) return;
    var contacts = gridContact.SelectedItems.Cast<Contact>();
    gridLead.ItemsSource = null;

    var res = Database != null ?
        (from lead in Database.GetTable<Lead>()
         // Loading Leads
         where contacts.Contains(lead.Contact)
         // by selected Contact
         select lead).ToObservableCollection() :
        null;
    gridLead.ItemsSource = res;
    // refresh Activities
    RefreshActivities();
}

// search by typed words
private void Search()
{
    // if database is not initialized, exit
    if (Database == null) return;

    // To find items by full-text search, we need to do following?
    // When user presses Enter in search box we find items with typed text
    // The arguments below will limit the number of results to 1000 and
    // limit the request time to 2000 milliseconds (most of the time takes just 10ms)

```

```

var prefixes = Database.SearchPrefix(tbSearch.Text, 1000, 2000, false);

var contacts = new ObservableCollection<Contact>();
var leads = new ObservableCollection<Lead>();
var activities = new ObservableCollection<Activity>();

var arrayRes = new List<FullTextSearchHit>();
if (prefixes != null) arrayRes.AddRange(prefixes.Hits);
// sort objects by found type
foreach (var hit in arrayRes)
{
    if (hit.Document is Contact)
    {
        if (!contacts.Contains((Contact)hit.Document))
            contacts.Add((Contact)hit.Document);
    }
    else if (hit.Document is Lead)
    {
        if (!leads.Contains((Lead)hit.Document))
            leads.Add((Lead)hit.Document);
    }
    else if (hit.Document is Activity)
    {
        if (!activities.Contains((Activity)hit.Document))
            activities.Add((Activity)hit.Document);
    }
}
// set data to corresponding grids
gridContact.ItemsSource = contacts;
gridLead.ItemsSource = leads;
gridActivity.ItemsSource = activities;
}

// show details by specified grid
private void ShowDetailPanel(DataGrid dataGrid)
{
    // skip this method if we are already showing same thing
    if (cacheDetailPanels.ContainsKey(dataGrid) && swDetail.Content
        == cacheDetailPanels[dataGrid]) return;

    DetailPanel detail;
    // get proper details panel for grid and create new one if it's absent in cache
    if (!cacheDetailPanels.TryGetValue(dataGrid, out detail))
    {
        Type typeObj;
        if (dataGrid == gridContact)
            typeObj = typeof(Contact);
        else if (dataGrid == gridLead)
            typeObj = typeof(Lead);
        else if (dataGrid == gridActivity)
            typeObj = typeof(Activity);
        else
            throw new ArgumentOutOfRangeException("dataGrid");
        // create new details panel and save in cache
        cacheDetailPanels[dataGrid] = detail = new DetailPanel(typeObj, dataGrid);
    }

    // set new data source
    if (detail.Target != dataGrid.SelectedItems)
        detail.Target = dataGrid.SelectedItems;

    // show details panel
    swDetail.Content = detail;
}

```

```

// event handler for search box typing
private void tbSearch_SearchChanged(object sender, EventArgs e)
{
    // search in case there is something to search
    if (!tbSearch.IsEmpty)
        Search();
    else
        RefreshContact(); // reload contacts in case search box empty
}
}

```

Details panel (DetailPanel.cs)

```

public class DetailPanel : StackPanel, INotifyPropertyChanged
{
    // create drop-down controls cache
    private readonly Dictionary<ItemsControl, Type> dropdowns
        = new Dictionary<ItemsControl, Type>();
    // initial target list
    private IList target;

    public DetailPanel(Type typeObj, DataGrid dataGrid)
    {
        if (typeObj == null) throw new ArgumentNullException("typeObj");
        if (dataGrid == null) throw new ArgumentNullException("dataGrid");
        // type of object shown in details panel
        TypeObj = typeObj;
        // grid to take out details
        DataGrid = dataGrid;
        Init();
    }

    public DataGrid DataGrid { get; private set; }

    public IList Target
    {
        get
        {
            if (target == null)
                target = new List<object>();
            return target;
        }
        set
        {
            // set preliminary targets list
            SetTarget(value);
            // reset state of Cover controls
            ResetControls();
            // find data object for connection with data source
            EvaluateDataContext();
            // tell system about changed properties
            InvokePropertyChanged(new PropertyChangedEventArgs("Target"));
            InvokePropertyChanged(new PropertyChangedEventArgs("Title"));
        }
    }

    public string Title
    {
        get
        {
            if (Target == null) return "";
            switch (Target.Count)
            {
                case 0:
                    return string.Format("No {0} Selected", TypeObj.Name);
            }
        }
    }
}

```

```

        case 1:
            return string.Format("{0} Details", TypeObj.Name);
        default:
            return string.Format("{0} {1}s Details", Target.Count, TypeObj.Name);
    }
}

public Type TypeObj { get; private set; }

#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged;

#endregion

// Utility method to split word by capital letters
private static string SeparateCapitalWords(IEnumerable<char> name)
{
    var array = name.ToList();
    var res = new List<char>();
    foreach (var c in array)
    {
        if (res.Count > 1 && c >= 'A' && c <= 'Z')
            res.Add(' ');
        res.Add(c);
    }
    return new string(res.ToArray());
}

public void FocusFirstTextBox()
{
    UpdateLayout();
    // find first cover control and set his focus
    foreach (var child in Children)
    {
        if (!(child is CoverControl)
            || !(((CoverControl)child).Control is TextBox)) continue;
        ((CoverControl)child).Control.Focus();
        break;
    }
}

// delete with confirmation selected objects
private void DeleteOnClick(object sender, RoutedEventArgs e)
{
    // exit if no items selected
    if (DataGrid.SelectedItems == null) return;
    // confirmations
    if (MessageBox.Show(
        string.Format("Delete record - {0}?",
            (DataGrid.SelectedItems.Count == 1
                ? DataGrid.SelectedItems[0]
                : string.Format("{0} items",
                    DataGrid.SelectedItems.Count))), "Delete",
        MessageBoxButton.OKCancel) != MessageBoxResult.OK) return;
    var selected = new ArrayList();
    // prepare list for deletion
    foreach (var item in DataGrid.SelectedItems)
        selected.Add(item);
    // Delete objects
    foreach (var item in selected)
    {
        // remove objects from grid
        ((IList)DataGrid.ItemsSource).Remove(item);
        // Remove Deleted object from Database
    }
}

```

```

        ((Persistent)item).Deallocate();
    }
    //Commit all changes to DB since previous Commit (you can also rollback those changes)
    ((App)Application.Current).Database.Storage.Commit();
}

// find data object for connection with data source
private void EvaluateDataContext()
{
    Base context = null;
    // if there is only 1 target - that is the context
    if (Target.Count == 1)
        context = (Base)Target[0];
    //if there are many targets we need to prepare special object for multi-edit
    else if (Target.Count > 1)
    {
        var type = Target[0].GetType();
        context = (Base)Activator.CreateInstance(type); // create instance by type
        context.IsTemp = true; // mark object as temporary
    }
    // subscribe context to property change events
    if (context is INotifyPropertyChanged)
    {
        ((INotifyPropertyChanged)context).PropertyChanged -= OnPropertyChanged;
        ((INotifyPropertyChanged)context).PropertyChanged += OnPropertyChanged;
    }
    // unsubscribe old context
    if (DataContext is INotifyPropertyChanged)
        ((INotifyPropertyChanged)DataContext).PropertyChanged -= OnPropertyChanged;

    DataContext = null;
    // if context is not empty then reload drop-downs
    if (context != null)
        RefreshDropDowns();
    else
        ResetDropDowns();

    DataContext = context;
    // initialise context with common data in case it's temporary
    if (DataContext is Base && ((Base)DataContext).IsTemp)
        IntersectProperties(context);
}

// initialize panel
private void Init()
{
    var detail = this;
    // iterate over properties and automatically create controls for them
    foreach (var propertyInfo in TypeObj.GetProperties(BindingFlags.DeclaredOnly
        | BindingFlags.Public | BindingFlags.Instance))
    {
        detail.Children.Add(
            new TextBlock { Text = SeparateCapitalWords(propertyInfo.Name) });
        FrameworkElement element;
        var binding = new Binding(propertyInfo.Name);
        if (propertyInfo.CanWrite)
        {
            // bind writable properties
            binding.Mode = BindingMode.TwoWay;
            // bind date-time properties
            if (typeof(DateTime).IsAssignableFrom(propertyInfo.PropertyType))
            {
                var dtp = new DatePicker();
                dtp.SetBinding(DatePicker.SelectedDateProperty, binding);
            }
        }
    }
}

```

```

        element = new CoverControl { Control = dtp, Name = propertyInfo.Name };
    }
    // bind enumerations
    else if (typeof(Enum).IsAssignableFrom(propertyInfo.PropertyType))
    {
        var cb = new ComboBox();
        cb.SetBinding(Selector.SelectedItemProperty, binding);
        var propertyInfo1 = propertyInfo;
        Utilities.FillEnums(cb, propertyInfo1.PropertyType);
        element = new CoverControl { Control = cb, Name = propertyInfo.Name };
    }
    // bind reference type properties
    else if (typeof(Persistent).IsAssignableFrom(propertyInfo.PropertyType))
    {
        var cb = new ComboBox();
        cb.SetBinding(Selector.SelectedItemProperty, binding);
        var propertyInfo1 = propertyInfo;
        dropdowns[cb] = propertyInfo1.PropertyType;
        element = new CoverControl { Control = cb, Name = propertyInfo.Name };
    }
    // text properties binding
    else
    {
        var tb = new TextBox();
        tb.SetBinding(TextBox.TextProperty, binding);
        element = new CoverControl { Control = tb, Name = propertyInfo.Name };
    }
}
else
{
    // bind one way read-only properties
    binding.Mode = BindingMode.OneWay;
    var tb = new TextBlock();
    tb.SetBinding(TextBlock.TextProperty, binding);
    element = tb;
}
detail.Children.Add(element);
}

var grid = new Grid { Margin = new Thickness(0, 50, 0, 0) };
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());

var b = new Button { Content = "Delete" };
b.Click += DeleteOnClick;
b.SetValue(Grid.ColumnProperty, 2);
grid.Children.Add(b);

detail.Children.Add(grid);
}

// find common value for same property of several selected objects
private void IntersectProperties(Base context)
{
    // iterate by controls in panel
    foreach (var c in Children)
    {
        var child = c as CoverControl;
        if (child == null) continue;
        // get properties by name
        var propInfo = target[0].GetType().GetProperty(child.Name);
        if (propInfo == null)
            throw new ArgumentNullException(

```

```

        string.Format("Property with name {0} not found.", child.Name));
    if (!propInfo.CanWrite) continue;
    var firstObject = true;
    object val = null;
    // iterate by targets
    foreach (var o in target)
    {
        var nextVal = propInfo.GetValue(o, null);
        if (firstObject)
        {
            val = nextVal;
            firstObject = false;
            continue;
        }
        if (Equals(nextVal, val)) continue;
        // if values are different showing cover instead of value
        child.IsShowingCover = true;
        break;
    }
    // if cover is not set that means all values are the same
    // and it can be saved in temporary object
    if (!child.IsShowingCover)
        propInfo.SetValue(context, val, null);
}
}

private void InvokePropertyChanged(PropertyChangedEventArgs e)
{
    var handler = PropertyChanged;
    if (handler != null) handler(this, e);
}

private void OnPropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var baseSender = (Base)sender;
    // find control with changed value
    var coverControl =
        (CoverControl)
        (from child in Children
         where child is CoverControl && ((CoverControl)child).Name == e.PropertyName
         select child).First();
    // hide control cover
    coverControl.IsShowingCover = false;
    // if this is multiselection and the object is temporary, then we need to
    // set value of all underlying objects, otherwise just save the object in DB
    if (baseSender.IsTemp)
        WriteValueToTargets(baseSender, e.PropertyName);
    else
        baseSender.Save();
}

private void RefreshDropDowns()
{
    // refresh drop-down controls
    foreach (var dd in dropdowns)
        Utilities.FillObjects(dd.Key, dd.Value, Target);
}
// reset all controls
private void ResetControls()
{
    foreach (var c in Children)
    {
        var child = c as CoverControl;
        if (child == null) continue;
    }
}

```

```

        child.IsShowingCover = false;
        child.AllowShowCover = Target.Count > 1;
    }
}
// clear values in drop-down controls
private void ResetDropDowns()
{
    foreach (var dropdown in dropdowns)
        dropdown.Key.Items.Clear();
}
// save targets list
private void SetTarget(IList value)
{
    target.Clear();
    foreach (var o in value)
        target.Add(o);
}
// save value of detail panel to multi-edited objects property
private void WriteValueToTargets(object sender, string paramName)
{
    // get property by name
    var pInfo = sender.GetType().GetProperty(paramName);
    if (pInfo == null) return;
    // get new value of property from changes to the temporary object
    var val = pInfo.GetValue(sender, null);
    // save value in each object
    foreach (Base obj in target)
    {
        pInfo.SetValue(obj, val, null);
        obj.Save();
    }
}
}
}

```

DataGenerator.cs

```

public class DataGenerator
{
    public const int OneObjectAvgSize = 800; // avg bytes per 1 obj

    private readonly string[] companies = "General Motors,Exxon Mobil,...".Split(',');
    private readonly string[] firstNames = "Jacob,Emma,Michael,...".Split(',');
    private readonly string[] lastNames = "Smith,Johnson,Williams,...".Split(',');

    private readonly Random random = new Random();

    public DataGenerator(Database dbBase)
    {
        if (dbBase == null) throw new ArgumentNullException("dbBase");
        Database = dbBase;
    }

    public int CountActivities { get; private set; }
    public int CountContacts { get; private set; }
    public int CountLeads { get; private set; }
    public Database Database { get; private set; }
    public bool IsComplete { get; set; }

    public const string StorageName = "PerstDemoDB.dbs";

    public void Clear()
    {
        IsComplete = false;
        try
        {

```

```

        Database.DropTable(typeof(Contact));
        Database.DropTable(typeof(Lead));
        Database.DropTable(typeof(Activity));
        Database.Storage.Commit();
        InvokeClearComplete(EventArgs.Empty);
    }
    finally
    {
        IsComplete = true;
    }
}

public void DropStorage()
{
    IsComplete = false;
    try
    {
        var storage = ((App)App.Current).Database.Storage;
        storage.Close();
        using (var store = IsolatedStorageFile.GetUserStoreForApplication())
        {
            if (store.FileExists(StorageName))
            {
                store.DeleteFile(StorageName);
            }
        }
        ((App)App.Current).Database = null;
        InvokeClearComplete(EventArgs.Empty);
    }
    finally
    {
        IsComplete = true;
    }
}

public void Generate(int maxObjects)
{
    bool changedAfterCommit = false;
    int i = 0;
    while (i < maxObjects)
    {
        changedAfterCommit = true;
        var firstName = GetRandomElement((IList<string>)firstNames);
        var lastName = GetRandomElement((IList<string>)lastNames);
        var company = GetRandomElement((IList<string>)companies);
        i++;
        //Create a Contact
        var contact = new Contact
        {
            FirstName = firstName,
            LastName = lastName,
            Company = company,
            Address = string.Format("Street {0}, b. #{1}",
                random.Next(500), random.Next(1000)),
            Email = string.Format("{0}.{1}@{2}.com",
                firstName, lastName, company.Replace(" ", ""))
        };
        var leadsPerContact = random.Next(2, 16);
        //Now for every contact we need to generate new Leads in cycle (cycle omitted)
        for (var j = 0; j < leadsPerContact && i < maxObjects; j++)
        {
            i++;
            var lead = new Lead
            {

```

```

        Name = string.Format("Lead Name #{0}", j + 1),
        Contact = contact,
        Amount = random.Next(50),
        ExpectedClose = DateTime.Now.AddDays(random.Next(150)),
        Probability = ((double)random.Next(100)) / 100,
    };
    var activitiesPerLead = random.Next(3, 11);
    var activities = new List<Activity>(activitiesPerLead);
    for (var k = 0; k < activitiesPerLead && i < maxObjects; k++)
    {
        i++;
        //For every lead we are creating several new activities in cycle
        var activity = new Activity
        {
            Subject = string.Format("Activity Subject #{0}", k + 1),
            Lead = lead,
            ActivityType = (ActivityType) GetRandomElement((IList)
                Utilities.GetEnumValues(typeof(ActivityType))),
            Priority = (ActivityPriority) GetRandomElement((IList)
                Utilities.GetEnumValues(typeof(ActivityPriority))),
            Status = (ActivityStatus) GetRandomElement((IList)
                Utilities.GetEnumValues(typeof(ActivityStatus))),
            Due = DateTime.Now.AddDays(random.Next(14)),
        };
        activities.Add(activity);
        //Adding Activity to Database
        Database.AddRecord(activity);
        CountActivities++;
    }
    lead.NextStep = GetRandomElement((IList<Activity>)activities);
    //Adding Lead to Database
    Database.AddRecord(lead);
    CountLeads++;
}

//Adding Contact to Database
Database.AddRecord(contact);
CountContacts++;
if (CountContacts % 1000 == 0)
{
    InvokeCommitting(EventArgs.Empty);
    Database.Storage.Commit();
    changedAfterCommit = false;
}
InvokeGeneratedContact(EventArgs.Empty);
if (IsComplete) break;
}
if (changedAfterCommit)
{
    InvokeCommitting(EventArgs.Empty);
    Database.Storage.Commit();
}
InvokeGenerationComplete(EventArgs.Empty);
}

private T GetRandomElement<T>(IList<T> list) where T : class
{
    if (list == null) throw new ArgumentNullException("list");

    return list.Count > 0 ? list[random.Next(list.Count)] : null;
}

private object GetRandomElement(IList list)
{

```

```

        if (list == null) throw new ArgumentNullException("list");
        return list.Count > 0 ? list[random.Next(list.Count)] : null;
    }

    private void InvokeClearComplete(EventArgs e)
    {
        var clearCompleteHandler = ClearComplete;
        if (clearCompleteHandler != null) clearCompleteHandler(this, e);
    }

    private void InvokeCommitting(EventArgs e)
    {
        var committingHandler = Committing;
        if (committingHandler != null) committingHandler(this, e);
    }

    private void InvokeGeneratedContact(EventArgs e)
    {
        var generatedContactHandler = GeneratedContact;
        if (generatedContactHandler != null) generatedContactHandler(this, e);
    }

    private void InvokeGenerationComplete(EventArgs e)
    {
        var generationCompleteHandler = GenerationComplete;
        if (generationCompleteHandler != null) generationCompleteHandler(this, e);
    }

    public event EventHandler ClearComplete;
    public event EventHandler GenerationComplete;
    public event EventHandler GeneratedContact;
    public event EventHandler Committing;
}

```