* 1. 

Hands-On Lab

Module 04: Sequences

Lab version: 1.0.0

Last updated: 12/6/2011

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS, Windows, MSDN, Visual Basic, Visual C++, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)Microsoft Biology Initiative Training by [Microsoft Corporation](http://research.microsoft.com/bio) is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/). Based on a work at [research.microsoft.com/bio](http://research.microsoft.com/bio).

[Module 04: Sequences 1](#_Toc306368278)

[Sequences in .NET Bio 4](#_Toc306368279)

[Objectives 4](#_Toc306368280)

[System Requirements 5](#_Toc306368281)

[Task 1: Introduction to Sequence Viewer 6](#_Toc306368282)

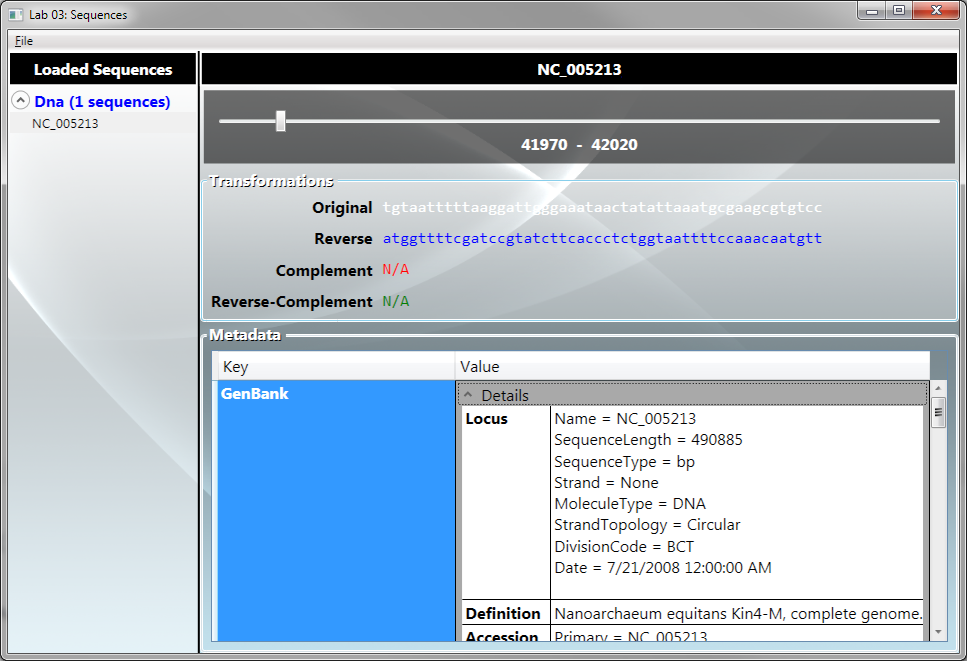
[Task 2: Loading and Displaying Sequences 12](#_Toc306368283)

[Task 3: Using Metadata 21](#_Toc306368284)

[Task 4: Saving Sequences 26](#_Toc306368285)

[Summary 30](#_Toc306368286)

Sequences in .NET Bio

* 1. This Hands-On Lab works through managing sequences and sequence items – using the properties and methods of the **ISequence** interface to view and manipulate the data. Through this exercise you will get more experience working with sequences and the parsers and formatters used to load and save them. You will also get some experience using the Windows Presentation Foundation (WPF) to create UI visualizations of the data. To save time, most of the actual UI will be pre-supplied and you will plug in the business logic to work with the data.
  2. The final application you will build looks like:
  3. 
  4. Objectives
  5. In this Hands-On Lab, you will get some experience building an application that uses .NET Bio
  + Work with a WPF application
  + Use .NET Bio parsers to read sequence files from disk.
  + Identify the sequence types.
  + Display and alter sequences as string text.
  + Use the basic sequence transformations to reverse, complement and reverse complement sequences.
  + Display and alter sequence metadata.
  + Save transcribed sequences back to disk in any supported format.

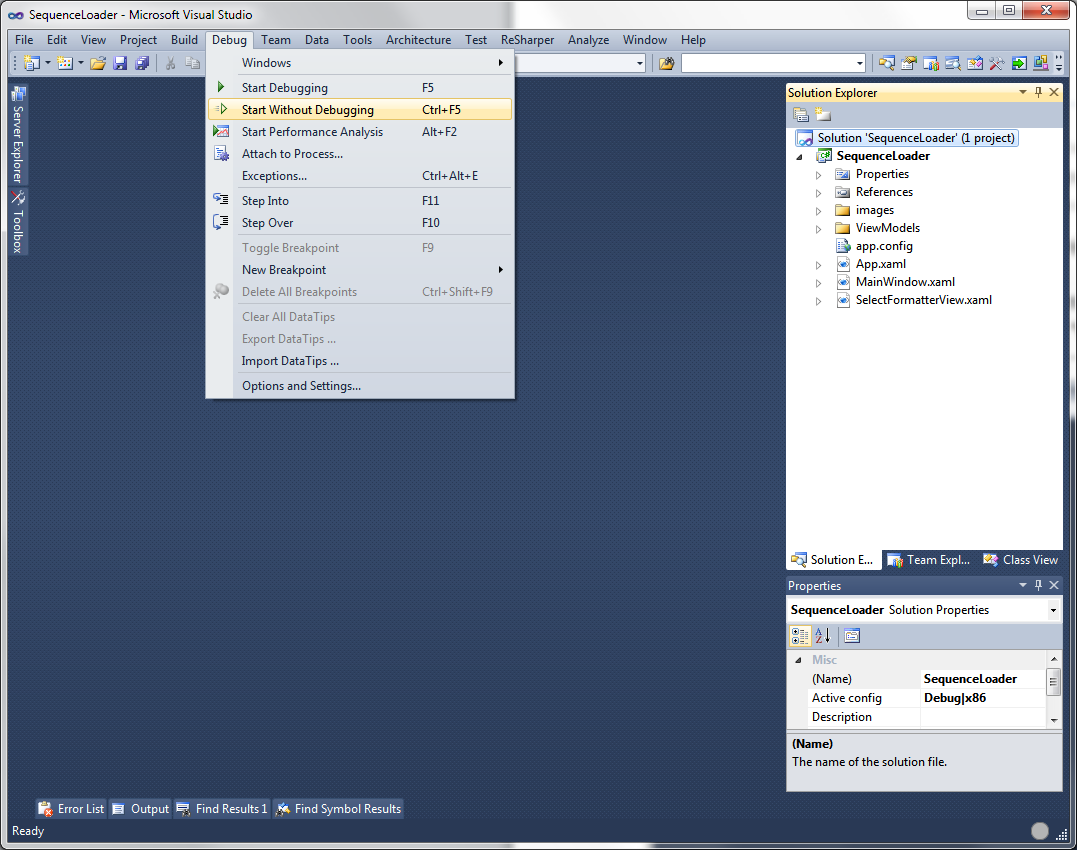
# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + .NET Bio 1.0 or later
  + Windows XP SP3 or better

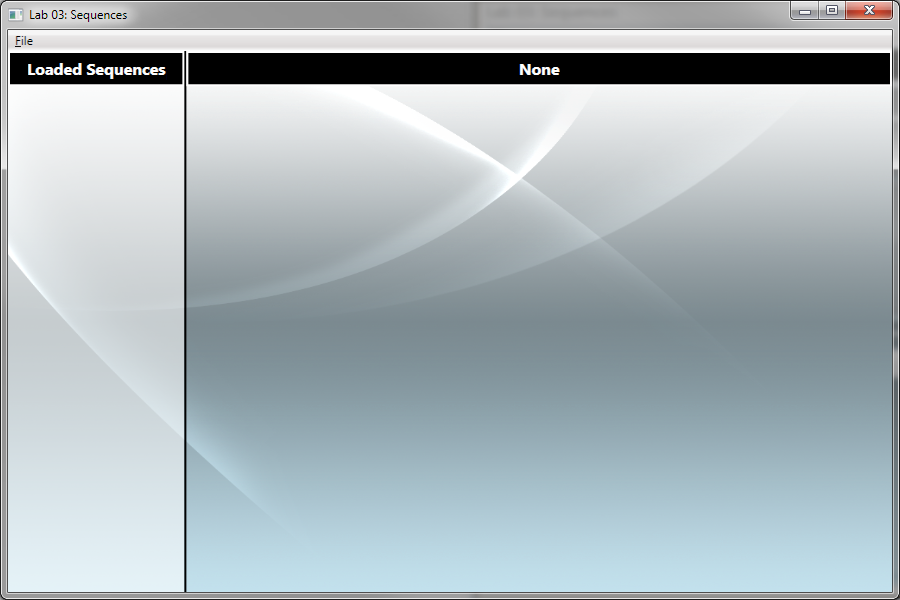
Task 1: Introduction to Sequence Viewer

* 1. In this first task you will open the start project that contains the pre-built user interface and be introduced to the components that make up the application. We will also add the requisite assemblies necessary to use .NET Bio and alter the project to support the full .NET framework.

1. To begin the exercise, open the starter project located at [task1/before/SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task1\before\SequenceLoader.sln). This should launch Visual Studio 2010.
   1. If it prompts you for developer settings, it does this the first time you open the application, make sure to choose “**General Development Settings**”.
2. Go ahead and run the project to see the UI – you can either press CTRL+F5, or use the “**Debug | Start Without Debugging**” menu option:



1. It should start the application with a blank UI:

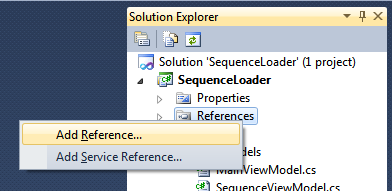


1. You can use “**File | Open**”, it will prompt for a filename, but do nothing with it. You can also use “**File | Exit**” to close the application. Go ahead and close the application.
2. Examine the **Solution Explorer**:

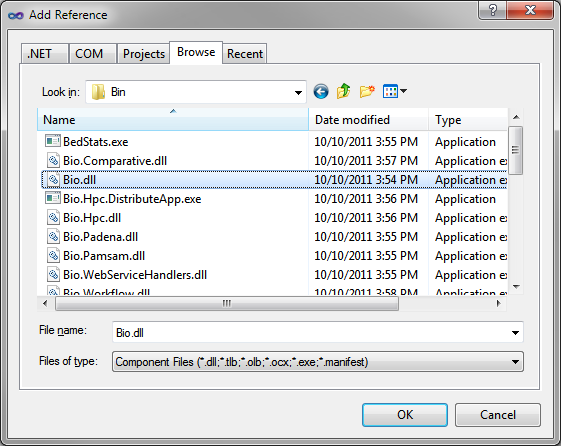
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  | | --- | --- | | File/Folder | Description | | Images | Contains image resources used in the user interface. | | ViewModels | Contains the View Model classes where business logic is placed. | | App.config | .NET application configuration file | | App.xaml | WPF Application object | | MainWindow.xaml | Main WPF application window | |

* 1. **What is a View Model?**
  2. This application uses a design pattern known as Model-View-ViewModel (or MVVM for short). It is a variation of the Model-View-Controller (MVC) and Presentation Model UI pattern intended to separate design elements (the UI) from the business logic driving them. The basic idea is the UI (referred to as the View) uses the WPF data binding infrastructure to bind to a view-specific object (referred to as the ViewModel) that provides access to the data (the Model). If you are unfamiliar with this pattern and want to learn more, you can get a great introduction from [WPF Apps with the Model-View-ViewModel design pattern](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx) article on MSDN.

1. Go ahead and explore the project:
   1. The **MainWindow.xaml** file is a XAML (pronounced Zaml, it stands for Extensible Markup Application Language) file that describes the UI with an XML based language. It has a corresponding code-behind file (click the “+” next to the XAML file) which will have some UI event handlers – similar to what you did in Lab 2 with Windows Forms.
   2. The **MainViewModel.cs** file is the primary view model – it contains the data the main window will manipulate, primarily the collection of sequences we want to display.
   3. The **SequenceViewModel.cs** file is a view model that will wrap a given sequence and provide data-bindable properties for the UI to display.
   4. The two view model files currently have very little code in them – the main shared element is an implementation of **INotifyPropertyChanged**.
   5. **What does INotifyPropertyChanged do?**
   6. The **INotifyPropertyChanged** interface is a core interface in WPF data binding. It allows WPF to determine that properties the UI is binding to have changed – thereby requiring the UI to update its state and display the new values. Anytime a property changes, the view model must raise the **PropertyChanged** event (the only member in the interface) to tell WPF about the change. The passed argument is a string containing the name of the changed property. Both of the view models in the project implement this interface and contain a helper method called **RaisePropertyChanged** that will do the heavy lifting of raising the event.
   7. This helper method has an additional line of coded executed in debug builds to verify the property name – since it’s passed as a text string there is opportunity for programmer error (due to misspellings, refactoring, etc.) The debug assertion in the code will verify the property exists at runtime and raise an assertion failure if not. If you get an **Assert Failed** box and it is coming from the **RaisePropertyChanged** method, it means you misspelled a property name! This is a best practice for implementing this support. It will ensure you find these errors during your development process. Without it, the call would silently fail and updates simply would not work!
2. As a final step before we begin adding logic, we will add support for .NET Bio. To do this, we need to add a reference to the core **.NET Bio** assembly.
   1. Right-click on the **References** folder in the **Solution Explorer**.
   2. Select “**Add Reference**”

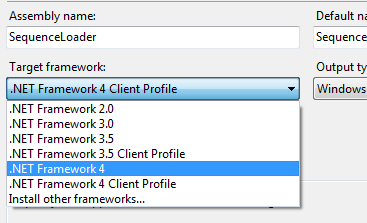


1. In the dialog, select the “**Browse**” tab.
2. The first step in using .NET Bio is to add a reference to the **Bio.dll** assembly.
   1. Right-click on the **References** folder and select “Add Reference”.
   2. In the dialog, select the “**Browse**” tab.
3. Change the directory to the “**.NET Bio**” folder. On a 32-bit machine this will normally be located in the “**C:\Program Files**” and on a 64-bit machine it will be in “**C:\Program Files (x86)**”. Select the version of .NET Bio you want to use by navigating to the next directory (as of this writing, the current version is “1.0”), and then finally select the **Bio.dll** file in the **Tools\Bin** directory. Below, the full path is **“C:\Program Files\.NET Bio\1.0\Tools\Bin”**



Recall that we still have one project setting we need to set in order to use .NET Bio. Remember that there are actually two versions of .NET 4.0 available – the full desktop edition, and the smaller client-profile edition. The Client Profile version is intended specifically for desktop-based Windows applications – it is a reduced feature set version that leaves out thing such as web server (ASP.NET) support, WCF service support, etc. By default, Visual Studio sets WPF applications to use this version and normally it would be fine. Unfortunately, at this time .NET Bio is not compiled to target this reduced set, so we need to convert the project to use the full desktop .NET version.

1. Double-click on the **Properties** folder in the **Solution Explorer** (or right-click on the solution and select **Properties**).
2. With the **Application** tab selected, locate the **Target framework** option – it should say “.NET Framework 4.0 Client Profile”. Pull this down and select “.NET Framework 4.0”. Visual Studio will prompt you to reload the project. Reload the project.



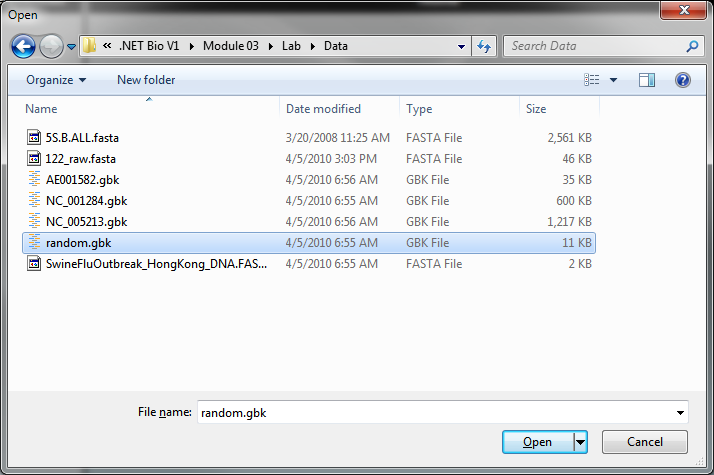
1. Once it is finished, compile the program (press CTRL+SHIFT+B, or use the menu option “**Build | Build Solution**”). It should build without any errors. You should be able to run the program as well.

If everything looks good then this task is complete. Move onto the next task to begin adding .NET Bio functionality!

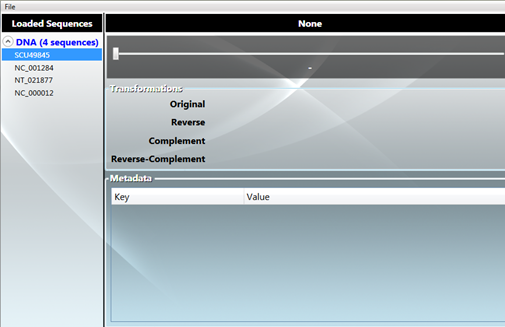
Task 2: Loading and Displaying Sequences

* 1. In this task you will add support to load sequences from files using sequence parsers, and display the loaded sequences – allowing a single sequence to be selected.

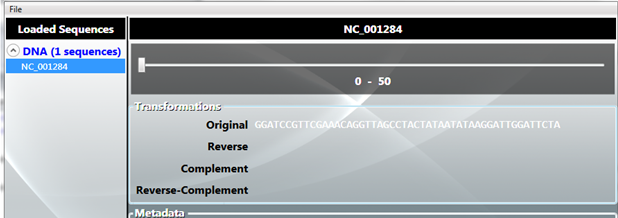
1. To begin, either continue from Task 1, or open the starter project located at [Task2\Before\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task2\before\SequenceLoader.sln).
2. The core bit of work we need to do is to implement the “File | Add Sequence”. Open the code behind file for **MainWindow.xaml** (click the “+” sign next to the XAML file, and double-click on the **MainWindow.xaml.cs** file).
3. Here you should find a method called **OnOpenFile**. This is the event handler for the menu option that adds sequences. It currently has code to invoke the **OpenFileDialog** (just like you did in Lab 2) to prompt the user for a file. Our goal is to use the **SequenceParser** class to load the selected file. See if you can implement the functionality to parse the file – returning a collection of sequences.
   1. You will need to add a reference to **Bio.IO.SequenceParsers**.
4. If you need help, here is an example:
   1. private void OnOpenFile(object sender, RoutedEventArgs e)
   2. {
   3. OpenFileDialog ofd = new OpenFileDialog();
   4. if (ofd.ShowDialog(this) == true)
   5. {
   6. var parser = SequenceParsers.FindParserByFileName(ofd.FileName);
   7. if (parser != null)
   8. {
   9. var sequenceList = parser.Parse();
   10. if (sequenceList != null)
   11. {
   12. }
   13. }
   14. }
   15. }
5. Our next step is to wrap each **ISequence** in a **SequenceViewModel**. Remember that the view model’s job is to wrap a model object (**ISequence** in this case) and provide WPF-friendly properties the UI can bind to.
   1. Open the **SequenceViewModel.cs** file (it is in the ViewModels folder).
   2. We want to wrap an **ISequence** object – create a private field to hold it, and add a constructor that takes an **ISequence** and stores it into the created field.
   3. public class SequenceViewModel : INotifyPropertyChanged
   4. {
   5. private readonly ISequence \_sequence;
   6. public SequenceViewModel(ISequence sequence)
   7. {
   8. \_sequence = sequence;
   9. }
   10. ...
6. Next, back in the **OnOpenFile** method, walk through the collection of sequences and wrap each one in a **SequenceViewModel** and add it to the **MainViewModel** **LoadedSequences** collection. You can get the **MainViewModel** instance using the “**\_vm**” field.
   1. var sequenceList = parser.Parse();
   2. if (sequenceList != null)
   3. {
   4. foreach (var sequence in sequenceList)
   5. {
   6. \_vm.LoadedSequences.Add(new SequenceViewModel(sequence));
   7. }
   8. }
7. Next, switch back to the **SequenceViewModel.cs** file – we want to add a property to the class called **ShortName**. This property is used by the UI to display the name in our tree. It should return the first 20 characters of the **ID** (the textual name of the sequence).
   1. Hint: remember to check to see if it *has* 20 characters!
8. Add a second property to expose the alphabet name – name it “MoleculeType”.
9. Your code should look something like this:
   1. public SequenceViewModel(ISequence sequence)
   2. {
   3. \_sequence = sequence;
   4. }
   5. public string ShortName
   6. {
   7. get { return \_sequence.ID.Substring(0,
   8. Math.Min(20, \_sequence.ID.Length)); }
   9. }
   10. public string MoleculeType
   11. {
   12. get { return \_sequence.Alphabet.Name; }
   13. }
10. Compile the application and run it.
11. Try loading a file – there are several sample data files located in the **Lab\Data** directory – pick one and see the results.



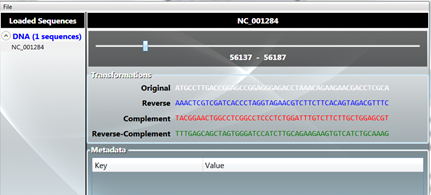
1. It should load the sequences and display them – grouped by the **MoleculeType** property:



1. Selecting a sequence does not do anything yet because we have not added the required properties to our view model – let’s fix that. The right side of the UI binds to several properties:
   1. The header across the top wants the full display ID in a property called **FullName**.
   2. The slider controls the section of the sequence we want to work with. It needs a **SegmentPosition** which is an integer index indicating where to start and **SegmentMaxRange** to determine its maximum scrollable value.
   3. The position is displayed in a TextBlock and requires the **SegmentPosition** and **SegmentEnd** which indicates the index of the last displayed item.
   4. The sequence data (based on the selected range of **SegmentPosition** and **SegmentEnd**) is expected in a property called **Segment**.
2. Switch to the **SequenceViewModel.cs** file. We will be adding a series of properties – mostly wrapping the properties of the internal sequence, or calculating some values based on our internal state. All of the properties must be **public** and have “getters”.
3. First, add a property to expose the full name of the sequence – remember this is the **ID** property. Name the property **FulName**:
   1. public string FullName
   2. {
   3. get { return \_sequence.ID; }
   4. }
4. Next, we want to display up to 50 sequence items at a time – add an integer constant to the class called **DisplayCount** so we can adjust it if necessary.
   1. public class SequenceViewModel : INotifyPropertyChanged
   2. {
   3. private const int DisplayCount = 50;
5. Next, add a new property called **SegmentPosition** – it should be an integer and be backed by a private field. It should have a public “getter” and “setter”.
   1. private int \_position;
   2. public int SegmentPosition
   3. {
   4. get { return \_position; }
   5. set
   6. {
   7. \_position = value;
   8. }
   9. }
6. When the property changes, we need to tell WPF – add a call to **RaisePropertyChanged** at the end of the “setter”:
   1. set
   2. {
   3. \_position = value;
   4. RaisePropertyChanged("SegmentPosition");
   5. }
7. Next, add a new property called **SegmentEnd** – it should be an integer property and return the current position added to **DisplayCount**. It will not need a “setter” as it’s calculated dynamically:
   1. public int SegmentEnd
   2. {
   3. get { return \_position + DisplayCount; }
   4. }
8. The value for **SegmentEnd** changes when we change our position – so when that happens we need to tell WPF to re-read the property value. Again, we will do this by raising a property change notification.
9. Add a call to **RaisePropertyChanged** for the **SegmentEnd** when the position changes:
   1. public int SegmentPosition
   2. {
   3. get { return \_position; }
   4. set
   5. {
   6. \_position = value;
   7. RaisePropertyChanged("SegmentPosition");
   8. RaisePropertyChanged("SegmentEnd");
   9. }
   10. }
   11. public int SegmentEnd
   12. {
   13. get { return \_position + DisplayCount; }
   14. }
10. We need a property to control the maximum slider range value – it should be an integer property that returns the sequence’s Length minus the **DisplayCount**. Call the property **SegmentMaxRange**:
    1. public int SegmentMaxRange
    2. {
    3. get { return (int)\_sequence.Count - DisplayCount; }
    4. }
11. Finally, we want to expose a segment of the sequence, using the current position and **DisplayCount** to bound it. We can return it as a string. There are several ways we could accomplish this:
    1. We could use our **GetString** method we wrote in the last lab on the sequence, and then use **Substring** to return a section of the string. This works – can you think of reasons why this is not a great approach? Can you think of a better way to do it?
12. A better approach would be to use the **GetSubSequence** method of the sequence to pull the specific range we want to work with. That is how we will implement this.
13. Add a new static method to create a string from an **ISequence**. It should use the same technique we used in the previous lab. Since we are always passing in a sub-sequence we should not need to restrict the size, although we wil add it here just as a guarantee.
    1. private static string GetString(ISequence sequence)
    2. {
    3. return new string(
    4. sequence.Select(b => (char) b).Take(255).ToArray());
    5. }
14. Add a new property called **Segment** that returns the specific range as a string:
    1. public string Segment
    2. {
    3. get { return GetString(\_sequence.GetSubSequence(\_position,
    4. DisplayCount)); }
    5. }
15. Run the program and load a data file. Select a sequence from the data file. Does it work as expected?

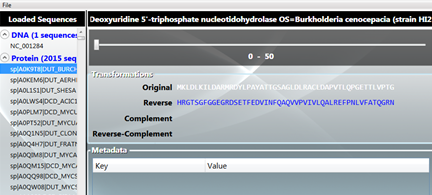


1. Try sliding the slider – what changes? Is there anything you expected to change but does not?
2. If you are closely watching – you should see the slider values changing, the position and sequence end changes, but the *sequence itself* is not changing. Can you think of why it is not changing? What do you need to do to fix it?
3. The problem is WPF does not realize the **Segment** property has been altered when the range changes, remember we need to tell WPF by raising a **PropertyChanged** notification. Add the **RaisePropertyChanged** call into the **SequencePosition** property:
4. Run the application again and verify it is working properly.
5. As a final step, expose the sequence transformations using the selected range with the following property names:
   1. **ReverseSegment** should call the sequence **GetReversedSequence** method.
   2. **ComplementSegment** should call the sequence **GetComplementedSequence** method.
   3. **ReverseComplementSegment** should call the sequence **GetReverseComplementedSequence** method.
6. In all three cases, the returned result should select the proper range and then return a string. Here is an example:
   1. public string ReverseSegment
   2. {
   3. get { return GetString(\_sequence.GetReversedSequence()
   4. .GetSubSequence(\_position, DisplayCount)); }
   5. }
7. Run the application again – did you forget to do anything in the implementation?



Is everything updating when you slide the slider?

1. Next, specifically load the **uniprot-dutpase.fasta** file from the data directory. This contains a whole bunch of protein sequences. It may take a second or two to parse and load. Once it is loaded, select a sequence from it.



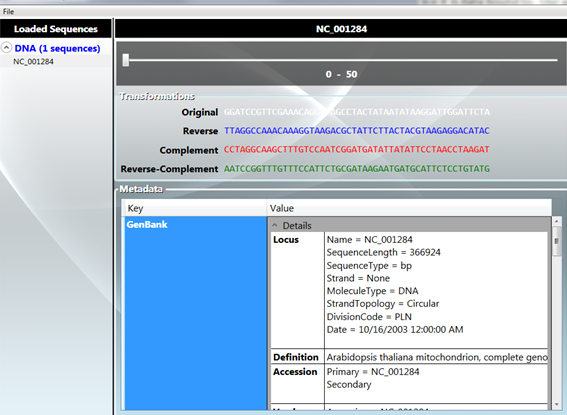
1. Notice that the **Complement** and **Reverse-Complement** are not being filled in? This is because they are proteins and those operations are not valid. What is more interesting however is: what is actually happening under the covers.
2. If you did not start the program in the debugger, stop it and debug it. You can hit F5 or use the **Debug** menu.
3. Open the **uniprot-dutpase.fasta** again and select a sequence. Now, open the *Output Window*(CTRL+ALT+O, or **View | Output**). There will be a whole bunch of binding errors. Most of the errors are ok because we have not implemented all the properties yet. There may be some exceptions at the top as well from loading the file (unknown symbols throw exceptions but are handled by the .NET Bio parsing process). The part we are interested in is towards the bottom – it is a binding error followed by an exception trace:
   1. A first chance exception of type 'System.InvalidOperationException' occurred in Bio.dll
   2. ...
4. What becomes obvious in the Output Window is that the **ReverseComplement** and **Complement** properties are both throwing a **System.InvalidOperationException** when executed on a protein. Now, normally when an unhandled exception occurs, the application is terminated. Here, the Binding is *catching and eating the exception*. This is nice in that the program does not terminate, but it can be misleading because we clearly have an issue we are not addressing. We could leave the code this way, but a better approach would be to add a test in the two properties and return some other text when the underlying sequence alphabet cannot support complements:
   1. public string ComplementSegment
   2. {
   3. get
   4. {
   5. if (!\_sequence.Alphabet.IsComplementSupported)
   6. return "N/A";
   7. return GetString(\_sequence.GetSubSequence(\_position, DisplayCount)
   8. .GetComplementedSequence());
   9. }
   10. }
5. Run the application again to test your changes.

This completes Task 2, in the next task we will add support to view and edit the metadata associated with the sequence. If you would like to examine a completed project, you can open the solution at [Task2\after\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task2\after\SequenceLoader.sln).

Task 3: Using Metadata

* 1. In this task you will add support to display and manipulate the metadata associated with the sequences. This starts with a new project – make sure to open the starter solution.

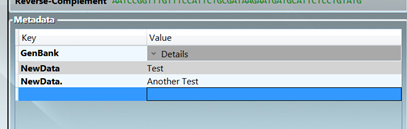
1. Open the starter project located at [Task3\Before\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task3\before\SequenceLoader.sln). This solution has a new source file called **MetadataViewModel.cs** which will wrap metadata for our WPF view.
2. The UI has a **DataGrid** which is expecting a **Metadata** property on the **SequenceViewModel** instance it is data bound to. Our goal is to provide this new property to expose the metadata to the grid so users can see the existing metadata, and add/remove new pieces of metadata.
3. To begin, open the **MetadataViewModel.cs** file in the **ViewModel** folder. It is not important to completely understand the source code here, just recognize that it wraps a **Dictionary** that maps strings to object – which is exactly what the **ISequence.Metadata** property exposes. We do not want to bind directly to that property since the UI will allow you to add and remove keys. That cannot be done through a property so what we need this wrapper to provide is the capability in code. Again, that is the purpose of a View Model, it wraps the model and provides view-specific functionality in the form of properties.
4. We will begin by adding the property expected in the UI.
   1. Open the **SequenceViewModel.cs** file.
   2. Add a new property to expose a collection of **MetadataViewModel** objects:
   3. public IList<MetadataViewModel> Metadata { get; private set; }
5. Notice we do not raise a **PropertyChange** notification on this. The actual property value will never change once we have assigned it – the *contents* of the collection might change, but not the property value itself. This is an important distinction.
6. Assign the property in the constructor. We will back it with a **List<T>** object and populate it with the existing metadata from the sequence. We can use LINQ (Language Integrated Query) to do this in one step, or if you prefer you can break it out across multiple steps and use a **foreach** statement for simplicity.
   * 1. **Note**: to use LINQ, you will need a **using System.Linq** statement at the top of the file.
   1. public SequenceViewModel(ISequence sequence)
   2. {
   3. \_sequence = sequence;
   4. Metadata = new List<MetadataViewModel>(
   5. \_sequence.Metadata
   6. .Select((kvp =>
   7. new MetadataViewModel(\_sequence.Metadata, kvp.Key))));
   8. }
7. Run the application and load a GenBank file – these typically have a **GenBankMetadata** object associated with them. Select the sequence and you should see metadata in the sequence:



1. You should be able to add and remove metadata from the grid as well. However, it will not correctly persist back to the metadata dictionary yet. We will implement that support next.
   1. **More about property change notification**
   2. As you have seen WPF uses the **INotifyPropertyChanged** interface quite a bit in order to know that properties have changed in the data objects it is bound to. This same requirement exists for collections of data as well. Since we normally expose **IList<T>** elements for collections, WPF needs to know when you have added and removed items from these collections so it can change the UI – the **DataGrid** in this case.
   3. There is a separate interface **INotifyCollectionChanged** that is used for this purpose and .NET provides a nice implementation in the form of **ObservableCollection<T>.** This is a **List<T>** class that also supports the collection notification feature WPF needs. It exposes a single event called **CollectionChanged** that is raised when the collection is altered in some way such as when items are added, removed, moved or changed.
   4. The **DataGrid** itself is capable of adding and removing items. It will actually create a new **MetadataViewModel** object when you add a new item – the default constructor will get called so we can distinguish it from one that was generated from existing metadata.

Normally the **INotifyCollectionChanged** support is necessary because the collection is being altered from the code, and we want the UI elements to know about the changes and respond accordingly. In this case, we do not actually modify the metadata from the code behind. All changes are always done from the **DataGrid** so WPF already knows about these. However, the code behind wants to know about these changes *so it can propagate them back to the original dictionary****!*** Remember, we are supplying an independent collection of items that WPF is altering; it is not the original Metadata dictionary on our sequence. So, when WPF makes a change (by adding a new item to the **DataGrid**) we want to know about that change and add or remove an item from the **Sequence.Metadata** dictionary.

1. We will start by changing the List into an **ObservableCollection** in the **SequenceViewModel** constructor. To use **ObservableCollection**, you will need to add a **System.Collections.ObjectModel** namespace at the top of your file, or fully qualify the class name. You should still populate it from the supplied sequence metadata. It is just changing the class.
   1. public SequenceViewModel(ISequence sequence)
   2. {
   3. \_sequence = sequence;
   4. Metadata = new ObservableCollection<MetadataViewModel>(
   5. \_sequence.Metadata
   6. .Select((kvp =>
   7. new MetadataViewModel(\_sequence.Metadata, kvp.Key))));
   8. }
2. Next, we want to know when the collection changes, so let’s hook the **CollectionChanged** event.
   1. You can either cast the **Metadata** property, or store it into an intermediate field in the method to get to the **CollectionChanged** event. This is necessary because the property is an **IList<T>**, not an **ObservableCollection<T>**.
   2. Attach the event handler to a method called **OnMetadataChanged**.
      1. **Hint:** Remember the double-Tab trick to get Visual Studio to generate the event handler for you? It is also useful to add a **using System.Collections.Specialized** namespace to your source file to avoid typing a lot of namespaces.
   3. public SequenceViewModel(ISequence sequence)
   4. {
   5. \_sequence = sequence;
   6. var md = new ObservableCollection<MetadataViewModel>(\_sequence.Metadata
   7. .Select((kvp =>
   8. new MetadataViewModel(\_sequence.Metadata, kvp.Key))));
   9. Metadata = md;
   10. md.CollectionChanged += OnMetadataCollectionChanged;
   11. }
   12. void OnMetadataCollectionChanged(object sender, NotifyCollectionChangedEventArgs e)
   13. {
   14. throw new **NotImplementedException**();
   15. }
3. Next, we want to look at the *action* being reported by the event. It will determine what has been done to the collection.
   1. When you add an item, we want to add the associated key to the real dictionary. Specifically, we want to associate the **ISequence.Metadata** dictionary with the **MetadataViewModel** that has been created by WPF as a result of adding the item to the **DataGrid**. There is already a method on the **MetadataViewModel**, **SetParentDictionary**, that will do this work for us. However, we need to ensure we have a unique key to add. We will just look in the existing collection and append a “.” to the key if we already see a duplicate.
   2. When you remove an item, we want to remove the item from the **ISequence.Metadata** Dictionary.
   3. Here is the code fragment you need:
   4. void OnMetadataCollectionChanged(object sender, NotifyCollectionChangedEventArgs e)
   5. {
   6. if (e.Action == NotifyCollectionChangedAction.Add)
   7. {
   8. foreach (MetadataViewModel item in e.NewItems)
   9. {
   10. string key = item.Key;
   11. while (\_sequence.Metadata.ContainsKey(key))
   12. key += ".";
   13. item.Key = key;
   14. item.SetParentDictionary(\_sequence.Metadata);
   15. }
   16. }
   17. else if (e.Action == NotifyCollectionChangedAction.Remove)
   18. {
   19. foreach (MetadataViewModel item in e.OldItems)
   20. {
   21. string key = item.Key;
   22. if (\_sequence.Metadata.ContainsKey(key))
   23. \_sequence.Metadata.Remove(key);
   24. }
   25. }
   26. }
4. Go ahead and run the application and try adding metadata. Just press **ENTER** in the value field to get a new line in the Data Grid.
   1. You should see a new key called “**New Data**”. If you add another, it should be “New Data.” (with an added period) and each subsequent new key will have an additional period.
   2. You should be able to change the key or data.



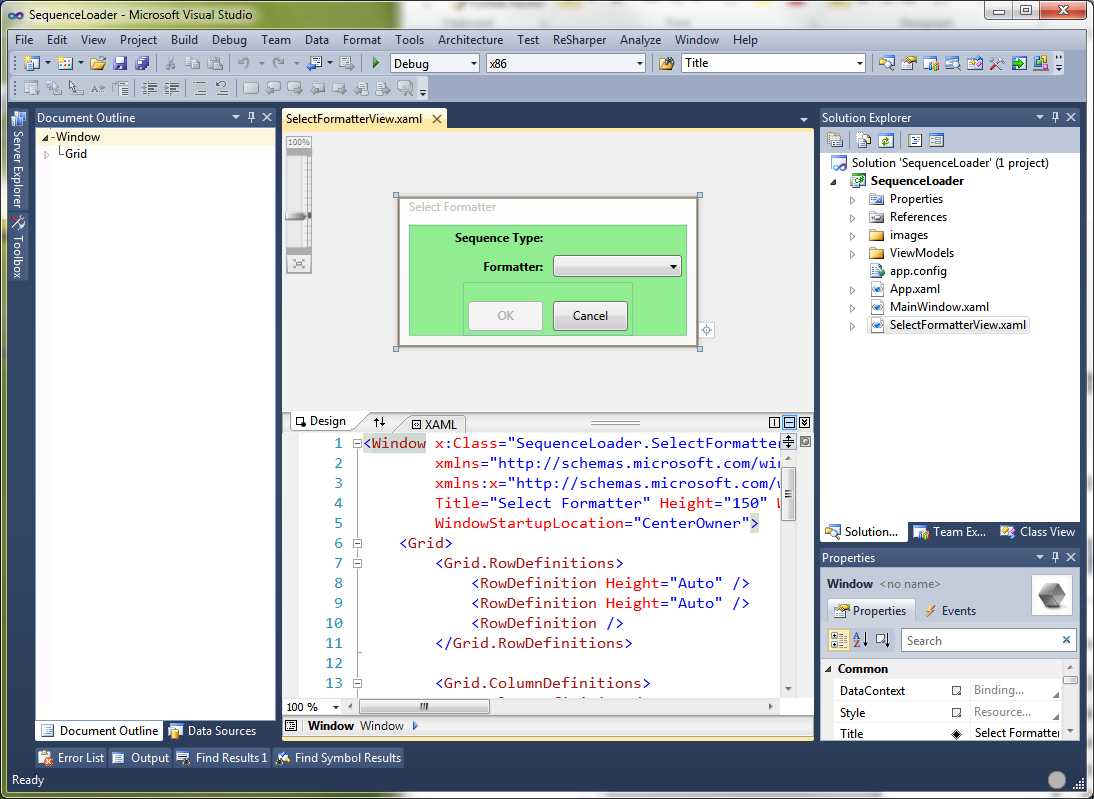
It is recommended that you step through this code in the debugger to get a full sense of what is happening since it is a bit complex with all the class interactions going on. The bottom line is it is just adding and removing a key from the dictionary owned by the sequence.

This completes Task 3; if you would like to examine a completed project, you can open the solution at [Task3\after\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task3\after\SequenceLoader.sln).

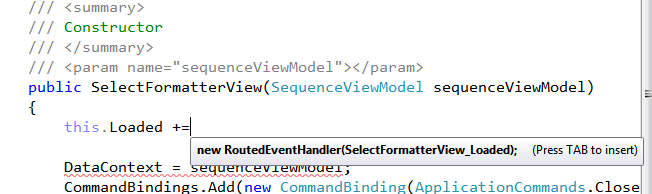
Task 4: Saving Sequences

* 1. In this final task you will add support to write a sequence back to disk using a formatter.

1. Open the starter project located at [Task4\Before\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task4\before\SequenceLoader.sln). This solution has a new source file called **SelectFormatterView.xaml** which will be used to prompt for the formatter to use. You will be editing this class to support the selection, but the UI itself has already been created for you.
   1. If you open the XAML file the designer will show you the basic UI:

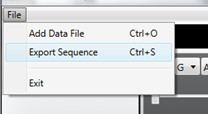


1. We want to execute the saving logic through the main menu – this already has an event wired up for you in the **MainWindow.xaml** code behind called **OnExportData**.
   1. Open **MainWindow.xaml.cs** (the code behind file) and locate the method. It’s currently empty.
   2. As a first step, we want to display the new formatter selection window.
      1. Create an instance of **SelectFormatterView**. The constructor requires a **SequenceViewModel.** We want to pass the *selected* sequence into the dialog.
      2. Call **ShowDialog** on the instance to display the window. If the method returns **true**, then it was dismissed with the **OK** button and we will want to prompt for the filename using the selected formatter – we’ll get to that in a moment.
   3. private void OnExportData(object sender, RoutedEventArgs e)
   4. {
   5. var selectFormatterView = new SelectFormatterView(\_vm.SelectedSequence);
   6. if (selectFormatterView.ShowDialog() == true)
   7. {
   8. }
   9. }
2. Now, let’s fill in some details in the select formatter view. Open the code behind for **SelectFormatterView.xaml**.
   1. First, we need to populate the ComboBox in the UI with the available formatters. We can retrieve the list using the **SequenceFormatters** type (recall it has a set of static properties).
   2. We want to add the formatters *after* the ComboBox has been created – this happens just before the **Loaded** event is raised on the window.
3. Wire up an event handler to the **Loaded** event in the constructor.
   1. Type “this.Loaded +=” and stop. Visual Studio will display a prompt indicating it can write some code for you:

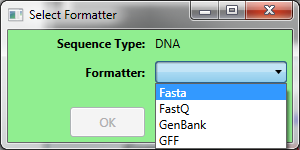


* 1. As indicated, hit TAB to insert the line, and TAB again to actually generate the method (so you will hit TAB twice). This will generate a method called **SelectFormatterView\_Loaded**.

1. In the new method, add each formatter to the ComboBox named “cbFormatters”.
   1. Add each formatter to the **cbFormatters.Items** collection.
      * 1. **Hint**: enumerate through **SequenceFormatters.All**
2. Try running the application – you should be able to load a sequence file, select a sequence and then the **Export** menu item will become available:



1. Selecting it should popup our new window – allowing you to select a formatter.



1. Once a formatter is selected you can dismiss the dialog , or cancel at any time.
2. Now, we need to capture the choice selected and allow the MainWindow to get at the selected item.
   1. Create a public property of type **ISequenceFormatter** – make the “getter” public, but the “setter” can be private. The lab will call the property **SelectedFormatter**.
   2. Set the property in the **OnOk** method to the **cbFormatters.SelectedItem**. This will be the selected item.
   3. public ISequenceFormatter SelectedFormatter { get; private set; }
   4. private void OnOk(object sender, RoutedEventArgs e)
   5. {
   6. SelectedFormatter = (ISequenceFormatter) cbFormatters.SelectedItem;
   7. DialogResult = true;
   8. }
3. Back in the **MainWindow.xaml.cs** file, if the dialog is dismissed with the **OK** button (it returns **true**), then get the selected formatter and use the following code to get a filename:
   1. private void OnExportData(object sender, RoutedEventArgs e)
   2. {
   3. var selectFormatterView = new SelectFormatterView(\_vm.SelectedSequence);
   4. if (selectFormatterView.ShowDialog() == true)
   5. {
   6. var sequenceFormatter = selectFormatterView.SelectedFormatter;
   7. SaveFileDialog sfd = new SaveFileDialog
   8. {
   9. Filter = sequenceFormatter.Name + "|" +
   10. sequenceFormatter.SupportedFileTypes.Replace(',', ';'),
   11. AddExtension = false,
   12. CreatePrompt = true
   13. };
   14. if (sfd.ShowDialog(this) == true)
   15. {
   16. }
   17. }
   18. }
4. This will display the **SaveFileDialog** and prompt for files of the type supported by the formatter. It will ensure the file exists.
5. Finally, use the selected filename and save the selected sequence using the selected formatter.
   1. You will need to expose the actual **ISequence** being held by the selected sequence view model. The lab will call the property **RawSequence**.
   2. Remember you need to open the formatter and close it when you are finished writing the sequence.
   3. Wrap it in a try/catch – just in case the formatter cannot save the file because of something such as an incorrect sequence type or bad data.
   4. If it fails, use **MessageBox.Show** to display the **Message** property of the exception. It takes a message and caption.
   5. Here is the code if you need help:
   6. try
   7. {
   8. sequenceFormatter.Open(sfd.FileName);
   9. sequenceFormatter.Write(\_vm.SelectedSequence.RawSequence);
   10. sequenceFormatter.Close();
   11. }
   12. catch (Exception ex)
   13. {
   14. MessageBox.Show(ex.Message, "Failed to export data", MessageBoxButton.OK);
   15. }
6. Go ahead and run the application. You should now be able to load a set of sequences and save them out in a different format.

There is one piece that will not completely work yet – editing metadata. If you load a GenBank file and either add new bits of metadata, or delete the existing metadata, the formatter will throw an error. It is very strict about the metadata allowed. We will add a new formatter and parser in the next lab that will support arbitrary metadata so we can try this out later!

You can examine a completed project at [task4\after\SequenceLoader.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp4_MBF.V2.zip\MBF.V2\Module%2003\Lab\Task4\after\SequenceLoader.sln).

Summary

In this lab you have worked with sequences, used the various properties of **ISequence** and played with metadata. There are many other things you could do if you’d like to spend more time in this area. One idea is to allow the user to select two sequences and combine them together with a **SegmentedSequence** and display them both as fragments and as a consolidated sequence in the UI.