* 1. 

Hands-On Lab

Module 06: Algorithms

Lab version: 1.0.0

Last updated: 10/14/2011

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 06: Algorithms 1](#_Toc293908624)

[Algorithms in .NET Bio 4](#_Toc293908625)

[Objectives 4](#_Toc293908626)

[System Requirements 4](#_Toc293908627)

[Task 1: Running Algorithms in .NET Bio 5](#_Toc293908628)

Algorithms in .NET Bio

* 1. This Hands-On Lab walks you through building an application to run algorithms against sequences loaded with .NET Bio. You will extend an existing WPF application to support sequence alignment, sequence assemblers, and transformations.
  2. Objectives
  3. In this Hands-On Lab, you will get some experience building an application that uses .NET Bio
  + Identifying sequence alignment and assembler algorithms.
  + Running sequence alignment algorithms against 2+ sequences.
  + Running sequence assembler algorithms against 2+ sequences.
  + Run transformations against selected sequences – to perform transcription and Protein Translation.

# 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

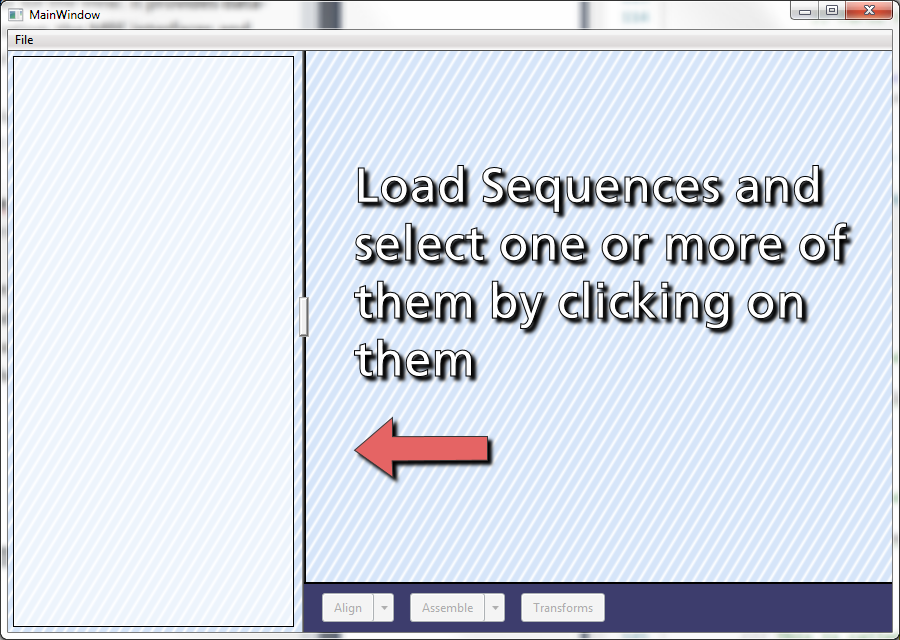
Task 1: Running Algorithms in .NET Bio

* 1. In this task you will examine an existing application and see how it is put together so you can add to and extend it. The first extension we will add is to support aligning two sequences together using the available alignment algorithms in the framework. Then we will continue by adding transform algorithms.

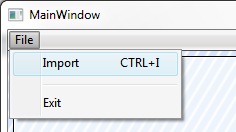
1. Open the starter project located at [Task1\before\Lab05.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2005\Lab\Task1\before\Lab05.sln).
2. Examine the project structure – it has the following folders and files:

|  |  |
| --- | --- |
|  | * **Infrastructure** – contains some helper classes and WPF converters that allow the application to separate behavior and UI with the Model-View-ViewModel (a variation of MVC) design pattern. * **Resources** – XAML based resources used to visualize various aspects of the UI. * **ViewModels –** logic that drives the UI – this is where you will make most of your changes. * **Views –** Controls used to display data in the UI. * **App.xaml** – the primary Application object. * **MainWindow.xaml** – main window UI, you make a few minor changes to this to add UI elements. |

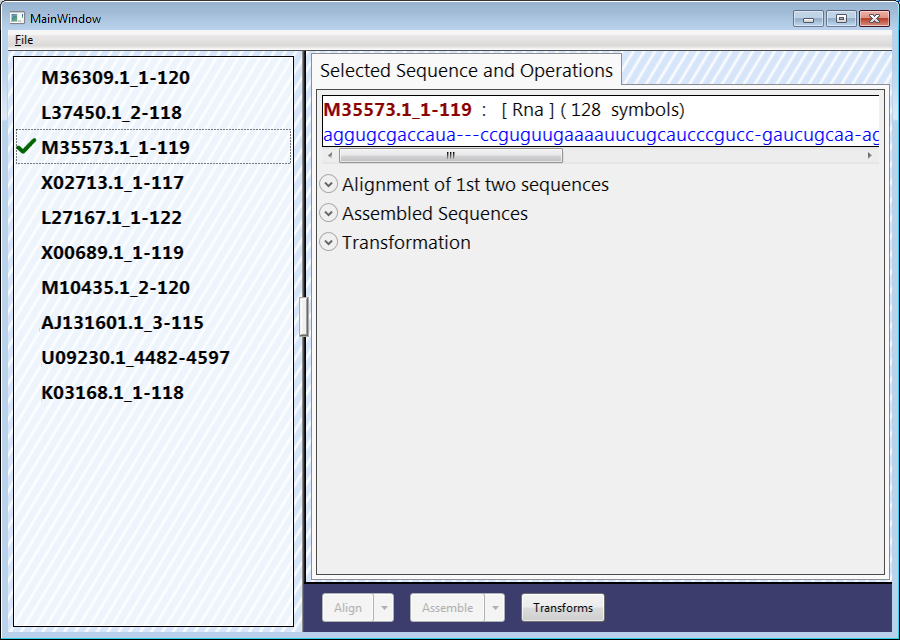
1. Expand the **ViewModels** folder – here you will find three view models. Remember from our previous WPF labs that a ViewModel is simply a model wrapper for the view: it provides data-bindable properties that WPF can then visualize. In this application, the .NET Bio interfaces and structures (such as **ISequence**) represent the underlying *model* and these classes wrap the data structures to make them easier to display in WPF.
   1. **BaseViewModel.cs** is the base class for all view models; it provides the **INotifyPropertyChanged** support required for any VM to communicate property changes with the View.
   2. **SequenceViewModel** is a VM that wraps an **ISequence** interface. It primarily provides an “**IsSelected**” state property to determine when the sequence is part of the “selected” set to run algorithms on.
   3. **MainViewModel** is the primary VM for the application; if you open it you will find it holds a collection of **SequenceViewModel** objects, and also manages a “selected” list which is just a LINQ expression that filters the full list using the **SequenceViewModel**.**IsSelected** property. This file also has some **ICommand** properties which the view executes when button presses occur in the UI. The **ICommand** properties, in turn, invoke methods in this class: **OnAlignSequences**, **OnAssembleSequences**, and **OnTransformSequences**. If you scroll to the bottom of the file you will find these methods are stubs which you will be implementing over the course of this lab.
   4. **Note:** if you want to explore the Model-View-ViewModel (MVVM) design pattern that is being used here more fully to understand the code, please read: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>
2. Build and Run the application – it will present a UI that looks like:



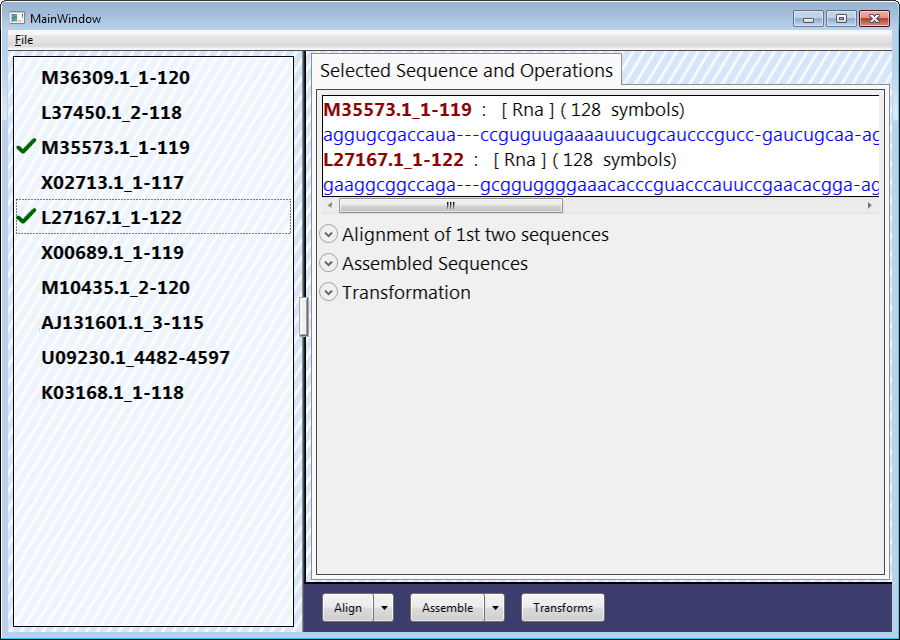
1. You can load sequences (from all the supported parser formats) using the **File | Import** menu option. If you are interested in the code that performs this, look at the



1. There are several sample files in the **Data** folder at the root of this lab and you can import as many files as you like – even of different molecule types and formats. Once you load a data file, you can click on the sequences loaded to select them. Selecting one or more will result in the sequence being displayed:



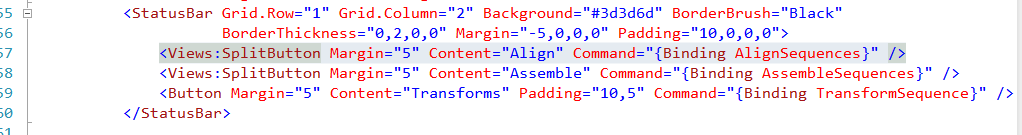
1. Notice that there is already a UI element present for alignment, assembled sequences and transformations; however, right now they do not function, nor will any data be displayed. Our job will be to implement this functionality!
2. We will start by implementing support for sequence alignment. First, load enough data so there is more than one sequence in your list, and then select at least two sequences:



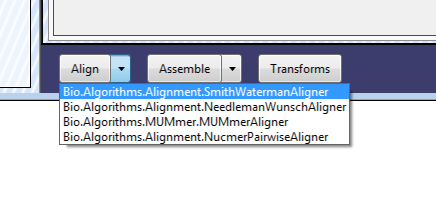
1. Notice that the buttons along the bottom are now enabled. These also have drop down arrows next to them which if you click on you will see that the list is empty. Our first step will be to populate the **Align** button with all the available alignment algorithms. Close the application.
2. Open the MainWindow.xaml file. It should open in the designer view. If does so, switch to the XAML view by clicking the **XAML** tab on the bottom of the designer window:
   1. Alternatively you can select the “split” button



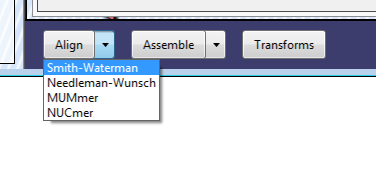
1. In the XAML view, find the alignment button. It should be around line 57 and look something like this:



1. The **SplitButton** control is a simple XAML-based **UserControl** that combines a **ComboBox** and **Button** (by overlapping them). It exposes properties from both of these elements to do its job. For this first step, we want to populate the **ComboBox** portion with all the available alignment algorithms. Recall that we can retrieve all fonts through the **SequenceAligners.All** static property. Since it is a property, we can data bind to it with WPF, and supply the data directly from it.
   1. We want to set the **ItemsSource** property. This must be a collection of data.
   2. We want to data bind to a static property that is part of .NET Bio. This will require we define the appropriate namespace in XAML at the top of the file so we can access the type in XAML.
2. Scroll to the top of the file and add the following namespace declaration:
   1. <Window x:Class="Lab05.MainWindow"
   2. xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   3. xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4. xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   5. xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   6. xmlns:Infrastructure="clr-namespace:Lab05.Infrastructure"
   7. xmlns:Views="clr-namespace:Lab05.Views"
   8. **xmlns:Alignment="clr-namespace:Bio.Algorithms.Alignment;assembly=Bio"**
3. This maps the **Bio.Algorithms.Alignment** namespace from the .NET Bio assembly to the XML “Alignment” namespace.
4. Go back to the **SplitButton** definition and set the **ItemsSource** property to **“{x:Static Alignment:SequenceAligners.All}”**. This will populate the **ComboBox** with each located aligner algorithm:
   1. <Views:SplitButton Margin="5" Content="Align" Command="{Binding AlignSequences}"
   2. ItemsSource="{x:Static Alignment:SequenceAligners.All}"/>
5. Run the application and select two sequences again. Pull down the alignment **ComboBox** to see the results:



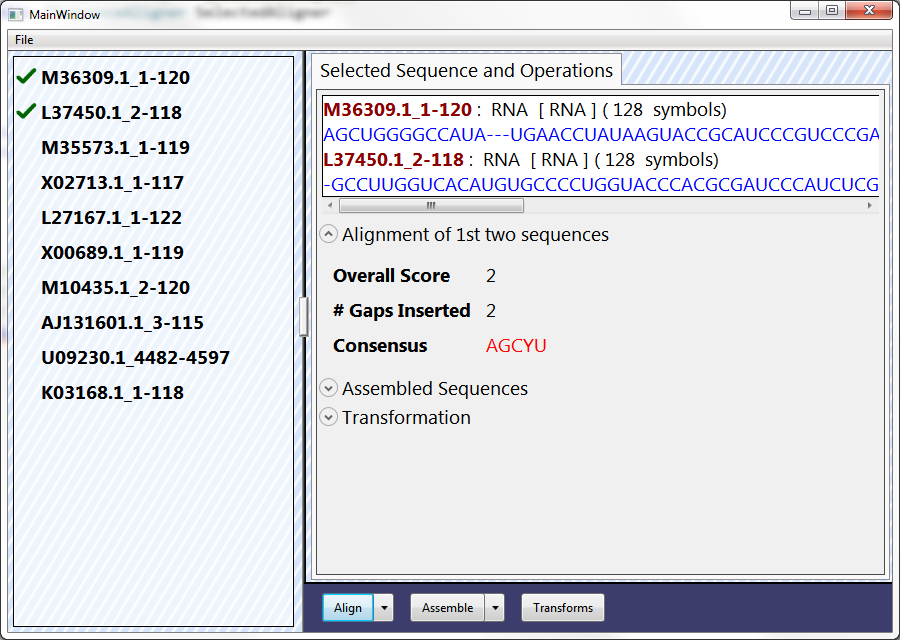
1. Notice we are getting the full type name of each alignment algorithm (**ISequenceAligner**) found. This works for this case but we can refine this by telling the ComboBox to use a specific property off the **ISequenceAligner** object it has a reference to. To do this, close the app and set the **DisplayMemberPath** property on the **SplitButton** to “Name” to use the **ISequenceAligner.Name** property in the display. Run it again and you should now just see the algorithm name in a nice displayable format:



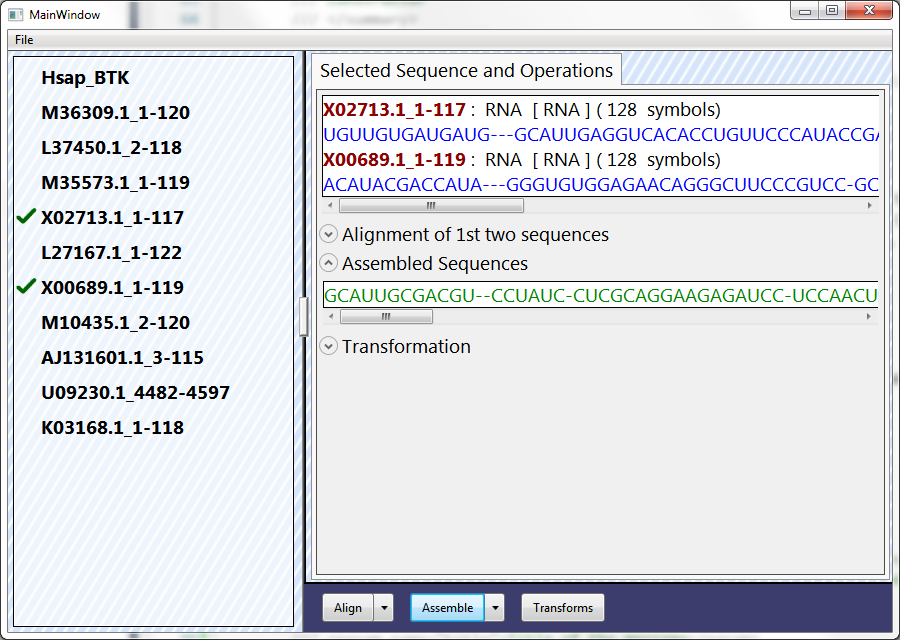
If you like, feel free to spruce it up by providing a **DataTemplate** that combines the **Name** and **Description** properties.

Next, we need to know which algorithm has been selected. To do this, we will rely on the **SelectedItem** property of the **ComboBox** (exposed by the SplitButton). We need to databind that property to a property of our MainViewModel. We will use **SelectedAligner**.

1. Add a **DataBinding** expression for the **SelectedItem** property of the align **SplitButton** that binds to a property called “**SelectedAligner**”. Since the MainViewModel is our default data binding context, you can just use the name in the binding statement, make sure to set the binding to “TwoWay” as shown below:
   1. <Views:SplitButton Margin="5" Content="Align" Command="{Binding AlignSequences}"
   2. ItemsSource="{x:Static Alignment:SequenceAligners.All}"
   3. DisplayMemberPath="Name"
   4. **SelectedItem="{Binding SelectedAligner, Mode=TwoWay}"** />
2. Now, switch to the MainViewModel.cs source file. Add a new public property named “SelectedAligner”.
   1. Since it is being data bound to the aligner list, the type should be **ISequenceAligner**.
   2. Back the property with a private field of the same type.
   3. call **OnPropertyChanged** (from our **BaseViewModel** class) when the property setter is called.
   4. You will need to add a **using** statement for **Bio.Algorithms.Alignment**.
   5. If you need a little helper, here is the property definition:
   6. private ISequenceAligner \_selectedAligner;
   7. public ISequenceAligner SelectedAligner
   8. {
   9. get { return \_selectedAligner; }
   10. set { \_selectedAligner = value; OnPropertyChanged("SelectedAligner"); }
   11. }
3. Find the constructor for the method and set the selected aligner property to one of the alignment algorithms. Use the **SequenceAligners** class to get an instance of an algorithm and use any one of them as your default.
   1. public MainViewModel()
   2. {
   3. LoadedSequences = new ObservableCollection<SequenceViewModel>();
   4. **SelectedAligner = SequenceAligners.SmithWaterman;**
   5. ImportFile = new DelegatingCommand(OnLoadFile);
   6. AlignSequences = new DelegatingCommand(OnAlignSequences, () => SelectedSequences.Count >= 2);
4. Now, locate the **OnAlignSequences** method in the file. It should be toward the bottom. This method is called when you click the “**Align**” button in the UI. Here we need to get the selected sequences, and use the selected aligner (defined above) to align them. The UI expects the resulting alignment to be in a public property named “Alignment”. That binding is already in place so let’s add a property to support it.
   1. Define the **Alignment** property. It should be of type **ISequenceAlignment**. Back it with a private field and make sure to call **OnPropertyChanged(“Alignment”)** when the property setter is called. This will ensure the UI knows the property has changed values.
5. In the **OnAlignSequences** method, use either **Align** or **AlignSimple** method – you can set the gap cost if you like, or leave the defaults in place.
   1. Assign the first (and likely only) **ISequenceAlignment** that is returned to your new property. Remember both of these methods actually return a list of alignments. You will need to test to see if any have returned and set the property appropriately.
   2. Most of the alignment algorithms only work with two alignments. You can either detect it (remember the interface for pairwise alignment?) or take the easy route and just pass in the first two selected sequences. This is what the example code will do.
   3. Make sure to wrap the code in a try/catch just in case the alignment fails. Display an error message if this happens by calling the **ShowError** method that has been defined for you. It takes three parameters: a title, message and the exception that has occurred.
   4. Finally, to ensure the alignment is not stale, set your **Alignment** property to **null** when a new set of sequences is selected – this occurs in the **OnSequenceSelectionChanged** method.
6. If you need some help, here is the basis of the code:
   1. private ISequenceAlignment \_alignment;
   2. public ISequenceAlignment Alignment
   3. {
   4. get { return \_alignment; }
   5. private set { \_alignment = value; OnPropertyChanged("Alignment"); }
   6. }
   7. /// <summary>
   8. /// This is called to align two sequences.
   9. /// </summary>
   10. private void OnAlignSequences()
   11. {
   12. var selectedSequences = SelectedSequences;
   13. Debug.Assert(selectedSequences.Count >= 2);
   14. Debug.Assert(SelectedAligner != null);
   15. try
   16. {
   17. var result = SelectedAligner.Align(selectedSequences.Take(2));
   18. Alignment = result.Count > 0 ? result[0] : null;
   19. }
   20. catch (Exception ex)
   21. {
   22. ShowError("Failed to align sequences", "Could not align sequences", ex);
   23. }
   24. }
7. Run the application, load some sequences, select at least two of them and click the **Align** button:



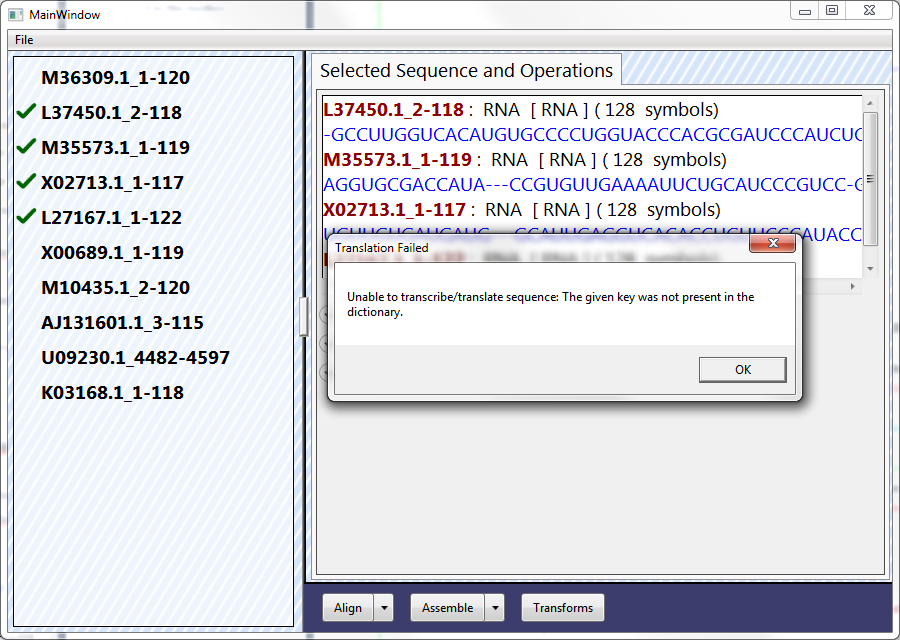
1. Try different algorithms by pulling the list down and selecting an alternative and then clicking the button. If the results do not change, you have likely missed the Two Way declaration on the binding (see step 18).
2. Once you have that working, let’s turn to the assembly algorithms. You will essentially follow the same steps as you did with alignments; but, now we will use the sequence assembler algorithms.
3. There is no **SequenceAssemblers** class, so just create **an IEnumerable<IDeNovoAssembler>** property which you can get each known assembler from. Here is an example:
   1. public IEnumerable<IDeNovoAssembler> AvailableAssemblers
   2. {
   3. get
   4. {
   5. yield return new OverlapDeNovoAssembler();
   6. }
   7. }
4. Use this as your **ItemsSource** for the **SplitButton**.
5. For the remainder, see if you can go back through steps [12-25] and apply the same basic logic to the Assemble **SplitButton** and **OnAssembleSequences** method. Here are a few hints:
   1. You will need to define a **SelectedAssembler** property of type **IDeNovoAssembler** – make sure the binding is TwoWay. It should be assigned to a default value (simple pairwise aligner) in the **MainViewModel** constructor.
   2. The UI expects the resulting **IDeNovoAssembly** to be in a public property named **AssembledSequence**.
   3. Make sure to set the **AssembledSequence** to null when different sequences are selected. (There is a method that is called when this happens – look in MainViewModel.cs for it.)
   4. You can pass more than two sequences into the assembler algorithm so there is no need to restrict the input.
   5. You will want to add a check if the **SelectedAssembler** is **null** before allowing the assembly function to run.
   6. If you need some help, refer to the completed solution ([Task1/After/Lab05.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2005\Lab\Task1\after\Lab05.sln)).
6. When you run the application, select multiple sequences, and then select an assembler and run it; you should see the resulting assembly (although it may not make much sense given that the data files are often different sequence strands).



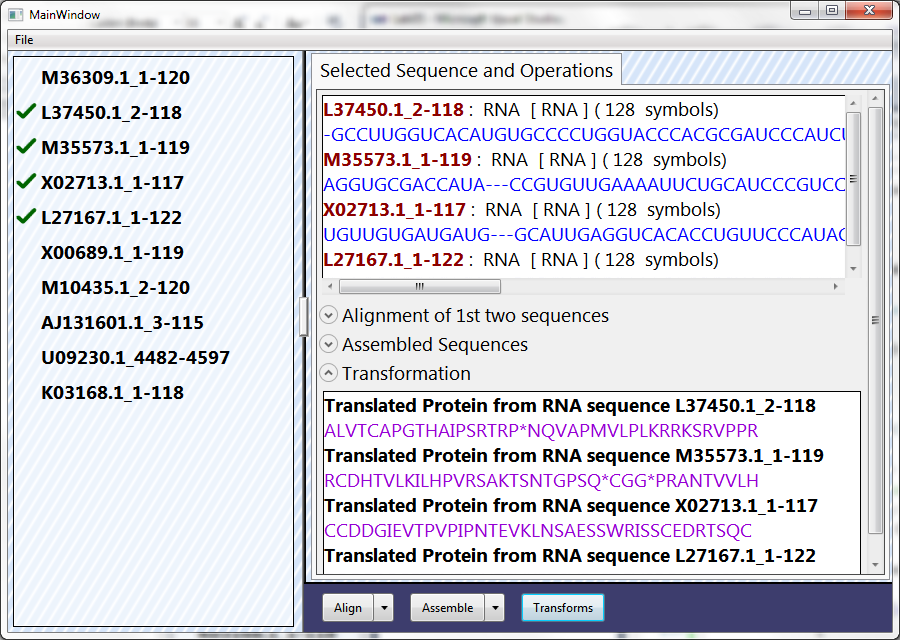
As a final step, we will implement transformations when the **Transforms** button is pressed. They will operate on all selected sequences:

* 1. If the sequence is DNA, transcribe it to RNA
  2. If the sequence is RNA, translate it to Protein.
  3. If the sequence is proteins, ignore it.

1. The UI has already been built for this. It expects an **IList<ISequence>** public property to be named **TranslatedSequences**. The backing storage should be an **ObservableCollection<ISequence>** to ensure it notifies WPF about changes made to the collection.
   1. Initialize the property in the **MainViewModel** constructor.
   2. **Clear** the collection when the selected sequences change.
2. In the **OnTransformSequences** method (this is called when you click the **Transforms** button), first clear the transform collection and then enumerate through the selected sequences using **foreach**. For each sequence
   1. Determine the type of sequence. An easy way to do this is to look at the alphabet used.
   2. Perform the appropriate transformation on the sequence (see above).
   3. Add the resulting sequence to your **TranslatedSequences** collection. Because it is backed by an **ObservableCollection**, WPF will be notified and UI should update appropriately.
   4. Make sure to wrap the work in try/catch and report any errors.
   5. Add a “Documentation” metadata property on the resulting sequence to a string indicating where it came from. For example, what type of translation you performed.
   6. If you need some help, here is the main function:
   7. private void OnTransformSequences()
   8. {
   9. TranslatedSequences.Clear();
   10. foreach (var sequence in SelectedSequences)
   11. {
   12. ISequence result = null;
   13. try
   14. {
   15. if (sequence.Alphabet == Alphabets.DNA)
   16. {
   17. result = Transcription.Transcribe(sequence);
   18. result.Metadata["Documentation"] =
   19. "Transcribed RNA from DNA sequence " +
   20. sequence.ID;
   21. }
   22. else if (sequence.Alphabet == Alphabets.RNA)
   23. {
   24. result = ProteinTranslation.Translate(sequence);
   25. result.Metadata["Documentation"] =
   26. "Translated Protein from RNA sequence " +
   27. sequence.ID;
   28. }
   29. }
   30. catch (Exception ex)
   31. {
   32. ShowError("Translation Failed",
   33. "Unable to transcribe/translate sequence", ex);
   34. }
   35. if (result != null)
   36. TranslatedSequences.Add(result);
   37. }
   38. }
3. Run the application and try transforming some sequences. If you select an RNA sequence with gaps and attempt to translate it to protein you will get an error indicating that an unknown symbol was encountered:



1. To fix this, try generating a new sequence from the RNA strand that removes all gaps and ambiguous symbols. The translation should work now:



The completed solution for this lab is available at [Task1/After/Lab05.sln](file:///C:\Users\v-dedewi\AppData\Local\Temp\Temp5_MBF.V2.zip\MBF.V2\Module%2005\Lab\Task1\after\Lab05.sln).