## Automated testing approach

### Make automated tests purposeful

Automated testing, like any other software development effort, needs to be properly architected and designed with the following factors in mind:

**Reliability:** A computer is being programmed to make PASS/FAIL decisions. These decisions must be accurate, repeatable and well documented.

**Effectiveness:** Automated tools should be easy to use and configure. Results logs need to be understandable.

**Maintainability**: Test code must be properly outlined and documented. General test case organization and execution should be planned and understandable.

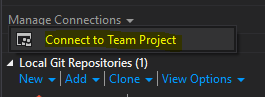
**Flexibility**: All test code design and implementation should be done with reusability and future enhancements in mind.

As with all project related work, the time spent creating automated testing must offset this effort by the return of value. This is demonstrated in various ways:

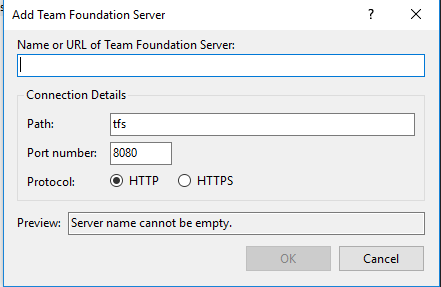
* Less time expended during the testing cycle than if manual testing is the only solution
* More thorough and consistent regression coverage
* Reduce human error
* Frees up people to test new functionality while computers test existing functionality

### How to Connect to the GurukuLA TEST Automation Project (Example)

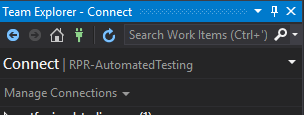
1. After launching the Visual Studio, a Team Foundation Server has to be added. In the Visual Studio IDE choose **View**🡪**Team Explorer**
2. In the Team Explorer Window click on **Manage Connections🡪Connect to Team Projects**



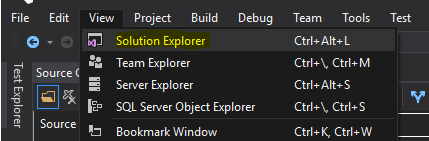
1. In the Connect to Team Foundation Server window click **Servers**
2. In the Add/Remove Team Foundation Server window click **Add**
3. In the Add Team Foundation Server window, in the Name or URL of Team Foundation Server enter “**https://gurukula.visualstudio.com**/” and click **OK.** Close the Add/Remove Team Foundation Server window



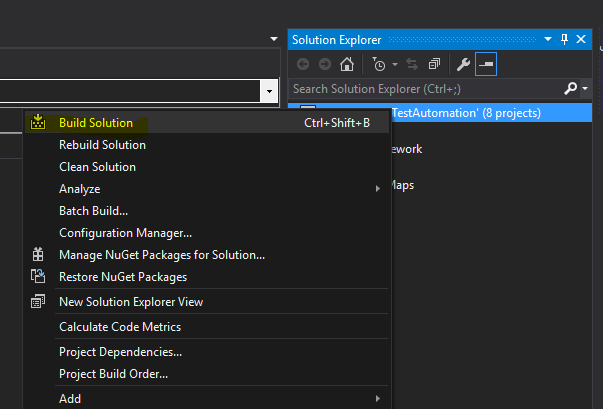
1. In the Connect to Team Foundation Server Window, select the “**Gurukula-AutomationTesting**” project and click **Connect**
2. In the Team Explorer window click **Home**



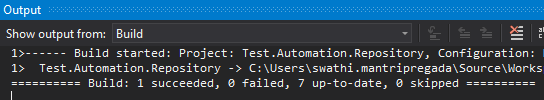
1. Click on the **Source Control Explorer**
2. From the Source Control Explorer get the latest version of the solution by right-clicking on the **Gurukula-AutomationTesting** folder and selecting **Get Latest Version**
3. To view the solution click View🡪Solution Explorer



1. Right click on the Solution🡪Build the Solution



1. The solution should build successfully



### Automated test framework architecture and design

The most practical way to automate tests is to create a well-architected and designed framework. The framework serves many purposes, all of which focus on bringing more value to the testing activity. Software engineers understand that the majority of software projects' success begins with proper architecture and design. Automated tests are code; therefore, these tests will benefit from upfront architecture and design.

#### Major components of an automated testing framework

The following list represents the major components of an automated test framework. Details of each component are provided in the sections below.

* Generic routines that do not interact with any application
* Generic, non-technology-specific manipulation routines
* Technology-specific manipulation routines
* Application-specific manipulation routines
* Tests



Figure 1: Automated test framework design layers

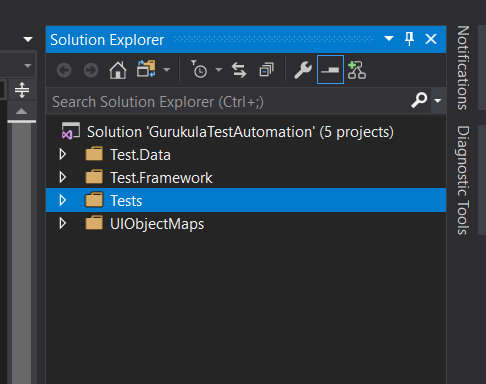


Figure 2: Gurukula Test Automation Different Layers

Each of these components is defined as a 'layer'. Each layer will be assigned its area of responsibility. It is up to the implementer of the framework to ensure that the layers are properly separated

The order of layers as listed is significant. The layers go from most generic to most specific to the Application Under Test (AUT). When deciding in which layer to add functionality, the goal is to add as much functionality as possible to the most generic layer in the framework. This will promote reusability and portability to as much framework code as possible. The goal is to create generic components that are easily shareable amongst teams and projects. As new code is written or existing code is modified, when placed in the correct layer, these changes are easily distributed and will be available to other layers (and users) without much effort and without requiring changes in other code that may already be using the routines that were changed.

When rewriting or significantly modifying existing code, do not change the signature of the existing functions or methods. Changing the number, order or types of the parameters will require immediate maintenance in other areas of the framework; otherwise, there will be compilation errors. If it is unavoidable to change a signature, then make all the necessary changes to the framework and tests prior to checking in the changes. The changes should not cause other areas of the framework to fail. Unless there is wide agreement already in place, do not make changes that will require that others get involved to adapt their code to the changes so that tests continue to work. Use optional parameters and overloading to reduce the need to change existing tests.

#### Generic routines that do not interact with any application

This layer is for any code that is not related to the UI or application. It is used to build more capability and utility that the test tool may not already have. These can include any functionality that a tester may need for any reason.

This layer can be reused for any application when using the same test tool. If written properly, these routines will work with test tools that use the same programming language. Optionally, these routines can be written in any programming language and then included as DLLs in the automated test solution.

Third party DLLs that meet the requirements of this layer can also be included and have their calls wrapped in the generic layers so that they can be called from the other layers.

#### Generic, technology-nonspecific manipulation routines

This layer is for technology-non-specific handling of any UI interaction with any application. Some of the user interface controls that are used in applications may not work consistently with the test tool. The framework code needs to be updated so that testers can successfully test that application. An example would be for grids. Grids generally have a dynamic number of rows. That, plus the inherent ability of the rows to be sorted, makes it very difficult to use the data that is in the rows. It is very typical to need to create routines to interact with grids. In this example, not all grids could be handled in this layer, as some technology-specific code may need to be written to handle specific implementations. However, in an object oriented programming language, a base class can be written in a layer, which is then derived in a more specific layer.

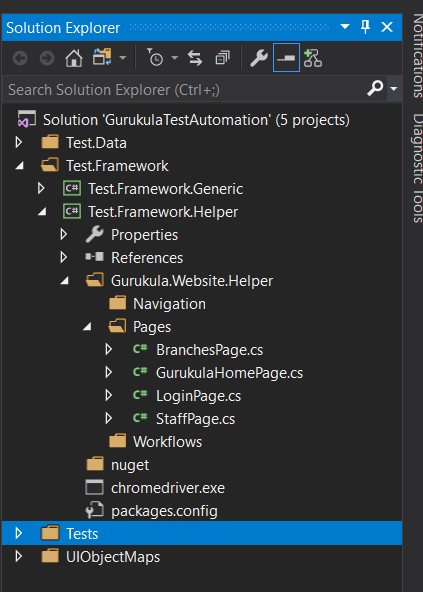
This is also the layer that additional logging is implemented. The logging will be written so that is has a very generic interface, and the actual logger that is written or selected will be used by the generic layer. If written properly, the logging interface can easily swap different logging solutions.

#### Technology-specific manipulation routines

This is the layer for handling specific technologies, for example, Browsers, WinForms and WPF. This layer will handle difficult interactions with the user interface and define custom interactions for “difficult” controls. In the case of browsers, it will hide differences between versions and brands. When there are third party custom controls that look and feel like typical controls, but would require the test tool to interact with them differently, this layer will be where the custom controls will be wrapped with calls that will seem to be identical to the similar controls from the testers’ perspectives. The intent is to make similar controls always have the same mode of interaction.

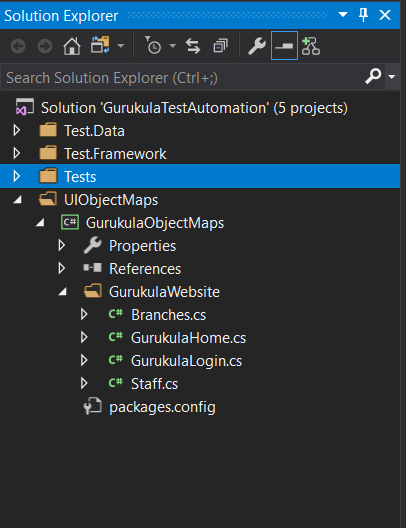
#### Application-specific manipulation routines

This layer is targeted to the specific AUT. It will be placed in its own project and reference the more generic layers. This layer will contain all routines necessary for the tests to interact with the AUT.



##### Repository for UI mappings

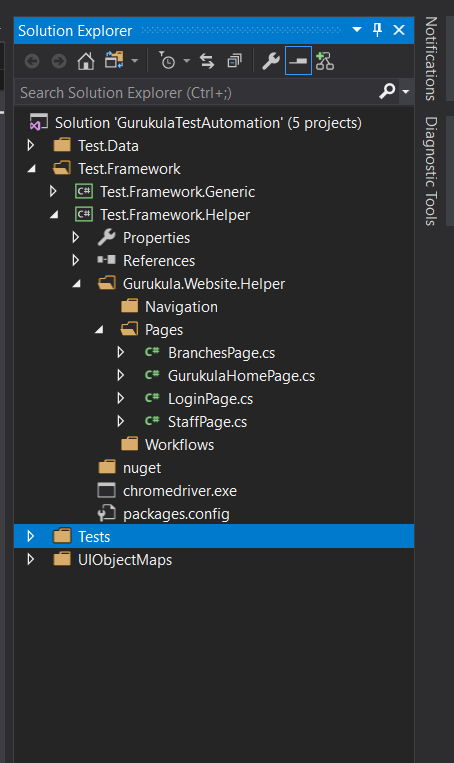
Each separate page or window will have its own dedicated file to hold UI object mapping. Each file needs to be given a name that indicates immediately what its contents are. That name is based upon the window or page that is mapped. The purpose of giving each window in the application its own mapping file is to make maintaining changes on the UI much easier in the testing framework. In addition, this will assist by eliminating unintentional duplication of windows and controls in the mapping files. The naming standard, which must be known to the team, will directly support the minimization of maintenance and reduce the learning curve. It is to be expected, therefore, for the test automation suite to have 200+ separate windows to test will have an equivalent number of unique UI mapping files. The undesired alternative would be to have fewer files with multiple UI mappings that would be difficult to find specific window definitions.



##### Defines handlers for every individual form/page

This handler will have a name that the mapping file. This consistency is important to identify the proper file to access whenever needed.

The handler/helper classes will support all test framework access to each window in the application. The handler/helper class will use a single UI mapping file (which in turn contains a single window mapping) that it will manipulate. This handler/helper class is generally the entry point for testing a particular window. It is generally discouraged for tests to interact with any window except by using the handler/helper class. These classes can have methods are basic to very complex routines. In the case where multiple windows will be used during a test, other classes will be used that combine the various handler/helper classes.



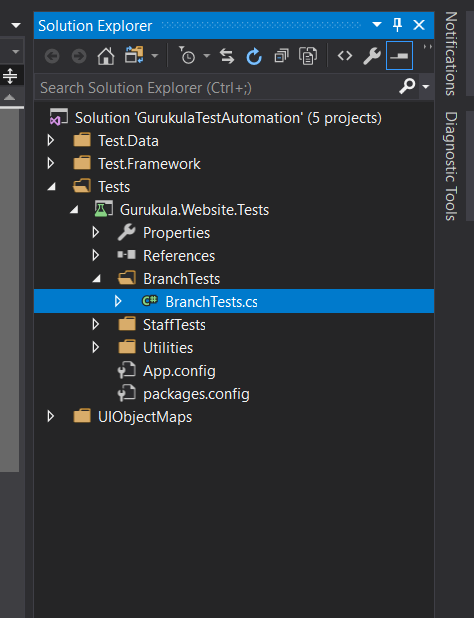
##### Implements the ability to interact with the Form/Page

The handlers can be extended to do more than basic operations. Consider a login screen that accepts a username, password and then a Login button needs to be clicked. In this instance, the tester does not write a line of code for each interaction with the login window. It is much more effective to create a single routine that accepts the username and password that then completes the entire login process.

#### Tests

This layer is specifically intended to define test cases. Test cases call the other layers; these should not be very complex, as all the complexity is handled in the other layers. No cross calling of test routines is allowed.

Ideally, each of these layers will be in their own project folders in the automated test solution. This is to encourage portability and reusability. A layer can only call into a more generic layer. This design requires that no layer ever call into a more specific layer. A generic layer calling into a less-generic layer will quickly break the overall architecture and stability of the framework. A layer can make calls into itself, but call mapping needs to be done to ensure that there are zero circular references created when this is done.



#### Automated Testing Coding standards

In general, the automated tests and supporting code will adhere to the same coding standards that are in place for the overall program. Review the overall architecture guidelines to learn the proper coding standards. The QA Architect will manage the test automation code review process. It is the responsibility of team members to adhere to the standards and to hold others to the same standards.

##### Additional guidelines for automated testing

* Naming of controls and windows - UI mappings
  + - Controls
      * Do not include the class or type of control in the name of the control. This will cause additional maintenance if the control type changes. i.e. **Address** rather than AddressEdit
      * No duplicate control names within any given window's collection of objects. This includes where duplicates are normally considered valid, such as in different control hierarchies.
      * Consistent naming of controls that are represented on several windows - including different applications. FirstName should always be FirstName, not sometimes First, Name or Firstname. This included those times where the labeling or naming of these controls in the AUTs are inconsistent.
    - Pages
      * Determine an adequate identifier that is unique to every application that will be tested. This identifier should be descriptive, but as short as possible. This is identifier is used as a prefix for all windows in the AUT. i.e. Admin for an Administration application or AIA for an automobile insurance application.
      * Prefix window names with this unique identifier. i.e. **Admin**Login
        + This will make it possible to write tests that will manipulate multiple applications while making it very simple to understand what the tests are doing.

* Naming of Classes, Methods and Properties
  + - Pages/Windows: For each WPFForm, WinForm or HTML page in an application, there will be a corresponding class created that is designed to manage and interact with the WPFForm, WinForm or HTML page. These classes will have names that mimic the name given the UI mappings.
      * Append to the class an established and consistent value like 'Handler' or 'Helper'.
      * Example, for an administrative application's login page, create a class that will interact with that page named 'AdminLoginHelper' or 'AdminLoginHandler'. Use the chosen value ('Helper' or 'Handler') exclusively for all enterprise-wide names
    - Methods and properties
      * Use a verb-noun pattern for naming methods, i.e. SaveFile, CreateNewAccount
      * Use a noun(-verb) naming pattern for properties, i.e. 'Filename', 'AccountToSave'
* Execute anywhere
  + - All tests need to be written so that they can be executed on any designated test machine with minimal, if any, setup. Any setup that is required will apply to other tests as well. What this means is that no test should be written that is dependent upon any special system configuration unless the test itself is able to determine the proper configuration and then make the appropriate changes. Such special configurations need to be minimized or else this will reduce the ability to batch execute tests unattended.
    - Tests need to be written so that they are capable of executing unattended. This allows tests to be executed overnight and as part of automated builds. Requiring human interaction to successfully complete tests must be avoided. In the cases where this is not practical for valid reasons, then these tests need to be segregated from the other tests, batched and run together as their own suite that all team members know will need to have human interaction. Any interaction should be purposeful, meaning that the author of the test has explicitly coded into the test this need for human interaction. It is not valid to use human interaction to merely help a test along that should be capable of running on its own.
* No hardcoding
  + - Environment
    - Other
* Minimize calling any part of the actual testing tool's syntax from the test
  + - Use the Handler/Helper classes that are defined for each unique window in the AUT. Calling a test tool’s methods generally mean that very specific information about the AUT needs to be used in the method call. If the AUT-specific information changes, then so must the calls in the tests. The automation framework is implemented specifically to shield tests from application changes. Calling test tools methods from tests erode the automation framework’s ability to do its job.
    - Avoiding direct calls to the test tool methods will reduce overall maintenance of tests. Conceptually, this could results in a set of tests that can be reused if a new test tool is brought in-house. The old tests would call the new automation framework that would be implemented in the new test tool. Ideally, the tests would not need to change.
* Overuse logging during execution
  + - Create the ability to log just about every activity that the test is doing. This will greatly assist with test debugging and give an audit trail when a bug needs to be reported against the AUT.
    - Use a debug flag to control the amount of logging that is desired based upon the situation.
* Design for test distribution and concurrent execution of tests
  + - A given test should not assume that it is the sole test executing against a given environment, unless that is an explicit requirement of an entire suite of tests
    - A given test should not assume that it will be executed only a single iteration. This is an issue with tests that are tasked with adding values that are unique to a system, or are modifying existing data. If that test needs to be re-executed, it will require that the data for the AUT is refreshed so that the test can be re-executed. Even if this is defined as a base state for a given test, any tests that are running concurrently against the AUT will likely fail. In these cases, it is important to use data in a way that adding or modifying it does impact the outcomes of other tests. Generating unique data as part of the test (unique to that specific test run) is a valid strategy. This can be done using timestamps or GUIDs. In the case of sequential data, read the highest sequential value from the data repository during test runtime, and increment that value.
    - All tests need to be absolutely independent from any other test. This independence allows for focusing testing on particular areas or to re-execute tests to verify the cause of errors. This ultimately means that if a particular test ever requires that another test execute previously, then there is a dependency issue that must be resolved. Computers do not get tired re-executing a similar set of actions redundantly, so there is no reason to eliminate actions that are repeated from test to test. In fact, this repetition is encouraged, since it can potentially prove that a particular path in the AUT is stable. Repeated actions should actually be defined as methods so that all new tests are using the identical procedure to navigate through the AUT functionality.
* Other specific test designs
  + - Always consider that any given test will accomplish two main goals, set up the system to a state to be tested, and verify the expected state against the actual state
    - A test will test just a single test. This will be defined by the name of the test case. This single test can be the state of all the values on a screen, or even a database. However, the single test is defined by testing a specific state. If a test verifies a particular state, then does more actions against the AUT and then tests that new state, those are two different tests. In this example, the solution is to create two separate tests that will verify each state independently.
    - Give test cases a name that will allow for tracing back to either the manual test case or user story/requirement
    - No test ever calls another test
    - Name or categorize tests in a way that will simplify certain types of testing. For example, rather than create tests that specific to smoke tests, create useful tests that you can batch into a smoke test suite, but will also be useful in regression testing.
    - Use care when writing tests. False passes or failures make automated tests useless. In fact, incorrect results will reduce overall confidence in automated testing. Tests (and all test code) needs to attempt to be better than the application code that is getting tested.
    - Specific target-of-test: Automated tests can verify results against any part of the AUT. Normally, if a test is executed against the UI, then verify at that UI level. It is valid, however, to verify at the database. There can be multiple similar tests that verify at each tire of the AUT if that is deemed a value add to the testing process.
    - Recovery from AUT failure. From time to time, the AUT will lock or crash. It is the responsibility of each test at initial execution to detect the state of the AUT, do any recovery actions, and get the AUT into the proper state to continue test execution. If a test cannot reset the AUT into a recovered state, then the test will log this and flag the test as a Fail.