EITI

FXU

Framework for eXecutable UML

Łukasz Zaremba under the supervision of Anna Derezińska

2011-09-01

This document describes a configuration of the FXU Runtime Library and a process of creating and execution of UML sematic mutants using the FXU framework.

A TABLE OF CONFIGURATION PARAMETERS OF THE FXU RUNTIME LIBRARY

This section presents a **FXU runtime configuration parameters table**.

No	Interface name	A description of an interface to communicate with an object implementing …	Implementing class name	A description of an implementation and its constructor parameters
1	`IAfterEvent`	A UML time event.	`AfterEvent`	A basic implementation of the UML time event.
2	`IAfterEvent Controller`	An after event controller. The controller handles time events in a UML state.	`AfterEventSingle ThreadController`	The controller handles all time events in a UML state using one thread.
			`AfterEventMultip leThreadsControl ler`	The controller handles time events in a UML state using a separated thread for each time event.
3	`IAfterEvent Supplier`	An after event supplier. The supplier provides time events for a particular UML state.	`StaticAfterEvent Supplier`	The supplier provides time events based on a time event set, that is created durring initialization of the state machine.
			`DynamicAfterEven tSupplier`	The supplier provides time events based on transitions outgoing the state dynamically durring executioin.
4	`ICallEvent`	A UML call event.	`CallEvent`	A basic implementation of the UML call event.
5	`IChangeEven t`	A UML change event.	`ChangeEvent`	A basic implementation of the UML change event.
6	`IChoice`	A UML choice pseudostate.	`Choice`	A basic implementation of the UML choice pseudostate.
7	`IClock`	A MARTE profile clock.	`LogicalClock`	A implementation of a MARTE logical clock.
			`ChronometricCloc k`	A implementation of a MARTE chronometric clock. Parameters: `unit` – a string of characters, that defined a unit for procesing in the clock. Correct values: `milisecond`, `second`, `minute`, `hour`, `day`, `week`.
8	`ICompletion Event`	A UML completion event.	`CompletionEvent`	A basic implementation of the UML completion event.
9	`IDebugWrite r`	A functionality of logging trace info of state machines execution. Must be defined one (unnamed `registry` element) for all machines.	`DebugWriter`	It log trace info of state machines execution. Parameters: `isDebug` – a parameter that indicates whether the logging is enabled. `debugLogFileFullPath` – a parameter that defines full path of the file with logs.
10	`IDeepHistor y`	A UML deep history pseudostate.	`DeepHistory`	A basic implementation of the UML deep history pseudostate.
11	`IDefaultEnt ryRule`	A functinality of choosing the first vertex in a region to enter after the region was entered.	`ExitRegionIfNoIn itialPseudostate`	An implementation that performs a default entry to the initial pseudostate in a region, if only one initial preudostate exists in the region. Otherwise, the region is considered as completed already after entry.
			`RequiredExactlyO neInitialPseudos tate`	An implementation that performs a default entry to the initial pseudostate in a region, if only one initial preudostate exists in the region. Otherwise, the state machine is considered ill-formed and the execution is stopped.
			`UseMostAppropria teState`	An implementation that performs a default entry to the initial pseudostate in a region, if only one initial preudostate exists in the region. Otherwise, it performs an entry to the state in the region such that it is not a target of any transition. If such state doesn't exist, the region is considered as completed already after entry.
12	`IEntryPoint`	A UML entry point pseudostate.	`EntryPoint`	A basic implementation of the UML entry point pseudostate.
13	`IEventBroad caster`	An event broadcaster to provide event communication for UML state machines.	`EventBroadcaster`	Genrates an event in a pool of every state machine. Parameters: `stateMachineFilter` – a reference to an object that implements a functinality of selecting state machines, to which a particular event is sent.
			`EventMulticaster`	Generates an event in a pool of state machines given in parameter. Parameters: `stateMachineFilter` – a reference to an object that implements a functinality of selecting state machines, to which a particular event is sent.
14	`IEventPool`	An event pool for UML state machine.	`EventQueue`	A FIFO implementation of the event pool for UML state machine.

			PriorityEventQueue	An implementation of the event pool that supports custom priorities for different event types. Parameters: callEventPriority – a parameter that indicates priority for events of call type, changeEventPriority – of change type, signalPriority – of signal type, afterEventPriority – of time type, completionEventPriority – of completion type.
15	IExecutionScheduler	An execution scheduler. The execution scheduler performs execution in a UML state machine.	ParallelExecutionScheduler	An implementation of the execution scheduler that perform the execution in orthogonal regions in separated .NET framework threads.
			CustomPrioritiesExecutionScheduler	An implementation of the execution scheduler that perform the execution in orthogonal regions in separated .NET framework threads, but this implementation takes into account prioriorites set for regions. The execution in regions, the entry to regions and the exit form regions are performed according these priorities.
16	IExitPoint	A UML exit point pseudostate.	ExitPoint	A basic implementation of the UML exit point pseudostate
17	IFinalState	A UML final state.	FinalState	A basic implementation of the UML fork pseudostate.
18	IFork	A UML fork pseudostate.	Fork	A basic implementation of the UML fork pseudostate.
19	IInitialPseudostate	A UML initial pseudostate.	InitialPseudostate	A basic implementation of the UML initial pseudostate.
20	IInternalTransition	A UML internal transition.	InternalTransition	A basic implementation of the UML internal transition.
21	IJoin	A UML join pseudostate.	Join	A basic implementation of the UML join pseudostate.
22	IJoinLikeExitPoint	A UML exit point pseudostate that behaves like a join pseudostate.	JoinLikeExitPoint	A basic implementation of the UML exit point pseudostate that behaves like a join pseudostate.
23	IJunction	A UML junction pseudostate.	Junction	A basic implementation of the UML junction pseudostate.
24	IJunctionLikeExitPoint	A UML exit point pseudostate that behaves like a junction pseudostate.	JunctionLikeExitPoint	A basic implementation of the UML exit point pseudostate that behaves like a junction pseudostate.
25	ILocalTransition	A UML local transition.	LocalTransition	A basic implementation of the UML local transition
26	IRegion	A UML region.	Region	An implementation of the UML region. Execution in the region has default priority.
			Region	An implementation of the UML region, that allow to define execution of action priorities. Parameters: defaultEntryRule – a reference to an object that implements a functinality of choosing the first vertex in a region to enter after the region was entered. entryPriority – a parameter that defines the priority of entry to the region. executionPriority – a parameter that defines the priority of execution in the region. exitPriority – a parameter that defines the priority of exiting from the region.
27	IShallowHistory	A UML shallow history pseudostate.	ShallowHistory	A basic implementation of the UML shallow history pseudostate.
28	ISignalEvent	A UML signal event.	SignalEvent	A basic implementation of the UML signal event.
29	IState	A UML state.	State	A basic implementation of the UML state.
30	IStateMachine	A UML state machine.	StateMachine	A basic implementation of the UML state machine. Parameters: eventsPool – a reference to an object that implements an event pool. afterEventsController – a reference to an object that implements an after event controller, which handles time events in the state machine. afterEventsSupplier – a reference to an object that implements an after event supplier, which provides time events for a the state machine. executionScheduler – a reference to an object that implements an execution scheduler, which performs execution in the state machine.
			TimedProcessingStateMachine	A MARTE profile implementation of the state machine. Parameters: duration – a parameter that defines a value of duration for the state machine. clk – a reference to an object that implements an clock

				for the state machine. `evStart` – a reference to the start event for the state machine. `evFinished` – a reference to the finished event for the state machine `eventsPool` – a reference to an object that implements an event pool. `afterEventsController` – a reference to an object that implements an after event controller, which handles time events in the state machine. `afterEventsSupplier` – a reference to an object that implements an after event supplier, which provides time events for a the state machine. `executionScheduler` – a reference to an object that implements an execution scheduler, which performs execution in the state machine. `generateStartEv` – a boolean value that indicates does the state machine generate an event on it is started. `generateFinishEv` – a boolean value that indicates does the state machine generate an event on it is finished. `runOnStartEv` – a boolean value that indicates does the state machine execute its internal action when a particular event occurs. `exitOnFinishEv` – a boolean value that indicates does the state machine stop the execution of its internal action when a particular event occurs.
31	`IStateMachinesFilter`	A filter selects state machines, to which a particular event is sent. The filter is used by an event broadcaster.	`AllStateMachinesFilter`	An implementation, that does not filter state machines durring an event dispatching. It always provide a set of all state machines in an executed model.
			`FilteredStateMachinesFilter`	An implementation, that filters state machines durring an event dispatching. It provide a set of state machines such that the event is subscribed by each one.
32	`ITerminate`	A UML terminate pseudostate.	`Terminate`	A basic implementation of the UML terminate pseudostate.
33	`ITransition`	A UML transition.	`Transition`	A basic implementation of the UML transition.

INSTRUCTION TO FXU MUTATION TEST ADD-IN

This section describes how to install **the FXU Mutation Test Add-in into Visual Studio 2010** and how to use it.

OVERVIEW:

The FXU Mutation Test Add-in performs mutation testing inside Visual Studio 2010 IDE.

SYSTEM REQUIREMENTS:

Supported Operating Systems:

- Windows 7,
- Windows Server 2003 R2,
- Windows Server 2003 SP2,
- Windows Server 2008 R2,
- Windows Server 2008 SP2,
- Windows Vista with SP2,
- Windows XP with SP3.

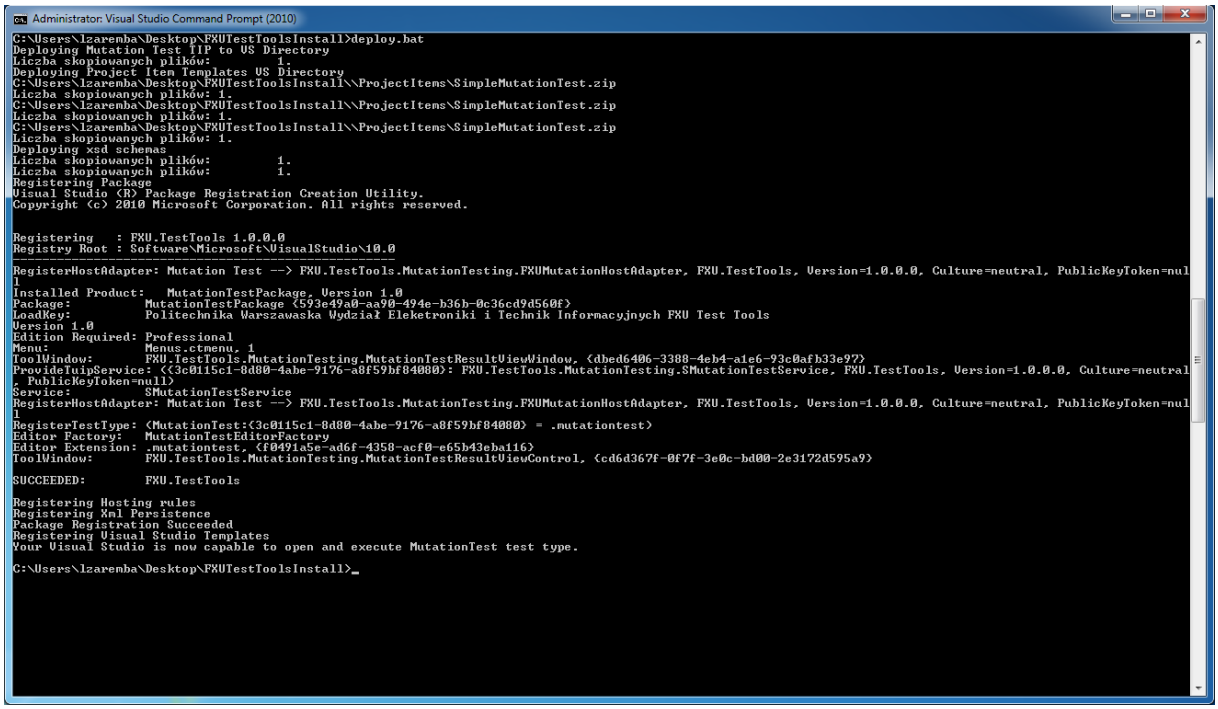
Prerequisites:

- Visual Studio 2010 Premium or better (for information please check http://www.microsoft.com/visualstudio/2010/sysreqs).
- Visual Studio 2010 SDK (for information please check http://www.microsoft.com/download/en/details.aspx?id=2680).

INSTALATION INSTRUCTIONS:

1. Download the **FXUTestToolsInstall.zip** file from FXU website (http://galera.ii.pw.edu.pl/~adr/FXU/download/FXUTestToolsInstall.zip).
2. Extract contents of the file to any location.
3. Make sure that Visual Studio 2010 is not running.
4. Run Visual Studio Command Prompt (2010) with administrator privileges.
5. In the command prompt change a current location to the location where you extracted content of **FXUTestToolsInstall.zip**. Then change a current location, to the **Deploy** folder.

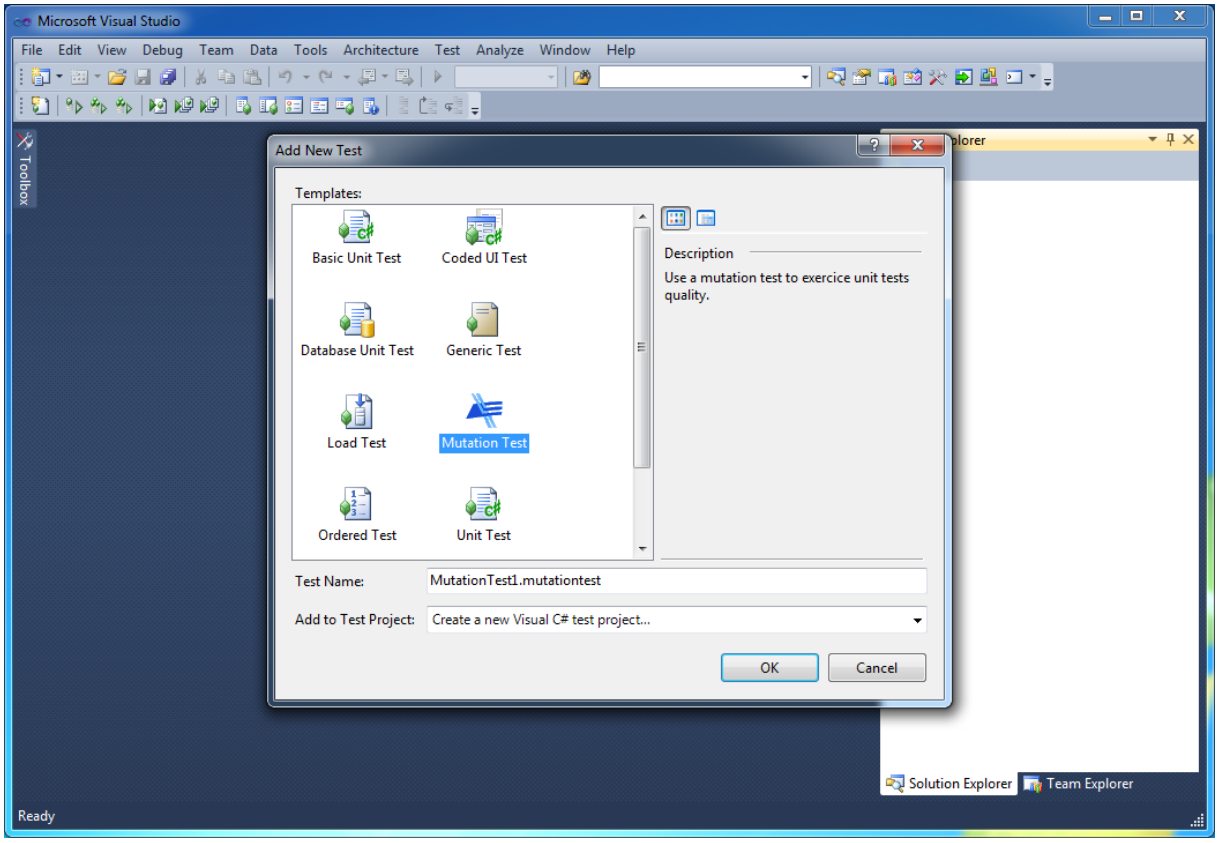
6. To begin installation run script `deploy.bat`.



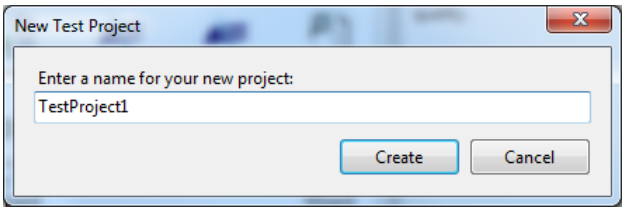
7. The installation may take about a few minutes. After all you see a picture as above. It means that the FXU Mutation Test Add-in was installed successfully.

USAGE INSTRUCTION:

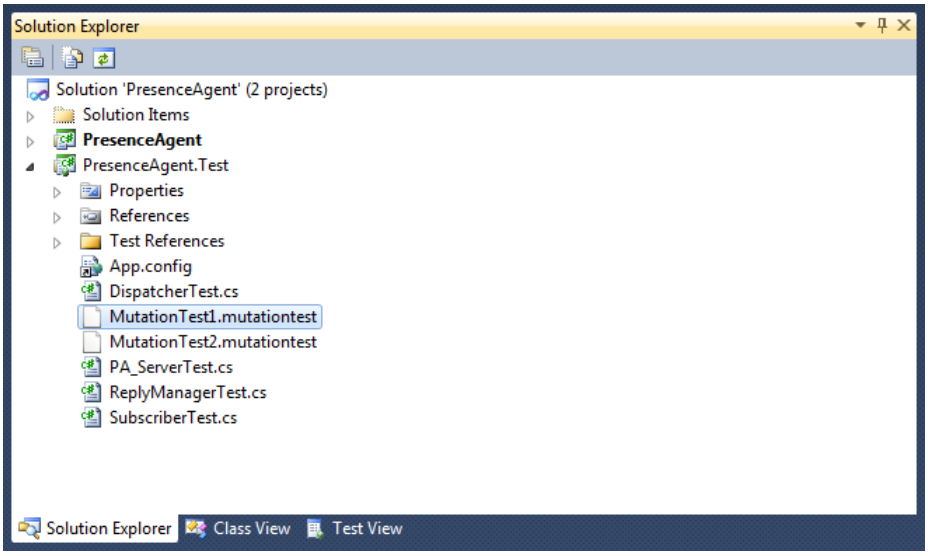
1. Open Visual Studio 2010.
2. On the **Test** menu, click **New Test…**. The **Add New Test** dialog box appears.



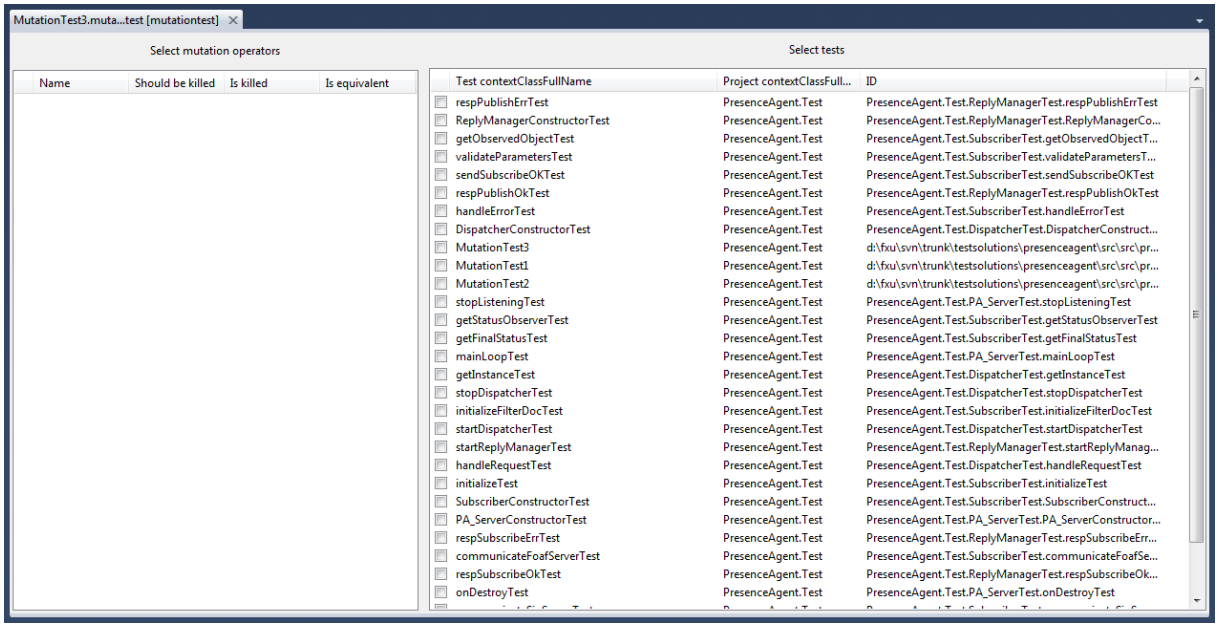
3. In the list of **Templates**, select **Mutation Test**.
4. In the **Test Name** box, type a name ended with the `.mutationtest` extension.
5. For **Add to Test Project**, select an existing test project or select **Create a new Visual C# test project…** and then click **OK**.
6. If **Create a new Visual C# test project…**option was selected, the **New Test Project** dialog box appears. In the **Enter a name for your new project** box, type a name of the test project and then click **Create**. This creates a project, which is displayed in **Solution Explorer**.



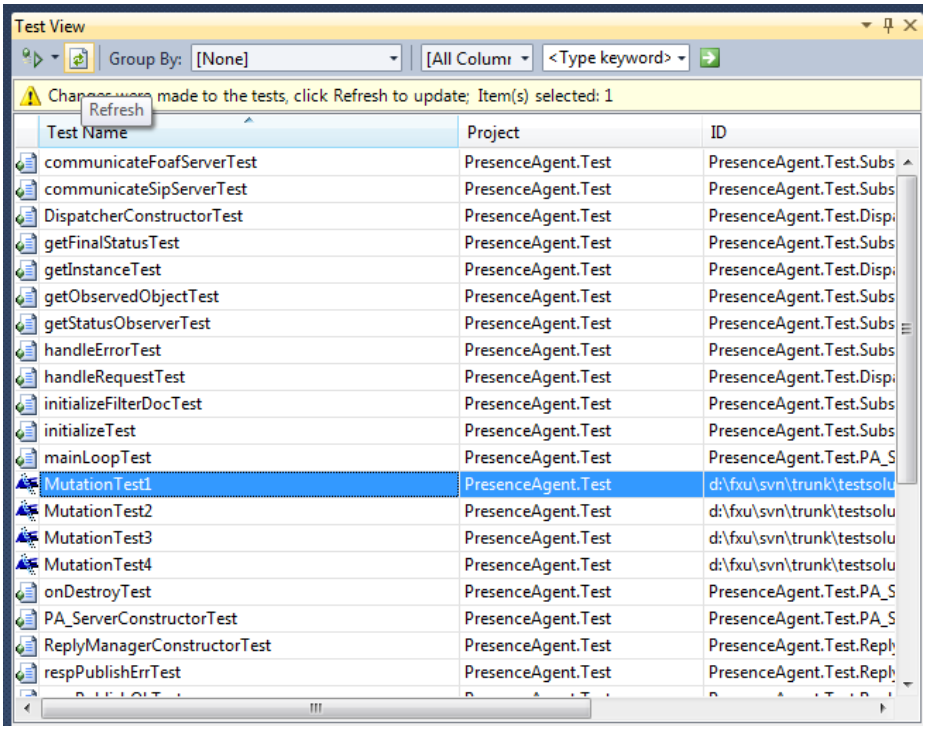
7. A file named as given in the **Test Name** box, which contains the definition of your mutation test, is added to the test project.



8. The added mutation test is opened in **Mutation Test Editor**. There is a list of mutants to run in the panel on the left side of the editor. It's empty initially. On the right there is a list containing all tests existing in the currently opened solution (building the solution is required to see new tests in the list).

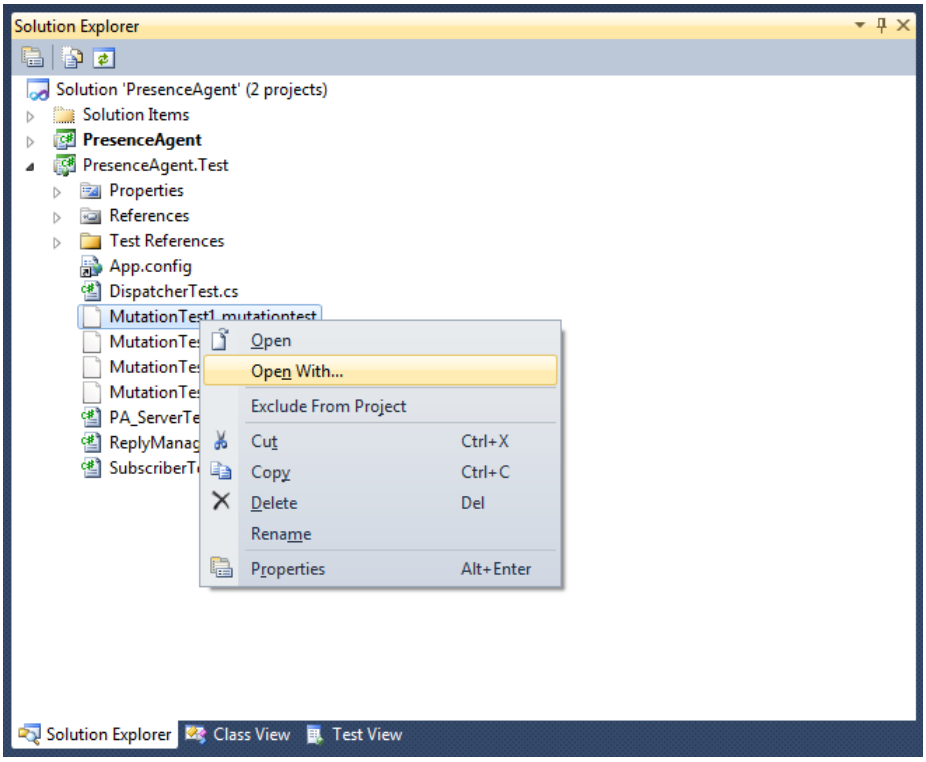


9. The added mutation test is also visible in **Test View**. If the **Test View** is not already open, on the **Test** menu, click **Windows** and then select **Test View**. Otherwise, refreshing the view may be required to see the test.

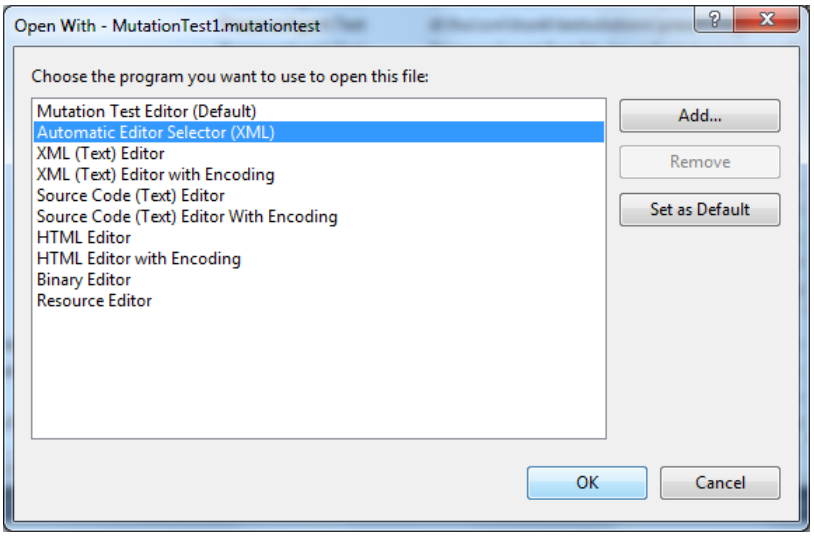


10. In **Mutation Test Editor** check tests, which will be executed for each mutant, and then save your mutation test (Ctrl+S). Tests that are checked should be prepared in accordance with the instruction **how to define a unit test for a classes that use FXU**.

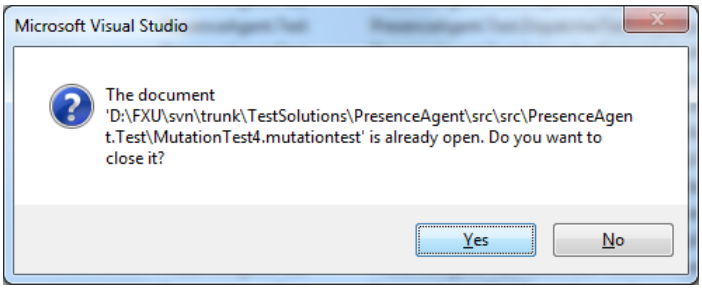
11. To add mutants to the mutation test, you need to open the test in a XML editor. Open **Solution explorer**, then right click the file containing the mutation test and select **Open with…**.



12. The **Open with …** dialog box appears. Select **Automatic Editor Selector (XML)** and click **OK**.



13. If the test was open in **Mutation Test Editor**, a warning dialog box appears. Click **Yes**.

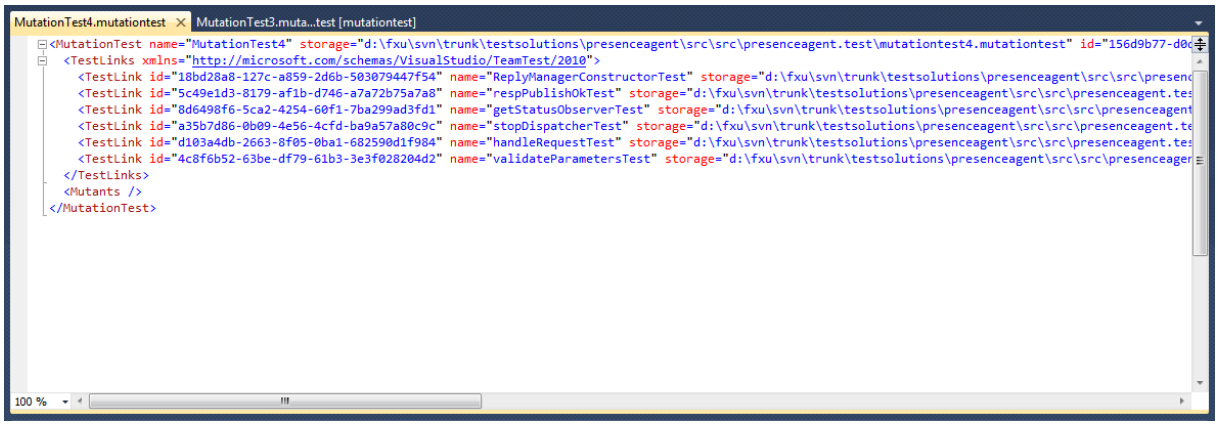


14. The mutation test is opened in **Automatic Editor Selector (XML)**. Locate a `Mutants` element in the test. All mutants should be entered as a sequence of child elements named `Mutant` of the `Mutants` element. For each mutant, following attributes have to be defined:

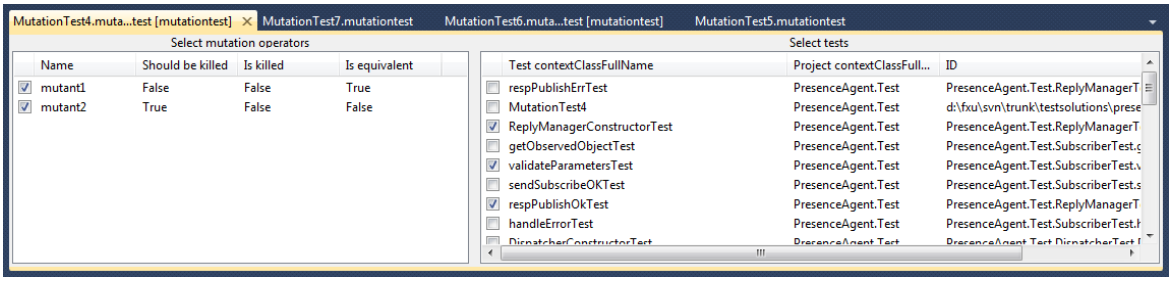
Attribute name	Type	Description
`name`	`xs:string`	A name of the mutant. The name will be displayed in the test editor window and in the the test results window.
`id`	`tt:IDSimpleType`	A GUID to identify mutants.
`shouldBeKilled`	`xs:boolean`	A value that indicates that the mutant should be killed. In future versions the value will be used to calculate the mutation test score.
`isEquivelent`	`xs:boolean`	A value that indicates that the mutant is equivalent to the original one. In future versions the value will be used to calculate the mutation test score.
`isKilled`	`xs:boolean`	A value that indicates that the mutant was killed durring the mutation test execution. In future versions the value will be used to calculate the mutation test score.
`type`	`xs:string`	A definiton of the class that implements a certain mutant type.

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tt="http://microsoft.com/schemas/VisualStudio/TeamTest/2010"
```

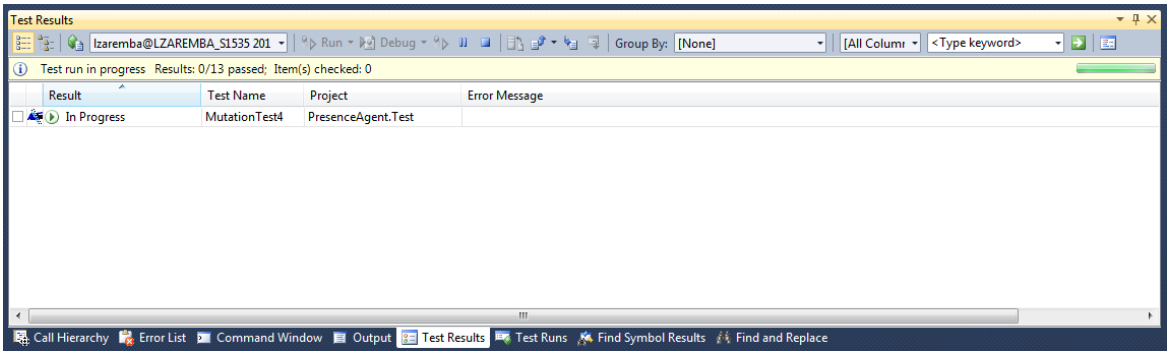
Other attributes are specific to a mutant type. At the moment, only one mutant type (**FXU Mutant**) is supported (see instruction for creation and configuration of **FXU Mutant** mutants).



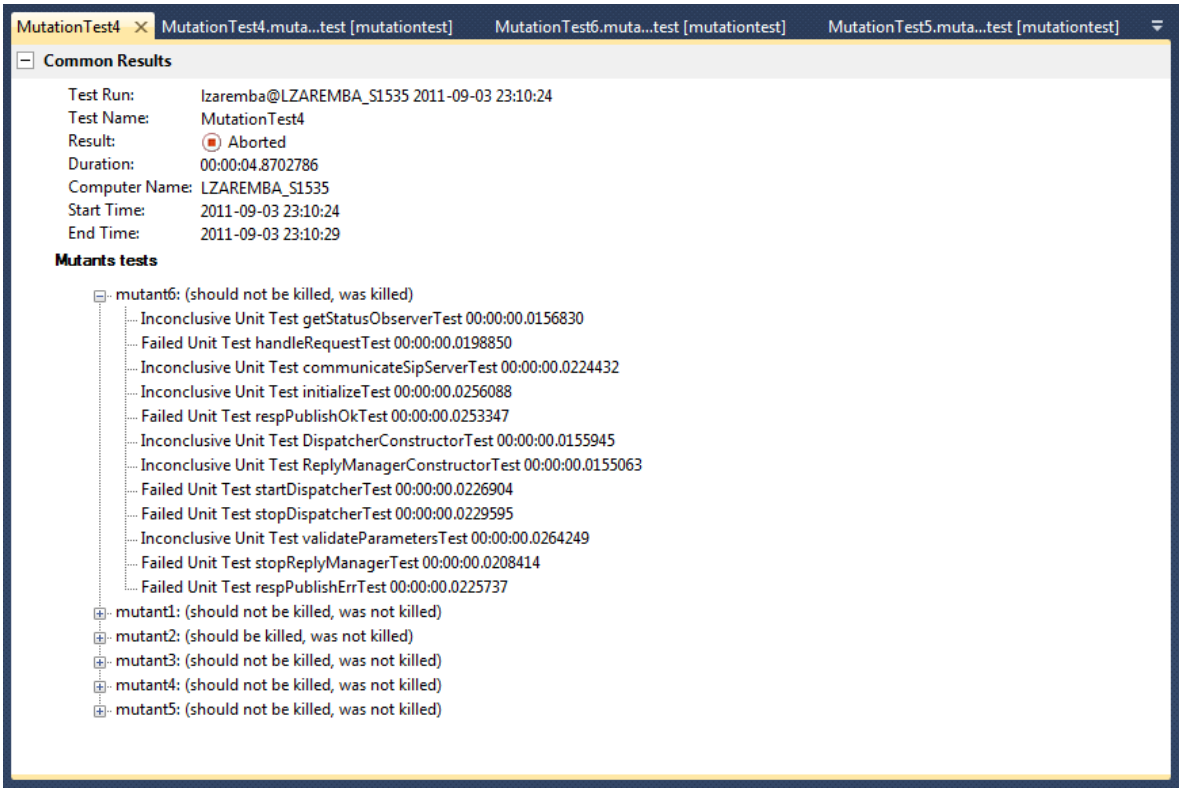
15. After adding mutants, save changes (Ctrl+S), build your solution and click refresh on the toolbar in the **Test View** window to update the list of tests. Double click on the mutation test in **Test View**. Added mutants are now visible in **Mutation Test Editor**.



16. To run the mutation test right click the test in the **Test View** list and then click **Run selection**. If the **Test Results** window is not already open, it opens now. The test runs.



17. In the **Test Results** window, right-click the row that represents your mutation test and then click **View Test Results Details**. In the **Test Results Details** page, there are summary information about the whole test execution. Detailed test results are presented bellow for each mutant, for each test.



INSTRUCTION HOW TO DEFINE A FXU MUTANT

This section describes how to configure a mutant of type **FXU Mutant**.

OVERVIEW:

Mutants of type FXU Mutant are UML state machine semantic mutants. To successful use a mutant of this kind while mutation testing, its definition and configuration are required. The definition contains basic information about the mutant and a reference to the configuration. On the other hand, the configuration consists of entries that indicate semantic variant used for all state machines, for a particular state machine or even for a particular region in a state machine.

Prerequisites:

- Visual Studio 2010 Premium or better (for information please check http://www.microsoft.com/visualstudio/2010/sysreqs).
- FXU Mutation Test Add-in into Visual Studio 2010.

MUTANT DEFINING INSTRUCTION:

1. Open Visual Studio 2010.
2. Open a mutation test in **Automatic Editor Selector (XML):**
    a. Open **Solution explorer**, then right click the file containing the mutation test and select **Open with…**.
    b. The **Open with …** dialog box appears. Select **Automatic Editor Selector (XML)** and click **OK**.
    c. If the test was open in **Mutation Test Editor**, a warning dialog box appears. Click **Yes**.
3. The mutation test is opened in **Automatic Editor Selector (XML)**. Locate a `Mutants` element in the test. All mutants should be entered as a sequence of child elements named `Mutant` of the `Mutants` element. For each mutant of type **FXU mutant**, following attributes have to be defined:

Attribute name	Type	Description
`name`	`xs:string`	A name of the mutant. The name will be displayed in the test editor window and in the test results window.
`id`	`tt:IDSimpleType`	A GUID to identify mutants.
`shouldBeKilled`	`xs:boolean`	A value that indicates that the mutant should be killed. In future versions the value will be used to calculate the mutation test score.
`isEquivelent`	`xs:boolean`	A value that indicates that the mutant is equivalent to the original one. In future versions the value will be

		used to calculate the mutation test score.
isKilled	xs:boolean	A value that indicates that the mutant was killed durring the mutation test execution. In future versions the value will be used to calculate the mutation test score.
type	xs:string	A definiton of the class that implements a certain mutant type. For mutatns of type FXU Mutant, the attribute have to have a following value: FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
unitySectionName	xs:string	The name of section in the test project configuration file that contains the semantic configuration for the mutant (unity by default). The name will be needed to configure the mutant.
unityContainerName	xs:string	The name of conainer in a configuration section in the test project configuration file that contains the sematic configuration for the mutant (unnamed container contains the configuration for non-mutated program). The name will be needed to configure the mutant.

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tt="http://microsoft.com/schemas/VisualStudio/TeamTest/2010"
```

Define as many **FXU Mutant** mutants as you need following the above instruction.

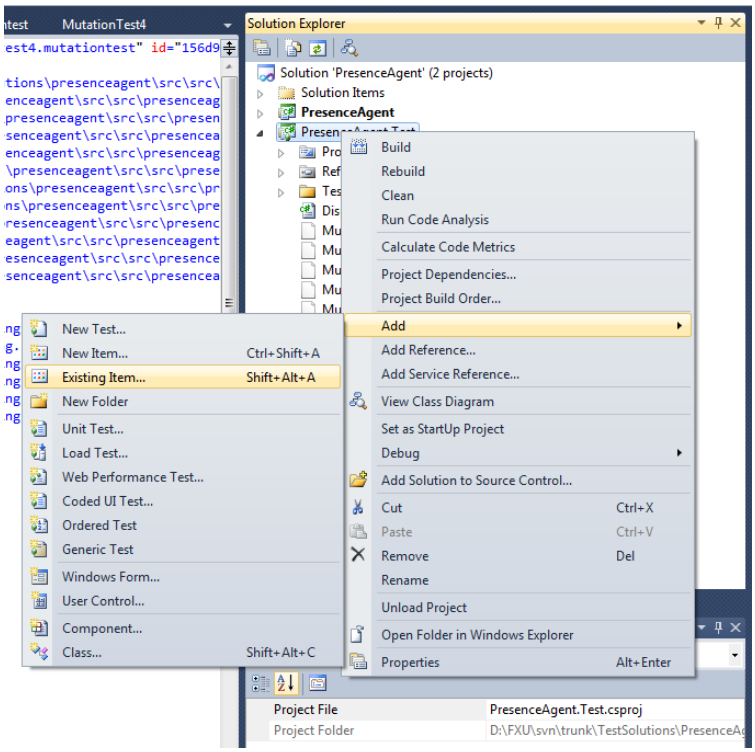
An example:

```
  <Mutants>
    <Mutant id="4e002d66-2aa9-42e0-8537-37b2c88d3fae" name="mutant1"
shouldBeKilled="false" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="true"
isKilled="false" unityContainerName="mutant1" unitySectionName="unity" />
    <Mutant id="a6d25057-8bd3-4409-9ba6-991e5fd4dc4d" name="mutant2"
shouldBeKilled="true" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="false"
isKilled="false" unityContainerName="mutant2" unitySectionName="unity" />
    <Mutant id="9fcefadb-0a6b-4130-8c62-e4200fa1e549" name="mutant3"
shouldBeKilled="false" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="true"
isKilled="false" unityContainerName="mutant3" unitySectionName="unity" />
    <Mutant id="20f1a58a-496a-4394-aad1-38484f0e0a2f" name="mutant4"
shouldBeKilled="false" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="false"
isKilled="false" unityContainerName="mutant4" unitySectionName="unity" />
    <Mutant id="2e2bbc63-264a-4e78-9298-48099f7e7df1" name="mutant5"
shouldBeKilled="false" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="true"
isKilled="false" unityContainerName="mutant5" unitySectionName="unity" />
    <Mutant id="48bba617-5b36-4a2b-8495-d1396e0623af" name="mutant6"
shouldBeKilled="false" type="FXU.TestTools.MutationTesting.FXUMutant, FXU.TestTools,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" isEquivalent="false"
isKilled="true" unityContainerName="mutant6" unitySectionName="unity" />
  </Mutants>
```

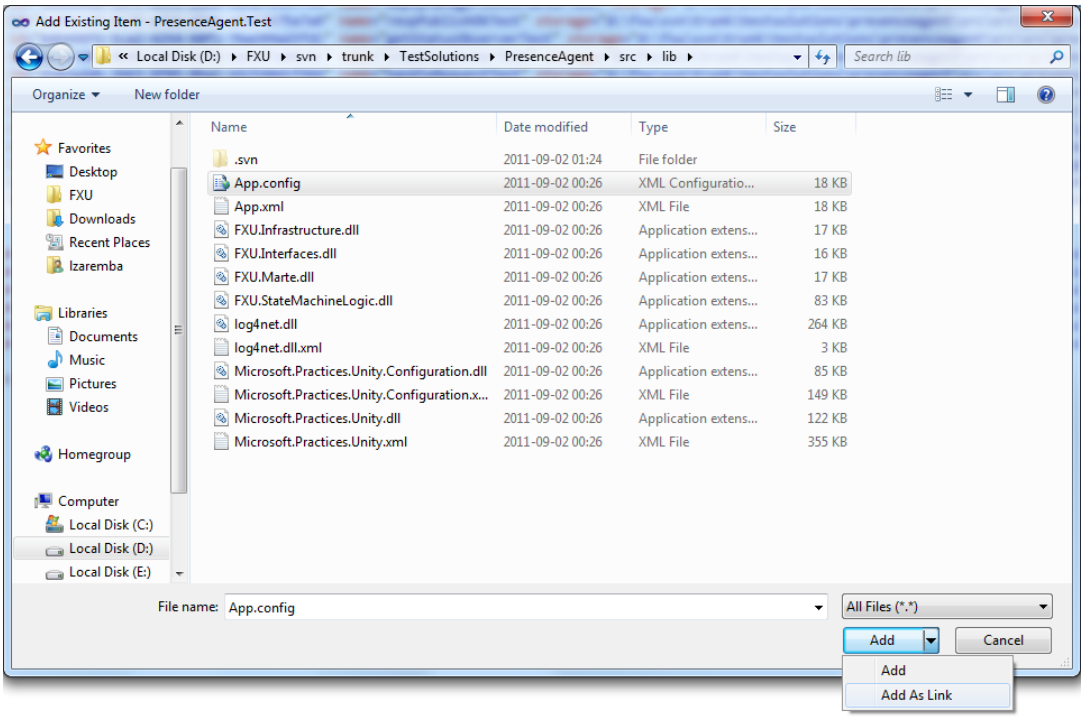
4. After adding mutants, save changes (Ctrl+S), build your solution and click refresh on the toolbar in the **Test View** window to update the list of tests. Double click on the mutation test in **Test View**. Added mutants are now visible in **Mutation Test Editor**.

CONFIGURATION OF MUTANT INSTRUCTION:

1. Open Visual Studio 2010.
2. Open the configuration file (`App.config`) included in the test project, in which tests are defined.
3. If the configuration file is not included in the test project, right click on the test project, select **Add…** and then click **Existing item**.



4. The **Add Existing Item** dialog box appears. Locate the `App.config` file generated with the project based on a UML model. Select it and click **Add As Link**. Then open the file.



5. In the file, locate the `configSections` element. Find its child element named `section` that has the `type` attribute with a value equals `Microsoft.Practices.Unity.Configuration.UnityConfigurationSection, Microsoft.Practices.Unity.Configuration`. A value of its `name` attribute is the name of the element that contains the semantic configuration of mutants. The value has to equal the value of the `unitySectionName` attribute in mutant definitions. In the bellow example, the value is `unity`.

```
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>
```

6. In the file, locate an element with the same name as the value of the `name` attribute of the element named `section` found in the previous point (`unity` in the above example). Find its child elements named `container`. The one that has not the `name` attribute, contains a configuration of the non-mutated program. Others contain configuration of single mutants.
7. To add a new mutant configuration, copy the `container` element containing the configuration of the non-mutated program. Paste the element as the last child element of the element containing all configurations (`unity` in the example).

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">

  <!-- Interfaces -->
  <namespace name="FXU.Interfaces.StateMachineElements"/>
  <namespace name="FXU.Interfaces.Events"/>
  <namespace name="FXU.Interfaces.Infrastructure"/>
  <namespace name="FXU.Interfaces.Marte"/>
  <assembly name="FXU.Interfaces"/>

  <!-- Statemachine  logic -->
  <namespace name="FXU.StateMachineLogic.UMLElementsImplementation"/>
  <namespace name="pl.edu.pw.elka.pilitowski.fxu"/>
  <namespace name="pl.edu.pw.elka.pilitowski.fxu.exceptions"/>
  <assembly name="FXU.StateMachineLogic"/>

  <!-- Infrastructure -->
  <namespace name="FXU.Infrastructure.Diagnostics"/>
  <assembly name="FXU.Infrastructure"/>

  <container>...</container>

  <container name="mutant1">...</container>

  <container name="mutant2">...</container>

  <container>...</container>

</unity>
```

8. Then add the `name` attribute to the copied `container` element. A value of the attribute have to equal the value of the `unityContainerName` attribute in the mutant definition.

```
<container>...</container>

<container name="mutant1">...</container>

<container name="mutant2">...</container>

<container name ="mutant3">
```

9. Then the copied `container` element can be configured. There are several possibilities. You can change the semantic used for all state machines, for a particular state machine or for a particular region in a state machine.
    a. To change **the general semantic used for all state machines**, in **the FXU runtime configuration parameters table** find an interface that is responsible for the semantic issue you'd like to change. Let's suppose `IEventPool` is the interface. Then in the `container` element locate its child element named `register` that has the `type` attribute equal to `IEventPool`.

```
<register type="IEventPool" mapTo="EventQueue">
   <constructor />
</register>
```

    Then you can change the class, to which the interface will be mapped, and its parameters according to **the FXU runtime configuration parameters table**. For example on `PriorityEventQueue`.

```
<register type="IEventPool" mapTo="PriorityEventQueue">
   <constructor>
      <param name="callEventPriority" value="1"/>
      <param name="changeEventPriority" value="2"/>
      <param name="signalPriority" value="3"/>
      <param name="afterEventPriority" value="4"/>
      <param name="completionEventPriority" value="5"/>
   </constructor>
</register>
```

    If the `register` element contains a child element named `lifetime`. The `lifetime` element have to remain unchanged.
    b. To change **a semantic used for a particular state machine**, in **the FXU runtime configuration parameters table** find an interface that is responsible for the semantic issue you'd like to change. Let's suppose `IEventPool` is the interface. Then in the `container` element locate its child element named `register` that has an `type` attribute equal to `IEventPool`.

```
<register type="IEventPool" mapTo="EventQueue">
   <constructor />
</register>
```

Copy the `register` element and paste it as a child element of the `container` element containing the configuration of the mutant. Then add a `name` attribute to the copied element. A value of the attribute have to equal the qualified name of the class, that is the owning class of the state machine, whose semantic you'd like to change. Finally you can change the class, to which the interface will be mapped, and its parameters according to **the FXU runtime configuration parameters table**. For example on `PriorityEventQueue`.

```
<register type="IEventPool"
name="PresenceAgent.presenceAgentController.Subscriber"
mapTo="PriorityEventQueue">
    <constructor>
        <param name="callEventPriority" value="1"/>
        <param name="changeEventPriority" value="9"/>
        <param name="signalPriority" value="3"/>
        <param name="afterEventPriority" value="4"/>
        <param name="completionEventPriority" value="7"/>
    </constructor>
</register>
```

NOTE: If the `register` element contains child element named `lifetime`, the `register` element should not be defined for a particular state machine.

c. To change **a semantic used for a particular region in a particular state machine** in the `container` element locate its child element named `register` that has the `type` attribute equal to `IRegion`.

```
<register type="IRegion" mapTo="Region">
    <constructor>
            <param name="defaultEntryRule" dependencyType="IDefaultEntryRule"/>
    </constructor>
</register>
```

Copy the `register` element and paste it as a child element of the `container` element containing the configuration of the mutant. Add a `name` attribute to the copied element. A value of the attribute have to equal the qualified name of the class, that is the owning class of the state machine, whose semantic you'd like to change, concatenated with "." and then concatenated with the qualified name of the region whose semantic you'd like to change. Then set priorities of actions (entry, execution, exit), which will be performed during an execution. Priorities are described in **the FXU runtime configuration parameters table**. An example:

```
<register type="IEventPool"
name="PresenceAgent.presenceAgentController.Subscriber.PresenceAgent::presenceAg
entController::Subscriber::SubscriberStateMachine::SubscriberMainRegion::Adapter
Communication::Region1" mapTo="PriorityEventQueue">
    <constructor>
        <param name="defaultEntryRule" dependencyType="IDefaultEntryRule"/>
        <param name="entryPriority" value="9"/>
```

```
        <param name="executionPriority" value="3"/>
        <param name="exitPriority" value="4"/>
    </constructor>
</register>
```

NOTE: If a state machine have to execute actions according to the custom priorities semantic (see **the FXU runtime configuration parameters table**, row no. 15), then the state machine takes into account priorities of actions in regions.

10. Save the configuration file. Your mutant has already been configured.

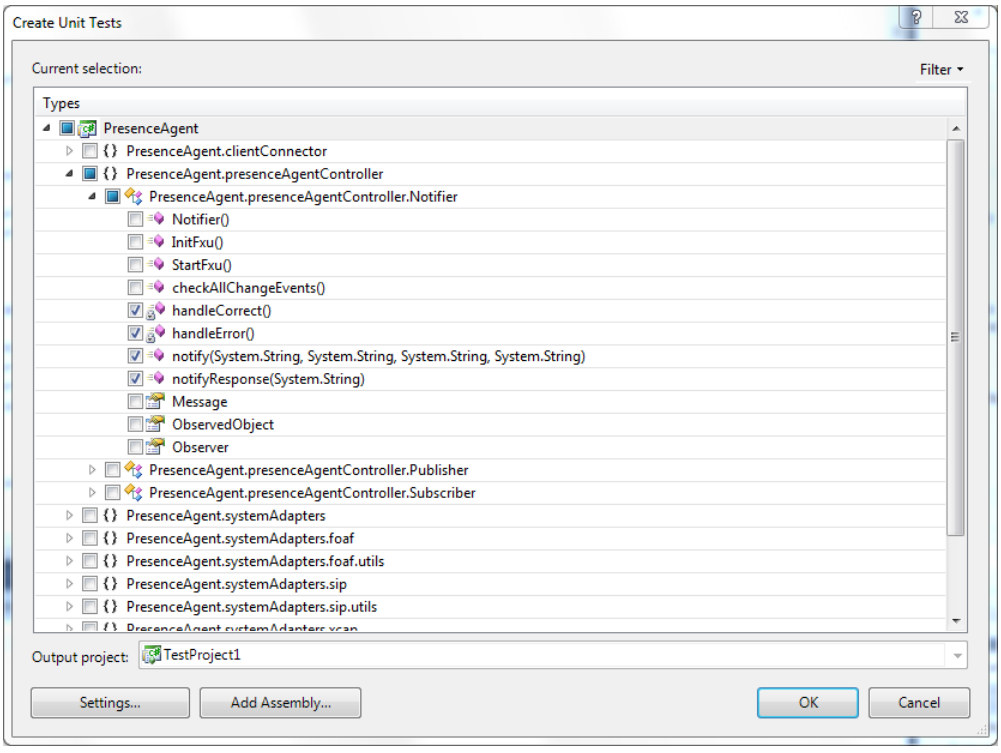
INSTRUCTION HOW TO DEFINE UNIT TESTS FOR CLASSES THAT USE FXU

This section describes how to define a unit test for a classes that use FXU.

INSTRUCTIONS:

1. Open **Visual Studio 2010**.
2. Open a solution generated from a UML model using **FXU Generator**.
3. On the **Test** menu, click **New Test…**. The **Add New Test** dialog box appears. In the list of **Templates**, select **Unit Test Wizard**. For **Add to Test Project**, select an existing test project or select **Create a new Visual C# test project…**and then click **OK**.

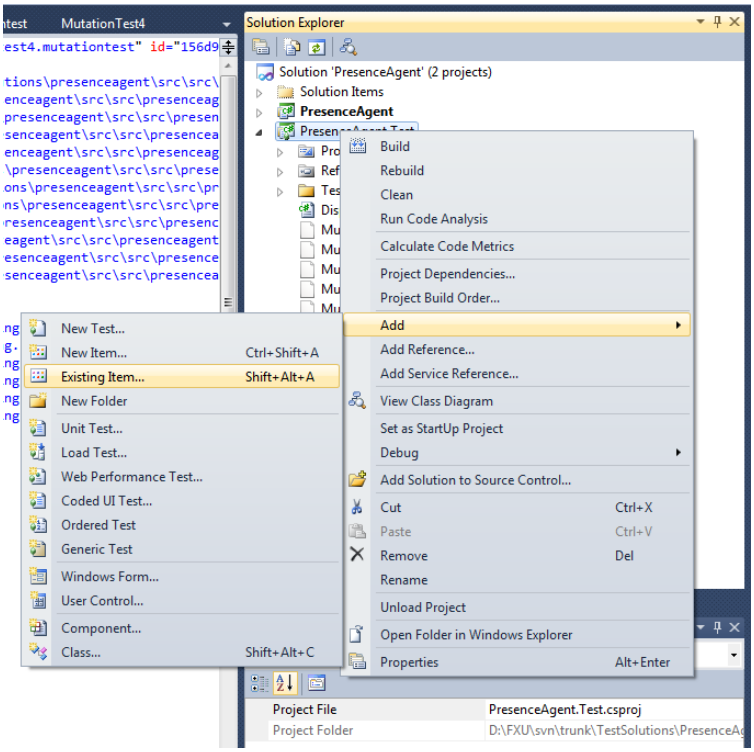
   If **Create a new Visual C# test project…**option was selected, the **New Test Project** dialog box appears. In the **Enter a name for your new project** box, type a name of the test project and then click **Create**. This creates a project, which is displayed in **Solution Explorer**.
4. The **Create Unit Tests** dialog box is displayed. Under **Current selection**, a tree structure shows the class and member hierarchy of the assembly of the project generated from a UML model. You can use this page to generate unit tests for any selection of those members. In the tree structure, select methods you'd like to test. Then in the **Create Unit Tests** dialog box, click **OK**.



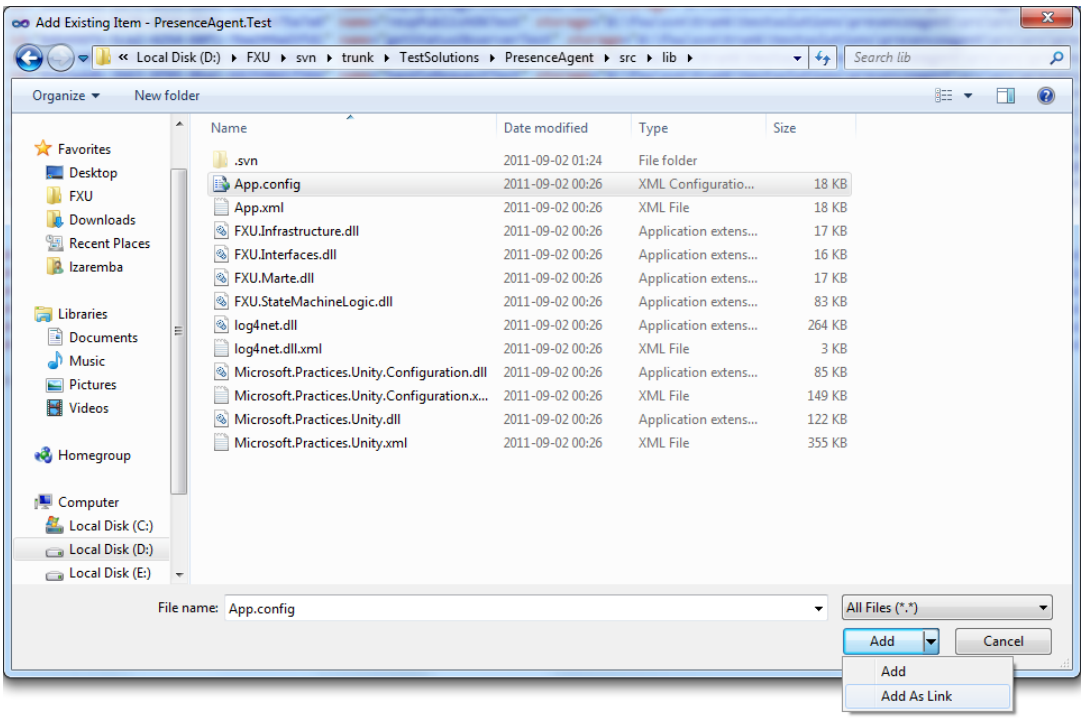
5. Files containing test classes are added to the selected test project. In general, a test class contains not only the individual test methods, but various methods for initializing and cleaning up tests as well. In fact, the **Create Unit Tests** wizard added some of these additional methods to the test class when it is created. **In each test class** locate `Additional test attributes` region and expand it.
6. Uncomment the method with the `[ClassInitialize()]` attribute and type the following line as the method body: `FXU.Infrastructure.Unity.FXUUnityContainer.Run();`. An example:

```
#region Additional test attributes
//
//You can use the following additional attributes as you write your tests:
//
//Use ClassInitialize to run code before running the first test in the class
[ClassInitialize()]
public static void MyClassInitialize(TestContext testContext)
{
   FXU.Infrastructure.Unity.FXUUnityContainer.Run();
}
//
//Use ClassCleanup to run code after all tests in a class have run
//[ClassCleanup()]
//public static void MyClassCleanup()
//{
//}
//
//Use TestInitialize to run code before running each test
//[TestInitialize()]
//public void MyTestInitialize()
//{
//}
//
//Use TestCleanup to run code after each test has run
//[TestCleanup()]
//public void MyTestCleanup()
//{
//}
//
#endregion
```

7. If at the second point you created a new test project, there is a need to link a configuration file to the test project. Right click on the test project, select **Add…** and then click **Existing item**.



The **Add Existing Item** dialog box appears. Locate the `App.config` file generated with the project based on a UML model, select it and click **Add As Link**. Then open the file.



8. Rebuild the solution. Your tests have already been adapted to execution with the FXU runtime.