

Using TESSY for C++

Abstract

This document describes how to test C++ code with TESSY.

Table of Contents

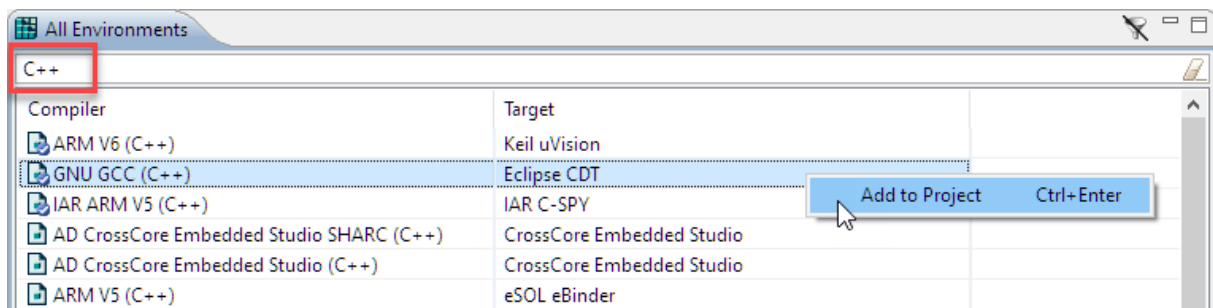
Abstract	1
1 Introduction.....	2
1.1 Enable C++ within TEE	2
1.2 Setup of a new C++ Module	2
1.3 Module interface.....	3
1.4 Creating test cases.....	5
1.4.1 Testing constructors.....	5
1.4.2 Testing destructors.....	6
1.4.3 Testing class methods	7
1.4.4 Creating instances	8
1.4.5 Checking for exceptions	10
2 Example	11
2.1 Step 1: Create a module with some test cases	12
2.2 Step 2: Create the instance object	12
2.3 Step 3: Add test data.....	14
2.4 Step 4: Review the coverage	14
2.5 Step 5: Create a test report	15
3 Special Use Cases	16
3.1 Declared but not defined classes	16
3.2 Stubs for Constructors.....	16
3.3 Attribute 'Generate Parameter Proxies'	17
3.4 Attribute 'Generate Constructors'	19
3.5 Attribute 'Enable Generate Default Constructors'	19

1 Introduction

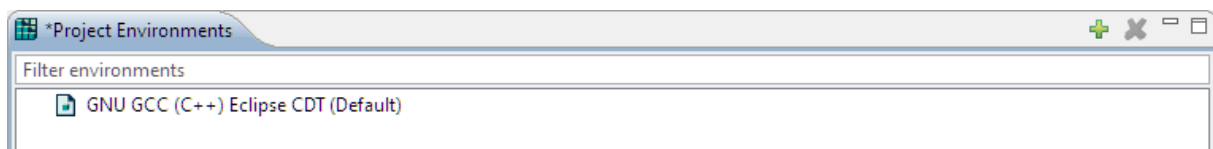
Testing of C++ modules requires the same setup of the TESSY modules as for normal C code: You need to specify the source files and select the required C++ compiler/target environment. If your specific environment is not available yet within the TESSY Environment Editor (TEE), please contact technical support. New C++ environments for the available C code compiler/target environments will be added on demand.

1.1 Enable C++ within TEE

You can filter all C++ compilers that are currently supported within the **All Environments** view of TEE (refer to the **GNU GCC (C++) Eclipse CDT** entry shown in the figure below). To enable the C++ compiler, you need to add the environment to the project.

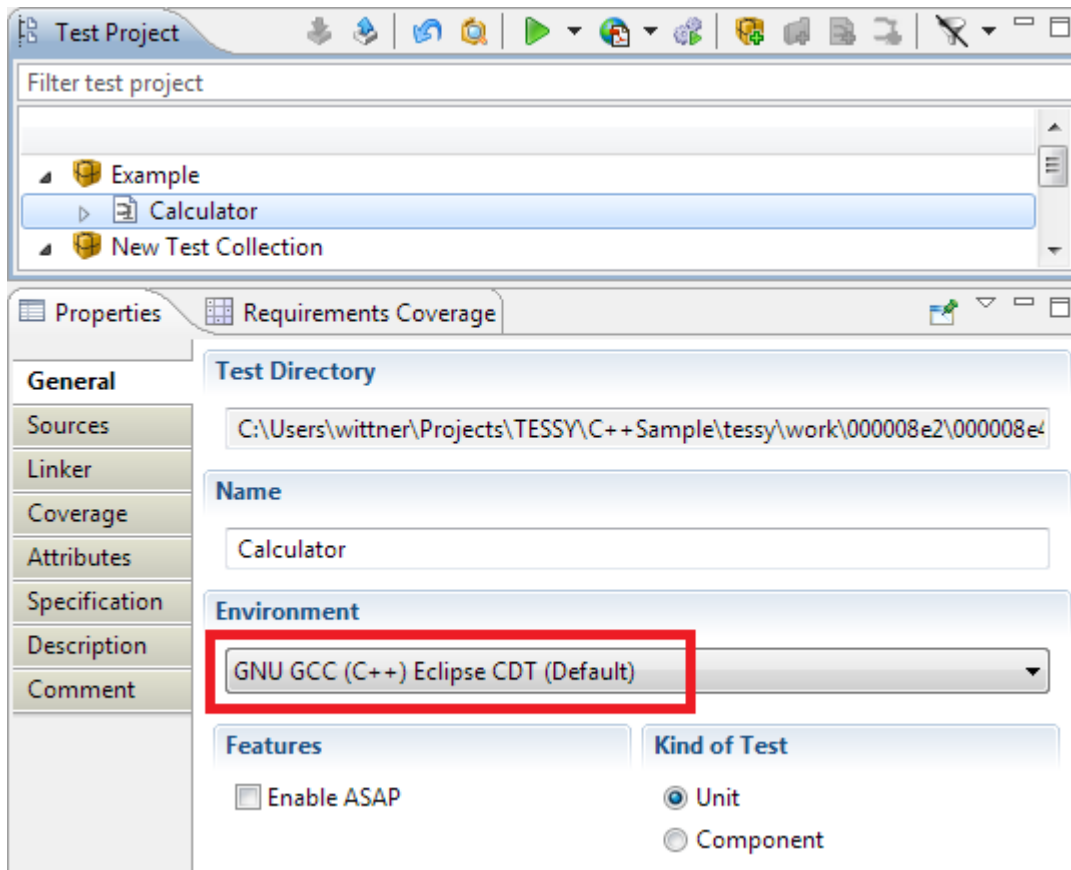


The C++ environment will now be available within the **Project Environments** view of TEE like shown below:



1.2 Setup of a new C++ Module

When creating a new module for C++ code, select the respective C++ environment for the desired compiler as shown below:

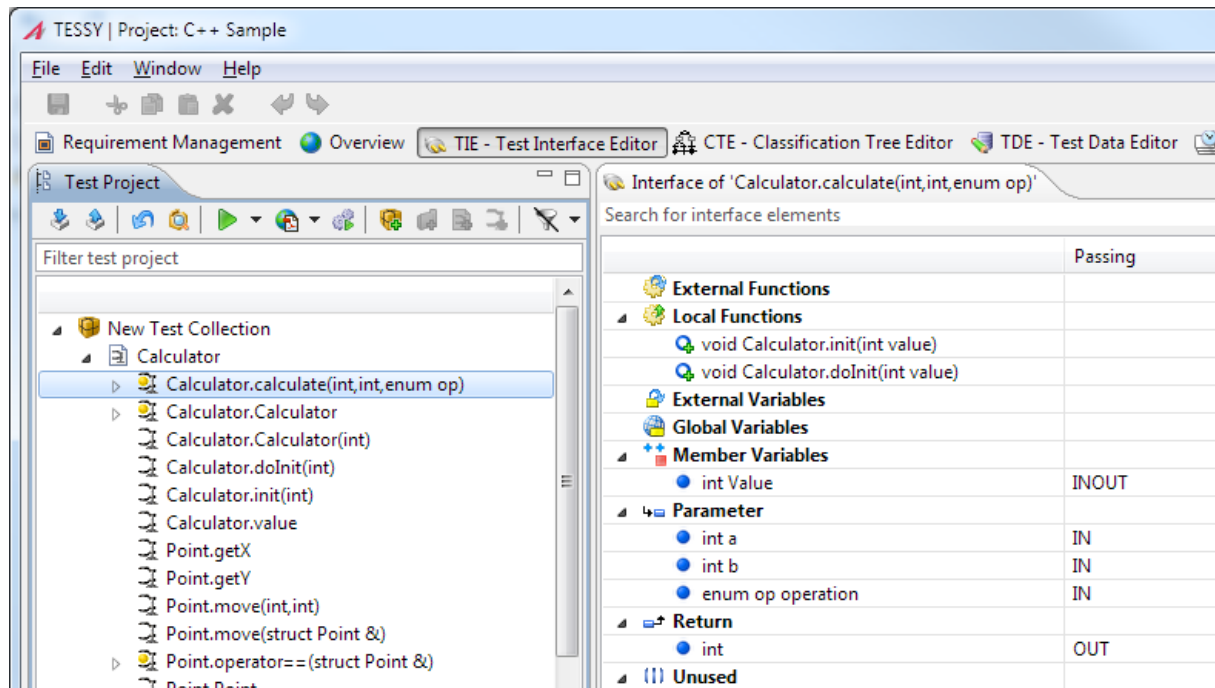


Then add the source files of your module, specify include paths, defines, and other settings as for normal C modules.

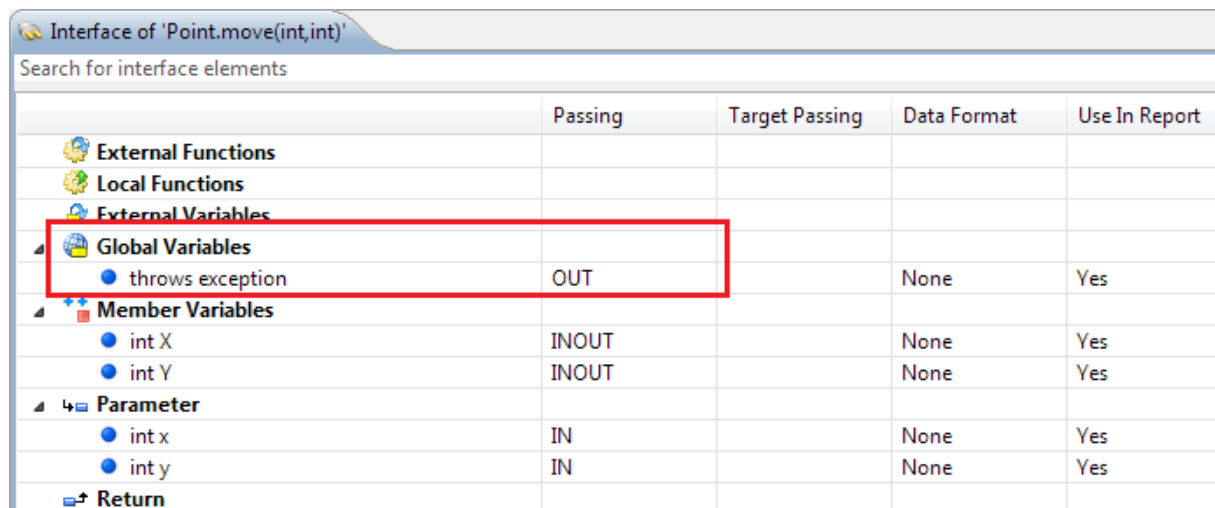
1.3 Module interface

When opening the module TESSY will analyze the source file(s) and provide the list of class constructors, methods, and functions for all classes contained within the given source file(s) of that module.

The interface of each testable method will be displayed within the TIE. Additionally, it will show the member variables of the class.



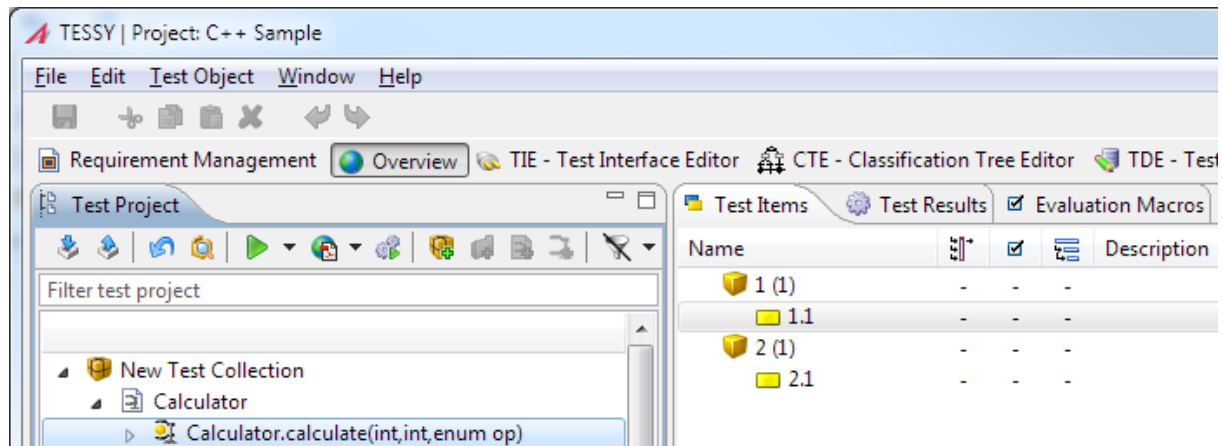
Since every method or function can throw exceptions, there is a special global variable named “throws exception”, which is an output value. You need to specify for each test step if an exception will be thrown or not.



If you don't want to check for exceptions, you can set the passing direction of this variable to IRRELEVANT.

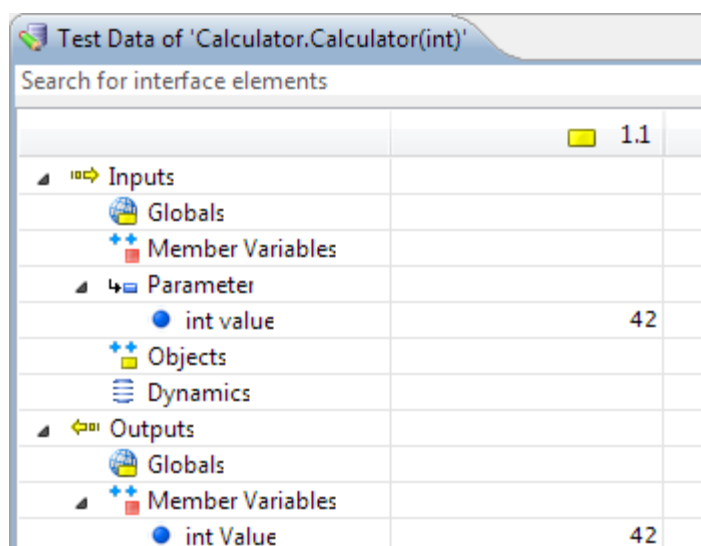
1.4 Creating test cases

You can specify test cases and test steps as for normal C code modules using either the CTE or by creating test cases manually.



1.4.1 Testing constructors

The default constructor without any arguments is always defined for each C++ class. If there are additional constructors defined for the C++ classes within your source files, they will appear as test objects within TESSY. Since constructors can contain initialization code for the member variables of a class instance, it is worth testing each constructor. You just need to specify the parameters to be passed as arguments to the constructor within TDE as shown below:



The result of the constructor invocation is an instance of the given class whose member variables can be checked.

1.4.2 Testing destructors

As described above, the test binary contains a global object of your class. Since calling the destructor of a C++ object destroys the object, it would not be possible to read the object's non-static members after the destructor was called. Thus destructors do not occur within the TDE. Nevertheless is it possible to test them. Since TESSY cannot read values from destroyed objects, you have to instantiate the object by yourself by using an auxiliary function as shown in the example below.

```
class Class {
public:
    Class(int value);
    ~Class();

private:
    int x;
    static int y;
};

int Status;

Class::Class(int value) : x(1) {
    Status = 1;
    y = value;
}


Class::~~Class() {
    Status = 0;
    x = 0;
    y = 0;
}

#ifdef TESSY

void Class_Dtor_Test(int value) {
    Class TestClass(value);
}

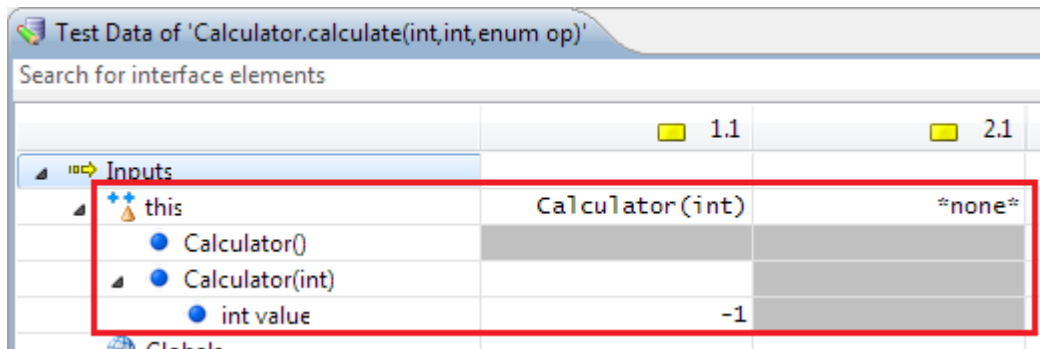
#endif
```

The example contains a non-static member `x`, a static member `y` and a global variable `Status`. Function `Class_Dtor_Test()` creates a local instance of class `Class` named `TestClass`. At the end of the function the destructor is called because the object is a local variable. Though `x` cannot be read anymore, `y` and `Status` are still available in memory and will be displayed in TDE as shown below.

	 1.1	
Inputs		
Globals		
Parameter		
int value		5
Dynamics		
Outputs		
Globals		
unsigned int Class.y		0
int Status		0

1.4.3 Testing class methods

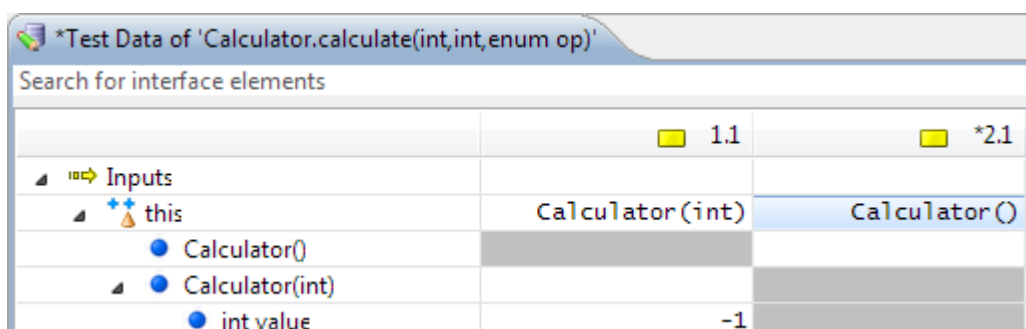
Unlike normal C tests you always need a C++ object to execute any method. You might also need other C++ objects for the parameters of the method. Therefore there is a new section within TDE containing the special “**this**” pointer:



In the screenshot above, the first test case will be executed using a test object instance created with the `Calculator(int)` constructor. This same instance will also be used for the second test case because of the `*none*` value for the “**this**” variable.

Please note: If you would like to execute test cases individually (especially when they have multiple test steps) it would be useful to specify a new “**this**” value for each first test step: Otherwise you would always need to execute the first test case containing the test object constructor.

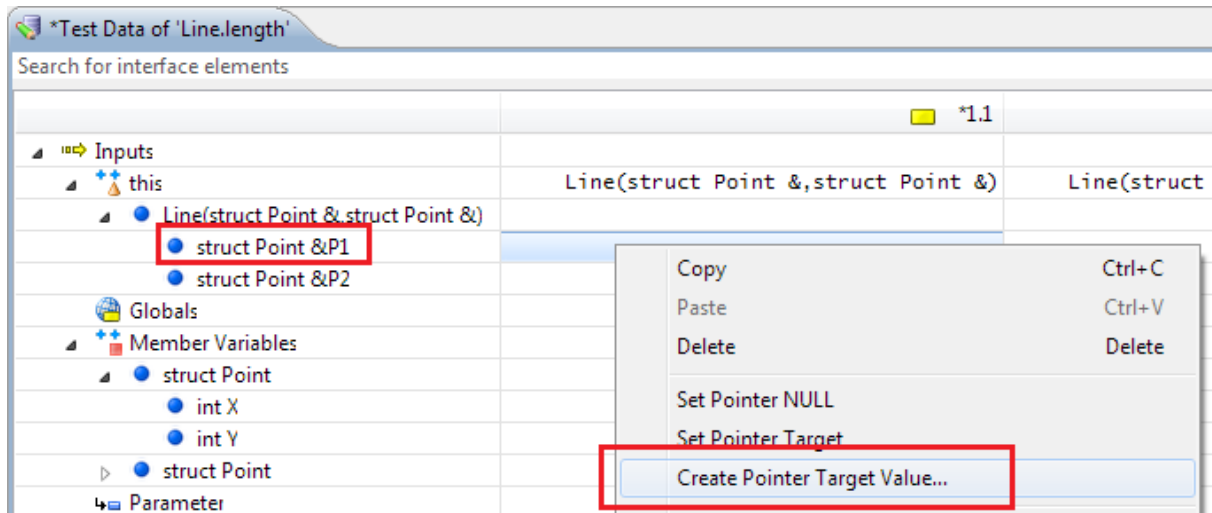
The parameter values for constructors are shown as child elements of the “**this**” variable and can be filled like normal variables:



In the example above the first constructor is called with the -1 value whereas the second constructor doesn't require any argument so that there are no fields accessible for passing parameter values.

1.4.4 Creating instances

If C++ methods require class instances to be passed as arguments, such instances can be created as with pointer variables using the context menu as shown below:



The new instance will appear within the **Objects** section and can be filled like normal variables (even recursively as shown below for the **Line** constructor with **Point** arguments):

Test Data of 'Line.length'

Search for interface elements

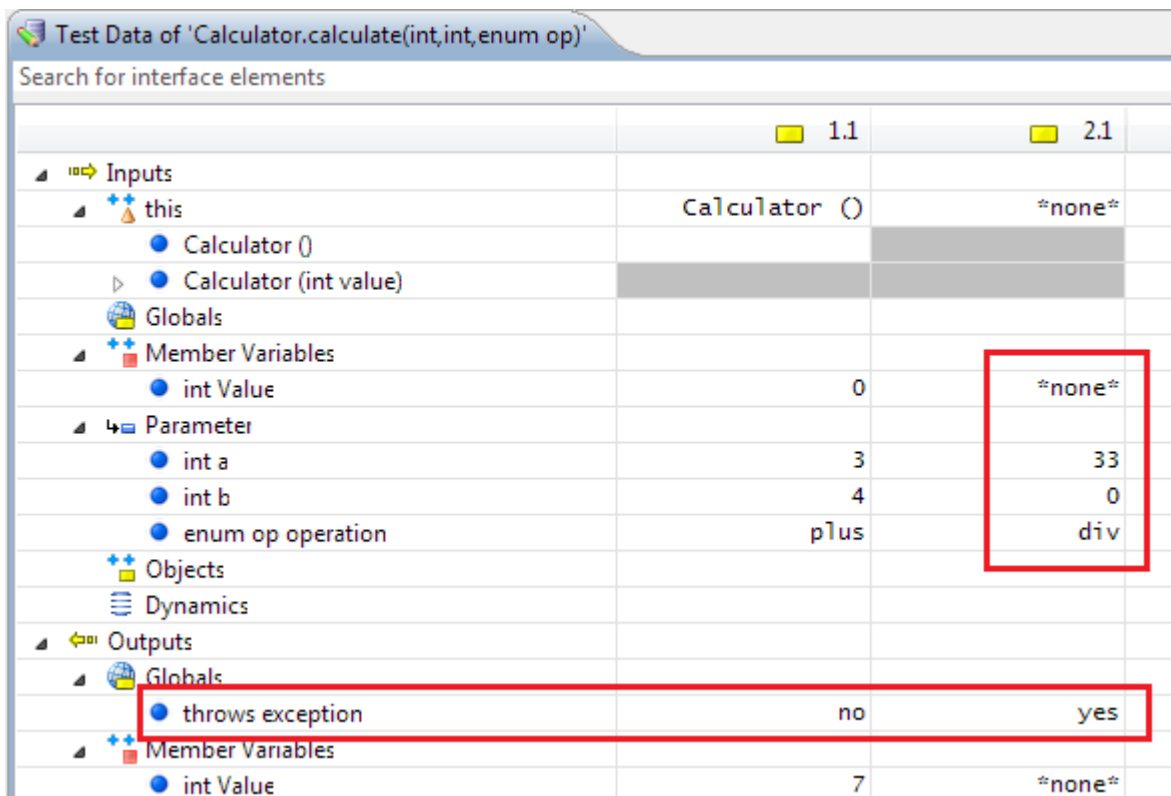
	1.1
Inputs	
this	Line(struct Point &,struct Point &)
Line(struct Point &,struct Point &)	
struct Point &P1	P1
struct Point &P2	P2
Globals	
Member Variables	
struct Point	
int X	*none*
int Y	*none*
struct Point	
Parameter	
Objects	
P1	Point(int,int)
Point()	
Point(int,int)	
int x	1
int y	1
Point(struct Point &)	
P2	Point(int,int)
Point()	
Point(int,int)	
int x	3
int y	3
Point(struct Point &)	

1.4.5 Checking for exceptions

When testing methods that can throw exceptions, you need to specify if an exception is expected for the execution of a test step. Suppose you are testing the code below, exceptions will be thrown for division by zero or an unsupported operation being passed as parameter.

```
int Calculator::calculate(int a, int b, enum op operation) {
    switch (operation) {
        case plus:
            return a + b;
        case minus:
            return a - b;
        case mul:
            return a * b;
        case div:
            if (b == 0) {
                throw "Division by 0";
            } else {
                return a / b;
            }
    }
    throw "Unsupported operation";
}
```

Within the example above, the method “calculate” will throw an exception when passing zero as second argument for the division operation. The respective test data and the setting for the “throws exception” variable are shown below:



Search for interface elements		1.1	2.1
Inputs			
this	Calculator ()		*none*
Calculator (int value)			
Globals			
Member Variables			
int Value	0		*none*
Parameter			
int a	3		33
int b	4		0
enum op operation	plus		div
Objects			
Dynamics			
Outputs			
Globals			
throws exception	no		yes
Member Variables			
int Value	7		*none*

2 Example

We will use the following C++ sample project within the TESSY installation directory:

```
C:\Program Files\Razorcat\TESSY_4.1\Examples\C++\tessy\tessy.pdbx
```

This is a readily prepared sample project that can be cloned into a location of your choice. When cloning the sample project you need to specify the desired project root location and all contents of the sample project will be copied into the new location.

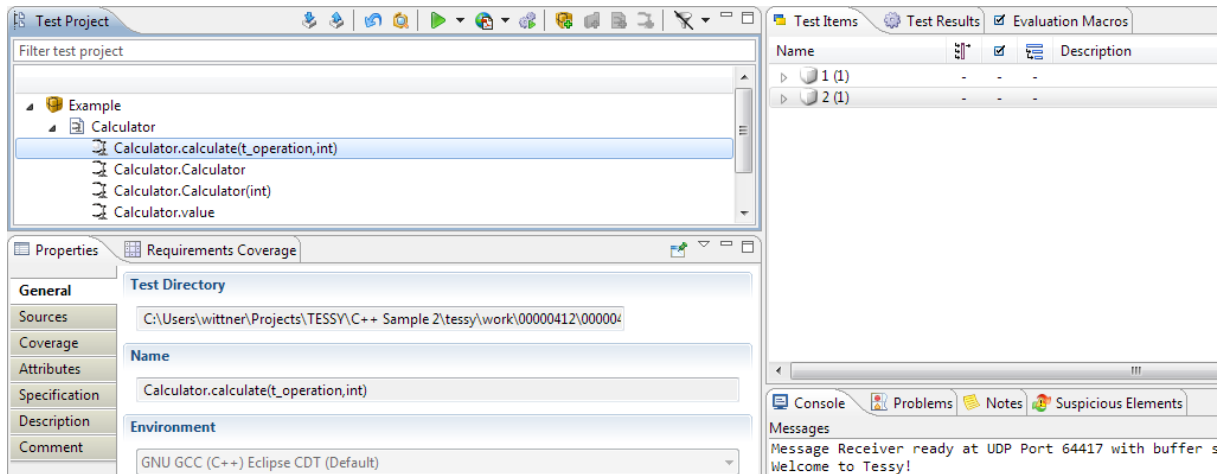
The following steps exercise the creation of C++ test cases for the “Calculator” example source file as shown below:

```
C:\Program Files\Razorcat\TESSY_4.1\Examples\C++\source\calculator.cpp
```

```
1  #include "calculator.h"
2
3  //
4  // Calculator
5  //
6  Calculator::Calculator() {
7  |   Value = 0;
8  | }
9
10 Calculator::Calculator(int value) {
11 |   Value = value;
12 | }
13
14 //
15 void Calculator::calculate(t_operation operation, int operand) {
16 |   switch (operation) {
17 |     case plus:
18 |       Value += operand;
19 |       break;
20 |     case minus:
21 |       Value -= operand;
22 |       break;
23 |     case mul:
24 |       Value *= operand;
25 |       break;
26 |     case div:
27 |       if (operand == 0) {
28 |         throw "Division by 0";
29 |       } else {
30 |         Value /= operand;
31 |       }
32 |       break;
33 |     default:
34 |       throw "Unsupported operation";
35 |   }
36 | }
37
38 int Calculator::value(void) {
39 |   return Value;
40 | }
```

2.1 Step 1: Create a module with some test cases

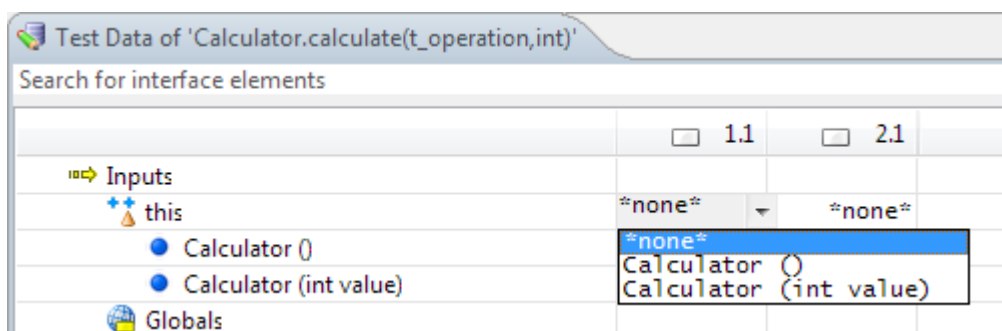
Create a new module, choose the C++ environment, open the module and create two test cases for the test object “Calculator.calculate(t_operation,int)” as shown below:



As you can see, the test object name includes the whole signature of the respective method which allows you to distinguish between overloaded methods with equal names but different parameter lists.

2.2 Step 2: Create the instance object

Switch to TDE perspective in order to enter test data. The first step is to choose the constructor for the instance to be created: An instance of the “Calculator” class is required to call any methods on it. The “Calculator” class provides two different constructors which can be chosen via the combo box of the “this” variable.



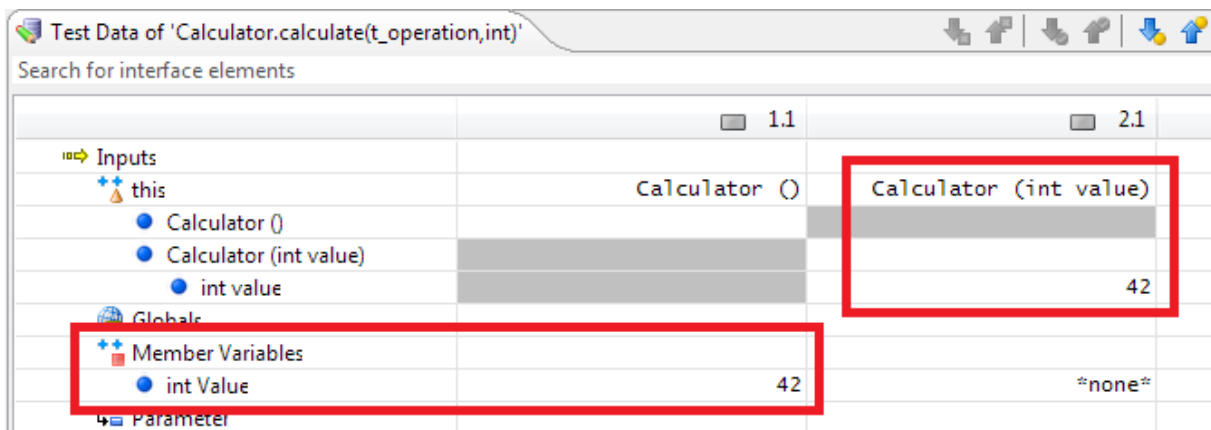
You can choose different constructors for each of your test cases or use the same instance for all test cases. Your choice has the following consequences:

- If you create an instance within the first test case and specify “*none*” for all other test cases, the same instance will be reused for execution of each test case (i.e. the tested method will be called with the same instance and the instance will retain internal states after each invocation).

- If you create new instances for each test step, every execution of the tested method will use another instance (i.e. the instance will always be in initial state when calling the method).

It depends on your test object which option is appropriate. Please keep in mind that when specifying the instance only within the first test case, you cannot execute one of the other test cases separately because the required instance object would be missing (The same applies when running test cases using the “Test Cases Separately” test execution option).

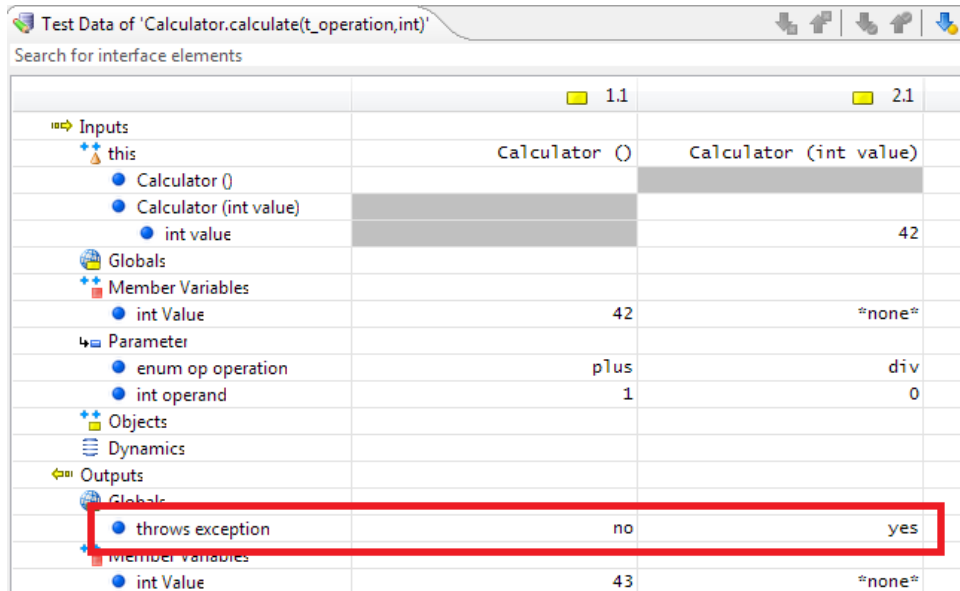
We will use two different constructors for our two test cases as shown below:



Within the first test case we use the default constructor without arguments and initialize the member variable “Value” of the instance afterwards. For the second test case we use another constructor with a number argument that will initialize the member variable directly. Therefore we do not provide a value for the member variable “Value” because this would overwrite the value passed via the constructor.

2.3 Step 3: Add test data

We will do a simple calculation for the first test case and force an exception with a division by zero for the second test case.

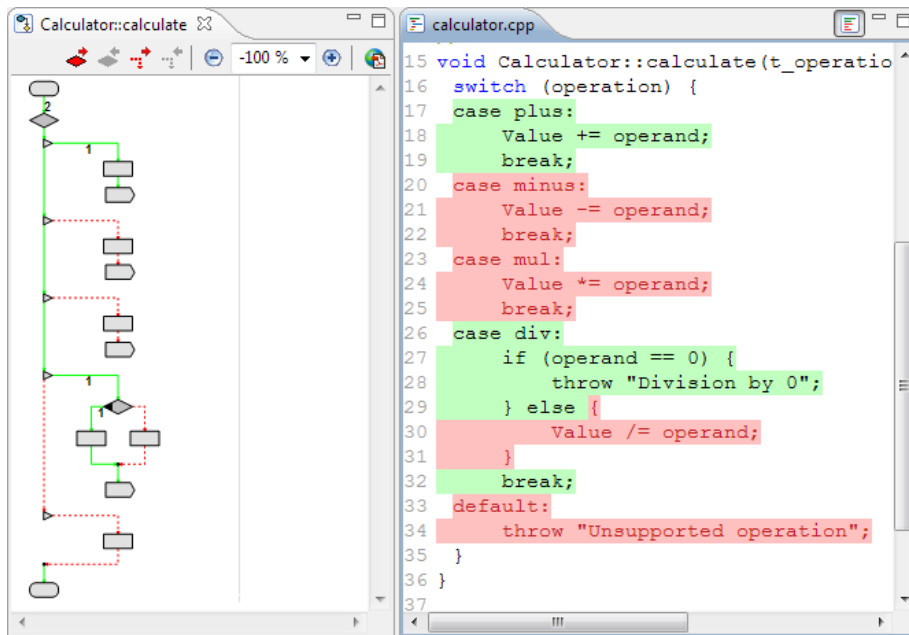


	1.1	2.1
Inputs		
this	Calculator ()	Calculator (int value)
Calculator ()		
Calculator (int value)		
int value		42
Globals		
Member Variables		
int Value	42	*none*
Parameter		
enum op operation	plus	div
int operand	1	0
Objects		
Dynamics		
Outputs		
Globals		
throws exception	no	yes
Member Variables		
int Value	43	*none*

Please note the setting for the “throws exception” variable: We expect an exception for the second test case. If there would be no exception, then the test case would fail.

2.4 Step 4: Review the coverage

After execution of the test case with branch coverage selected, switch to the coverage perspective and review the achieved coverage. The first switch case should be executed and the branch containing the division by zero exception:



2.5 Step 5: Create a test report

Within the test report you will see the constructor as well as the special “throws exception” variable with their respective values:

Test Case 1			
Test Step 1.1 (Repeat Count = 1)			
Name	Input Value		
this	Calculator ()		
this.Calculator.Value	42		
operand	1		
operation	plus		
Name	Actual Value	Expected Value	Result
this.Calculator.Value	43	43	✓
throws exception	no	no	✓

Test Case 2			
Test Step 2.1 (Repeat Count = 1)			
Name	Input Value		
this	Calculator (int value)		
this.Calculator (int value).value	42		
operand	0		
operation	div		
Name	Actual Value	Expected Value	Result
throws exception	yes	yes	✓

3 Special Use Cases

In this chapter is a description for special use cases that may occur when testing C++ code. Examples for these cases are also available in the example project within the TESSY installation directory:

```
C:\Program Files\Razorcat\TESSY_4.1\Examples\C++\tessy\tessy.pdbx
```

3.1 Declared but not defined classes

If classes are declared and not defined in the source, but are needed to create objects used for the test object, you will get an error when compiling the test driver like:

```
error: invalid use of incomplete type 'class Unused'  
note: forward declaration of 'class Unused'
```

In this case the classes must be defined in the user code **Definitions** of the respective module.

Definitions (Module specific)

```
class Unused{};
```

Refer to the **Definitions** contents of the module within the C++ example project.

3.2 Stubs for Constructors

Constructors that must initialize members in the member initializer list, or that must call a base class constructor with a parameter, must be defined by the user in the **Definitions** of the module with the proper initialization list.

Definitions (Module specific)

```
Class::Class(Used& used, Unused &unused)  
: m_used(used), m_unused(unused) {}
```

Refer to the **Definitions** of the module in the C++ example project.

3.3 Attribute 'Generate Parameter Proxies'

When class reference parameters are passed to the test object it is in some cases not possible that TESSY can set these parameters correctly, especially when the parameter is a template class or an abstract base class.

In these cases, a parameter proxy has to be used and the parameter has to be set and evaluated in the user code.

The following code is used for this example:

```

parameter_proxy.cpp
1
2 #include <ostream>
3
4 class Point {
5 public:
6     Point(int x, int y) : X(x), Y(y) {}
7
8     int X;
9     int Y;
10 };
11
12 // stream operator to write line coordinates
13 std::ostream &operator<<(std::ostream &out, const Point &point)
14 {
15     out << "X: " << point.X << ", Y: " << point.Y;
16     return out;
17 }
18

```

The function operator<< has to be tested but the ostream parameter cannot be set or evaluated in TDE. A parameter proxy has to be used.

Set the Attribute **Generate Parameter Proxies** to true for the test object operator<<.

Coverage	
Attributes	
Name	Value
<input checked="" type="checkbox"/> Generate Parameter Proxies	true

Set the **Pass Direction** of the parameter to EXTERN.

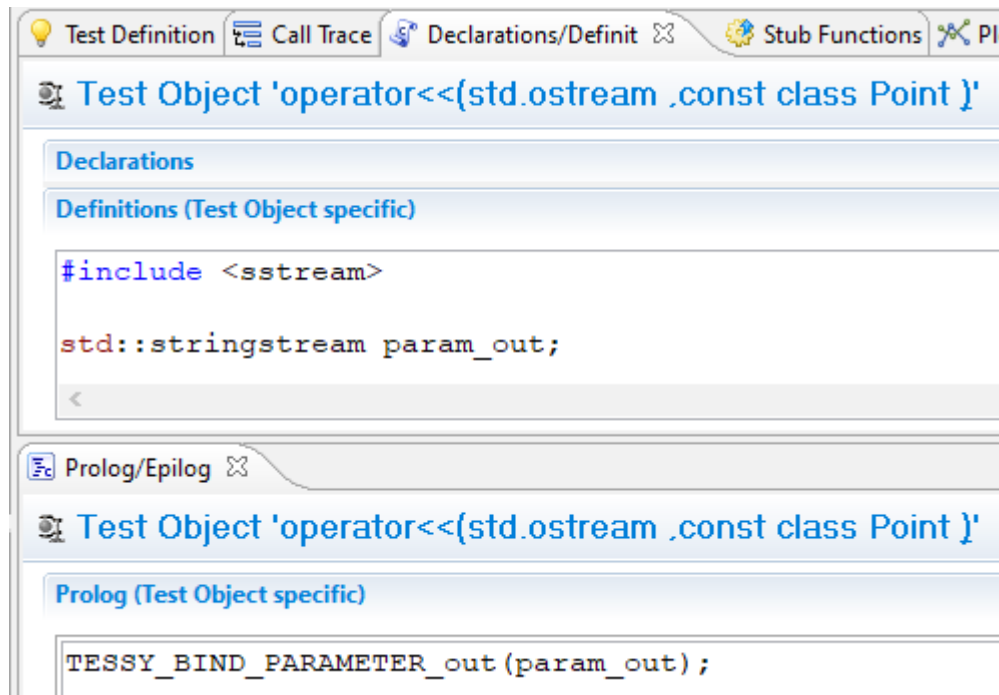
Parameter			
>	class std::basic_ostream<char,struct std::char_traits<char> > & out	EXTERN	EXTERN
>	class Point & point	IN	IN

Define a variable to be passed as parameter to the test object in the **Definitions** of the user code.

In the **Test Object Prolog** you can bind the variable to the parameter using a macro

```
TESSY_BIND_PARAMETER_parameter_name
```

The macro will be listed in the text proposals (by pressing Ctrl+Space).



Test Case or **Test Step Prolog** can be used to call methods or set members of the parameter object to have it in a state fitting the needs for the current test case. Also different variables may be defined for different test cases or test steps and bind to the parameter in the respective test case or test step prolog.

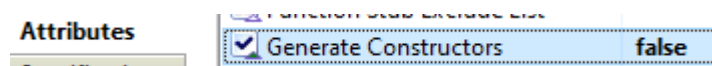
If the parameter is changed by the test object it can be evaluated in the test step epilg using evaluation macros.

Refer to module "Parameter Proxy" in the C++ example project.

3.4 Attribute ‘Generate Constructors’

Normally, the object needed to test a method is created by the constructor selected in the TDE for the “**this**” variable. But in some cases TESSY is not able to generate correct code to create the test objects instance or using the TDE to setup the parameters would be too much effort. In this case the object can be created in the user code.

Set the attribute **Generate Constructors** to false for the module (or all test objects of the affected class).

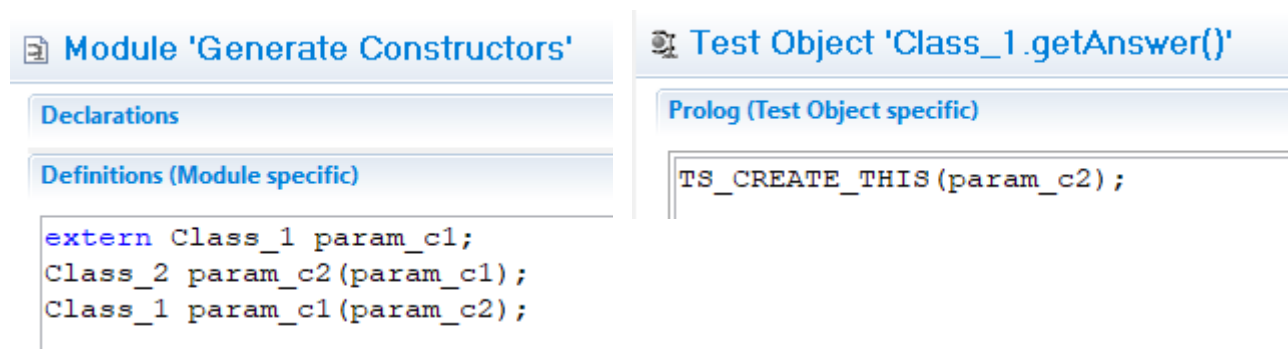


Define global variables that are needed as constructor parameters in the **Definitions** of the user code.

In the **Test Object Prolog**, use the predefined macro

```
TS_CREATE_THIS
```

and pass the defined variables needed for the constructor. You may also use the macro in **Test Case** or **Test Step Prolog** if a new object is needed for the method under test.



Refer to module called “Generate Constructors” in the C++ example project.

3.5 Attribute ‘Enable Generate Default Constructors’

When this attribute is set to true TESSY generates a default constructor to all classes that does not have one, to make it possible to declare extern global class variables. However, these default constructors may lead to compiler errors, e.g. when classes need to initialize constant members in the member initialization list.

If this is the case, this attribute must be set to false and extern global class variables that cannot be defined using a default constructor must be set to **Don’t define Variable** in TIE and must be defined in the user code **Definitions** of the module using a proper constructor.