



# PalantirCASH Module Writing Guide

**Prepared by Palantir**

Confidential

**Disclaimer****Copyright**

The information in this document is subject to change without notice. The software described in this document is furnished under a license agreement. This software may be used or copied only in accordance with the terms of such agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement. No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Palantir Economic Solutions Ltd.

Copyright © 2010 Palantir Economic Solutions Ltd.

**Registered Trademarks**

All names and trademarks are the property of their respective owners.

## Table of Contents

<b>Prerequisites .....</b>	<b>4</b>
<b>Chapter 1. Introduction .....</b>	<b>5</b>
<b>Chapter 2. The VSTA Module Editor .....</b>	<b>6</b>
2.1 Getting Started.....	6
2.2 The Integrated Development Environment .....	8
<b>Chapter 3. PRL Module Code.....</b>	<b>12</b>
3.1 Partial Classes .....	12
3.2 The Module Editor Window .....	12
3.3 The Generated Module Code .....	18
3.4 The Designer File .....	20
3.5 The Implementation File .....	22
<b>Chapter 4. PRL Programming .....</b>	<b>25</b>
4.1 Module Programming.....	25
4.2 Subroutine Arguments and Local Qualifiers .....	26
4.3 Adding Functions.....	28
4.4 Calling Functions .....	28
4.5 Variables.....	29
4.6 Data Types .....	29
4.7 IntelliSense .....	32
4.8 Calculation Phases.....	34
4.9 Periodicity.....	35
4.10 Partners and Working Interests .....	36
4.11 Subroutine Programming.....	37
4.12 Function Programming .....	37
4.13 Building, Validating and Applying Projects .....	38
<b>Chapter 5. Testing and Debugging .....</b>	<b>41</b>
5.1 Attaching the Debugger .....	41
5.2 Debugging .....	41
<b>Chapter 6. Code Examples .....</b>	<b>44</b>
6.1 Declaring Calculation Periods and Periodicity Loops .....	44
6.2 First Calculated Period Check .....	44
6.3 TimeSeries Object Declaration.....	45

<b>Chapter 7. Accessible Framework Attributes.....</b>	<b>46</b>
7.1 Me.CalcParm .....	46
7.2 Me.CalculationCurrency .....	46
7.3 Me.CalculationCurrencyScenario .....	46
7.4 Me.CalculationProjects .....	47
7.5 Me.CurrentPriceScenarioDefinition .....	47
7.6 Me.CurrentScenario .....	47
7.7 Me.CurrentScenarioWeighting .....	48
7.8 Me.Price.....	48
7.9 Me.PriceName .....	49
7.10 Me.ThisCalculation .....	49
7.11 Me.ThisEngine .....	49

# Prerequisites

This document assumes a basic understanding of Visual Studio, Visual Basic and PalantirCASH. Users with previous programming experience should be able to use this document; however, some terminology and concepts will be new.

Palantir provides courses on CASH Regime design and module writing which can be tailored to your requirements.

This document is supported by the PRL Module Guide which describes and defines the standard modules in the PRL.

# Chapter 1. Introduction

The Palantir Regime Library (PRL) modules are the core fiscal calculation components for each Regime. Each box on the Regime Designer canvas represents a single module behind which underlying code defines the functionality of the module.

The **Modules** pane contains a list of all available modules. It is not possible to add, edit or delete modules here; these tasks can be accomplished using the VSTA module editor described in Chapter 2 "The VSTA Module Editor".



Each module is written in Visual Basic and is stored in the database as both compiled and uncompiled code. The compiled code is later executed at runtime to carry out the fiscal calculations.

**Note:** Non-country-specific modules are generally considered core Palantir modules and it is highly recommended that these modules are not edited due to the potential impact on the existing Regimes that use them to perform fiscal calculations.

This guide is intended as a reference for users reviewing existing module code to gain a better understanding of the terms and concepts discussed.

## Chapter 2. The VSTA Module Editor

VSTA (Visual Studio Tools for Applications) is a tool allowing direct access to the underlying code behind each module. Coupled with PalantirCASH, VSTA provides a powerful customized toolset for users to view, edit and build modules.

The main advantages of using VSTA over a standalone module editor are that VSTA:

- Gives users the ability to view and organize their module classes by files (in the Project Explorer) or by class (in the Class View).
- Provides IntelliSense, Microsoft's implementation of auto completion, which provides a quick and convenient way to access descriptions of variables and functions, particularly their parameter lists.
- Allows for design-time syntax checking allowing users to build and validate their changes before they can be applied to the database and used within the Regimes.
- Provides a runtime debugging functionality allowing users to test and debug module code.

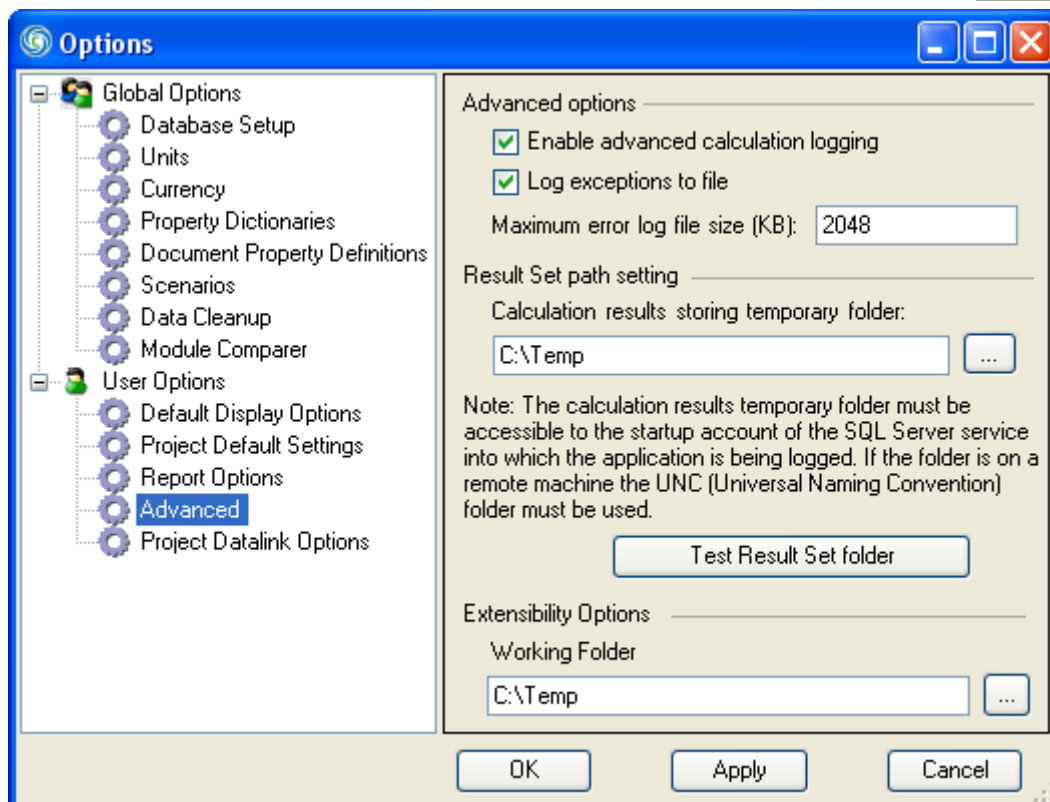
### 2.1 Getting Started

With an installed and licensed version of CASH running on your machine, you can launch the VSTA Module Editor by either going to **Tools > Edit Modules** in the CASH menu or by pressing F6.

**Note:** Access to the VSTA Module Editor can be restricted. In this instance a user cannot see the **Edit Modules** option in the **Tools** menu. If you need access, contact the CASH administrator who will be able to change your permission settings appropriately.

While the VSTA application initialises, a temporary set of modules is created and saved in a specified working folder. Any module changes will be saved in this folder until committed to the database. The module code in the working folder will be overwritten with the code from the database each time VSTA is restarted.

**Note:** It is not possible to commit any changes to the database until **all** modules have been successfully built and validated by the VSTA compiler.



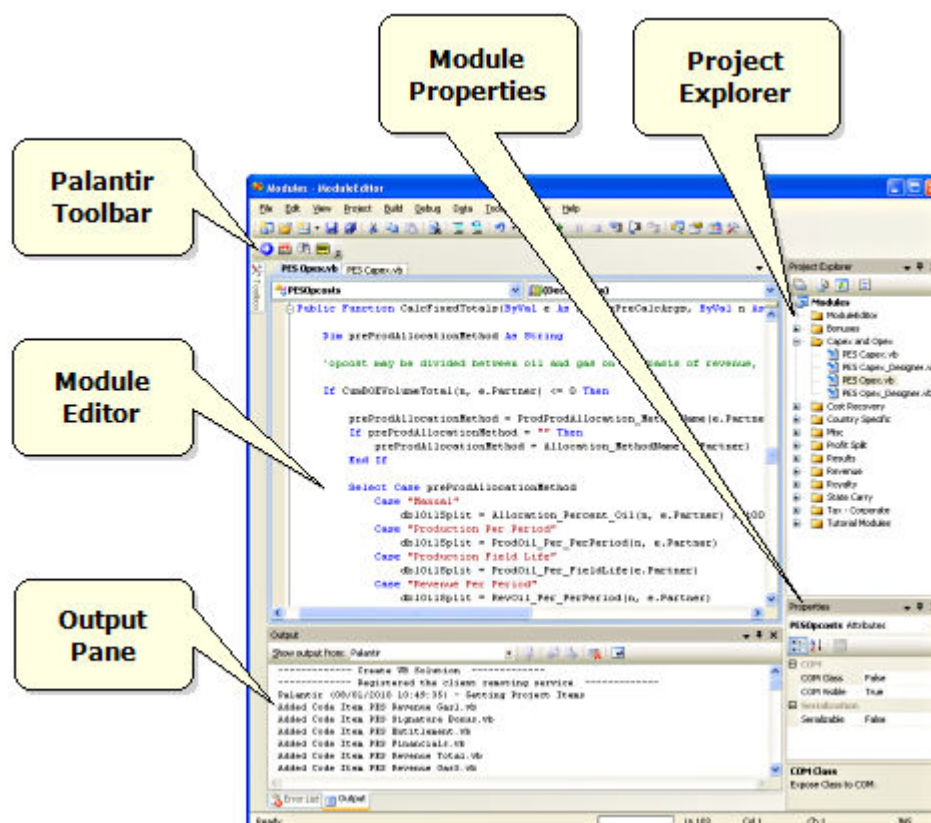
The working folder is set using the **Options** window. To specify the path to this folder, go to **Tools > Options** and then select **User Options > Advanced > Extensibility Options**. By default, the working folder is set to the user's Windows Temp directory. It is highly recommended that a new directory with a short path name be selected, for example *C:\Temp*.



## 2.2 The Integrated Development Environment





Users with previous coding experience, in particular with Visual Studio, will find the VSTA Module Editor very familiar. The VSTA IDE (Integrated Development Environment) has a similar look and feel to the standard Visual Studio IDE.

The IDE contains several elements, each of which can be docked, auto-hidden or displayed as required. The main components of the IDE are shown below.



### 2.2.1 Palantir Toolbar

The Palantir toolbar provides the core functionality allowing for changes to code, files and the file structure to be compiled, validated and applied to the database.

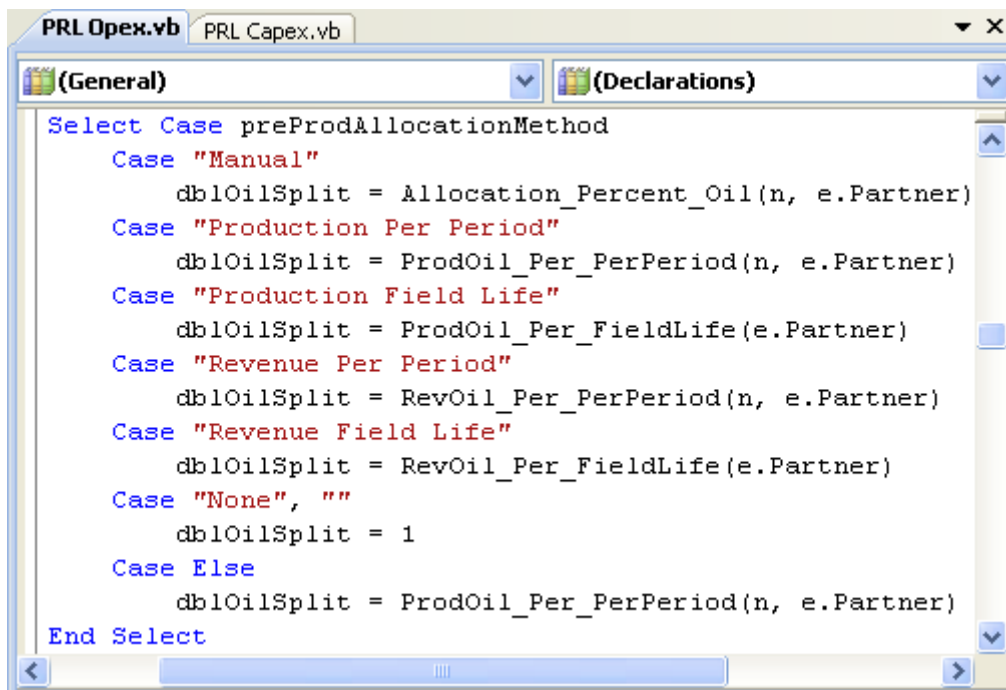
Icon	Name	Description
	<b>Attach Debugger</b>	Attaches the debugger to the active Excel project allowing for runtime debugging of the modules.
	<b>Build and Validate</b>	Performs a standard compilation without writing modules back to the database.
	<b>Apply Changes</b>	Performs a compilation and then, if successful, writes the modules into the database.
	<b>Module Editor</b>	Opens the <b>Module Editor</b> window where you can add, delete and modify modules.


For more information about how the toolbar options are used, see Chapter 4 "PRL Programming" and Chapter 6 "Code Examples".

## 2.2.2 Code Editor

The Code Editor (opened in the Module Editor Space) is the word processor for the Integrated Development Environment (IDE). For the VB.Net language in which we are developing, the editor offers features such as:

- Access to object properties and methods at design time.
- IntelliSense statement completion which makes language references easily accessible and allows users to perform searches through language elements.
- Collapsible code sections.
- Code definition windows that display the source code of an object.
- Auto alignment to ensure the code is correctly formatted and easy to read.



There are several ways to open the code for editing; the most common is to select a module in the Project Explorer and click on  on the toolbar. Or, double-click on a module to open it.

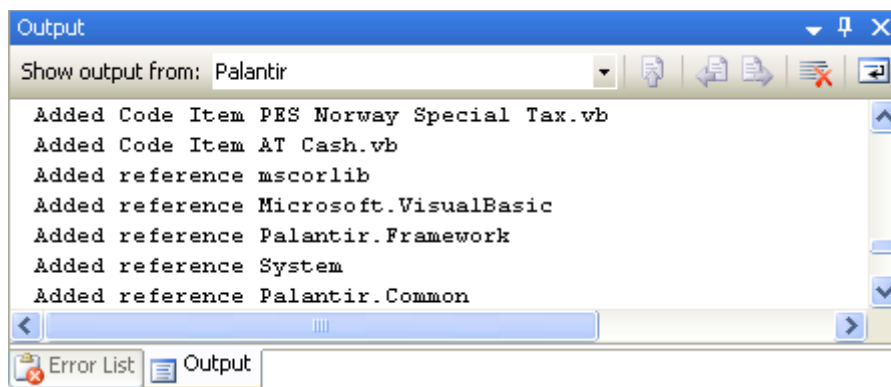
As with most editors, the IDE allows users to open multiple files simultaneously. This gives users the option to view, edit, and copy/paste between modules as required. Switching between the modules is also straightforward: simply select the tab corresponding to the module you want to access.

## 2.2.3 Output Pane

The output pane (also known as the **Output** window) is located by default at the bottom of the IDE and can be easily moved anywhere on the screen. The pane displays status messages for a number of features.

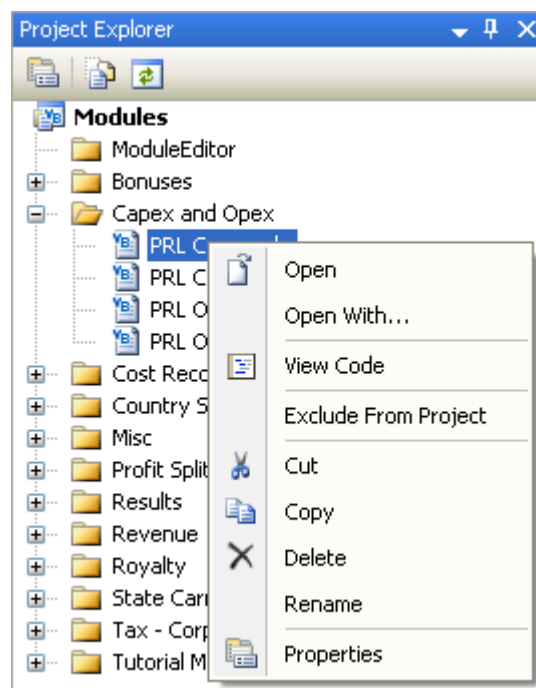
In this context, the pane commonly shows initialisation, validation, build and database application status messages. Programming errors are generally listed in the **Error** list (see Section 4.13 "Building, Validating and Applying Projects").

If the **Output** pane is not displayed in your IDE, go to the **View** menu and check the **Output** option.



## 2.2.4 Project Explorer

The Project Explorer (also known as the Solution Explorer) is used to organize module files and directly access commonly used commands associated with an item from the tree view.



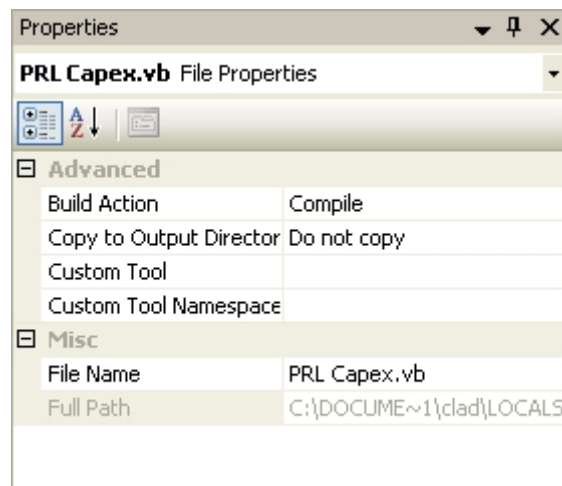
The Module Suite is displayed as a tree of VB files. The explorer allows you to open a VB file for modification and to perform other standard management tasks by right-clicking and selecting options from the context menu.

If the Project Explorer is not displayed in your IDE, go to the **View** menu and check the **Project Explorer** option.

## 2.2.5 Module Properties

The module properties pane is used to indicate what actions are to be performed on a selected file. By default, all module files have their **Build Action** field set to *Compile*, with the alternative being to embed the file into the build as a resource.

This dialogue is also used to determine the file location or rename the file.



## Chapter 3. PRL Module Code

Each module within the PRL is broken down into two distinct components:

1. The Visual Basic implementation file which defines the functionality of the module;
2. The Visual Basic designer file which contains class and variable declarations.

The implementation code and designer files work together to define the functionality of the module. Both files are linked via the class declaration in the designer file and the reference to the class in the source file. These partial classes are therefore merged to build the class from which objects are created.


### 3.1 Partial Classes

As mentioned previously, each module within the PRL is broken into two distinct components - the implementation file and the designer file - which are merged to form the complete class definition.

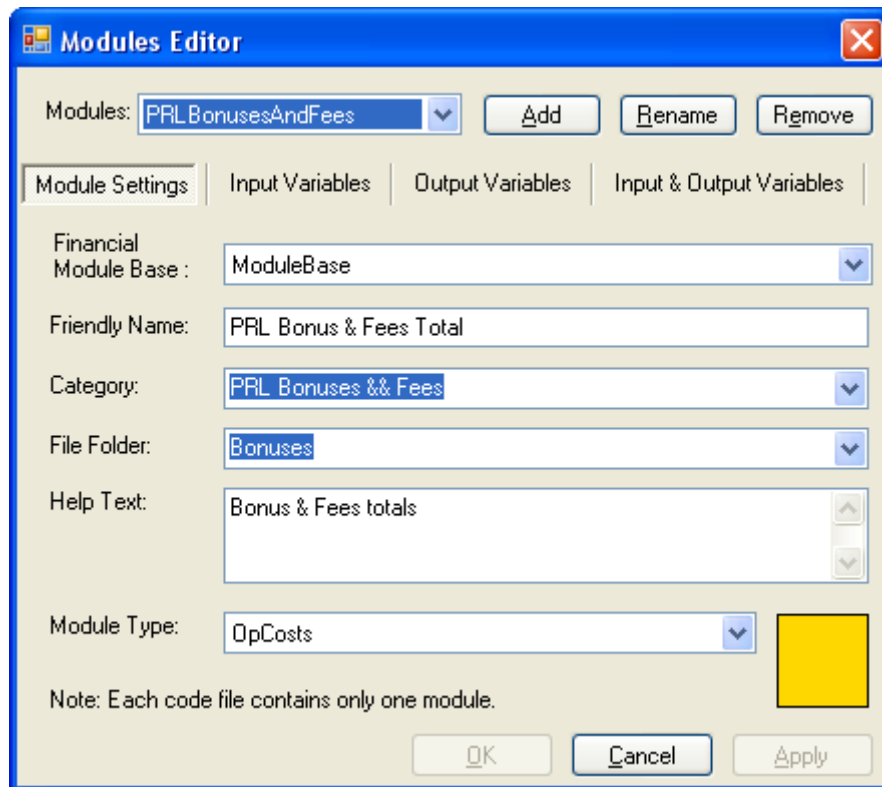
The concept of partial classes allows a class definition to be split over two or more source files. Each source file contains a section of the class definition with all parts being combined when the module is compiled.

In the PRL Module Suite, the designer files contain the class and variable definitions which should only be generated and updated by the Module Editor. This means that the designer files should be considered READ ONLY, and the module implementation file can be used to model the fiscal calculations.

### 3.2 The Module Editor Window

The Module Editor provides basic functions to manage the modules. To open it, click on  on the Palantir toolbar and the **Modules Editor** window will be opened (shown below). Here you can:

- Add new modules to the Module Suite and define their properties;
- View and edit existing modules;
- Delete existing modules from the Module Suite.



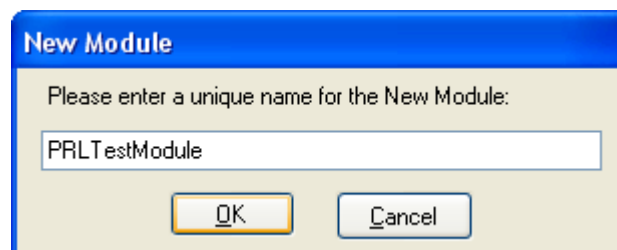
### 3.2.1 Adding, Renaming and Removing Modules

The Module Editor allows you to carry out basic module operations by using the **Modules** menu and buttons.

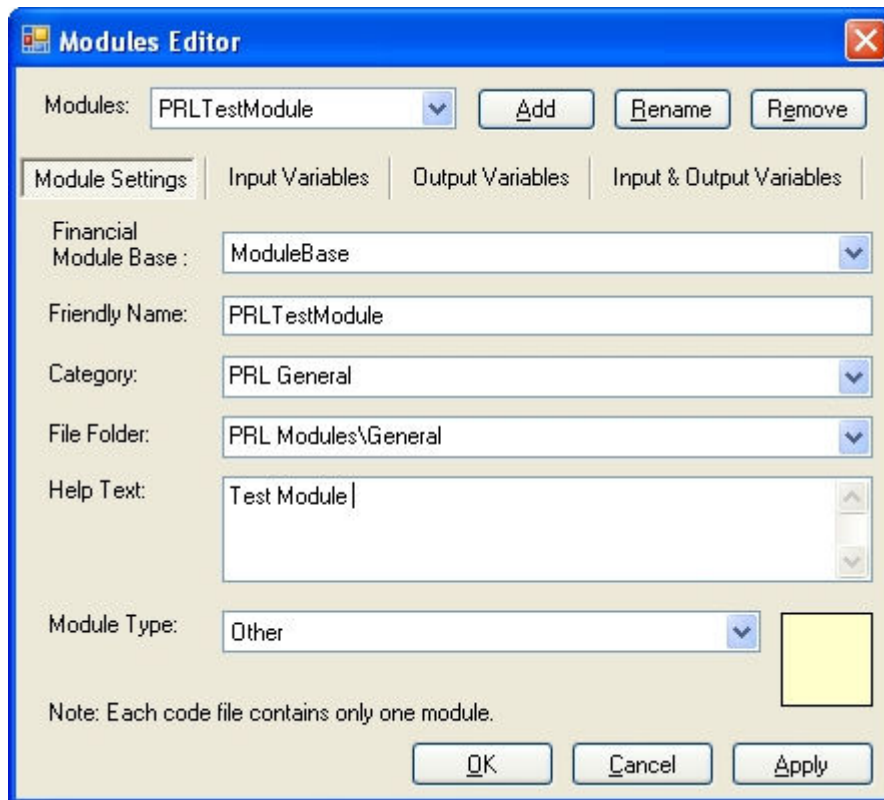


#### Adding a Module

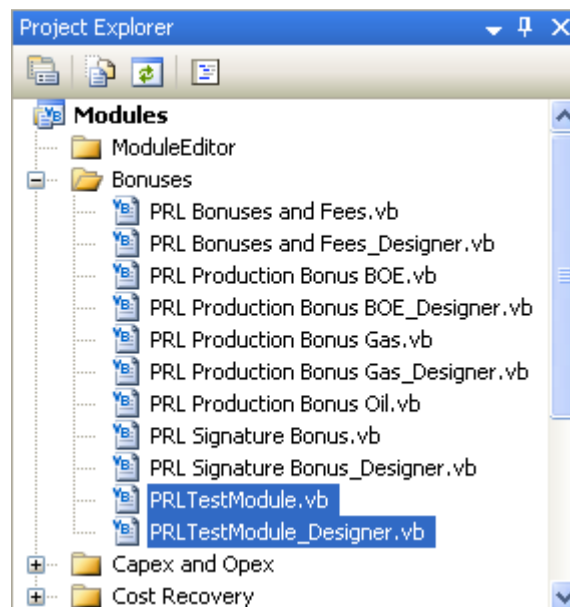
To add a new module, click on the **Add** button. In the pop-up window, enter a valid name for the new module as shown below. The name must be alphanumeric and must not contain spaces or special characters.



Once a name has been assigned, you can customize the module in the **Modules Editor** window, setting the **Financial Module Base**, **Friendly Name**, **Category**, **File Folder**, **Help Text** and **Module Type** as shown below.



Click **Apply** or **OK** to add the new module to the hierarchy and to display the module and designer files in the Project Explorer.



## Renaming a Module

Selected the module you want to rename from the **Modules** drop-down list and click on the **Rename** button. In the pop-up window, specify a new module name. This name must be alphanumeric and must not contain spaces or special characters.



Click on the **Apply** or **OK** button to save the changes in VSTA.

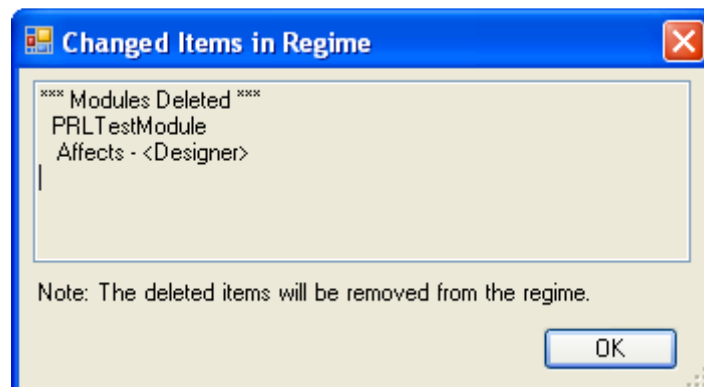
## Removing a Module

Selected the module you want to remove from the **Modules** drop-down list and click on the **Remove** button. In the pop-up window, you can confirm or cancel deletion. The module will be deleted from the Module Suite but this change will only be visible in VSTA once you click on the **Apply** or **OK** button in the **Modules Editor** window.

**Note:** Removing a module from the Module Suite can have a profound effect if the deleted module is used by one or more Regimes.

If you want to delete a module which is no longer needed, first check whether it is used in any Regimes. If it is used, delete it from Regimes and re-link all variables. After that, open VSTA and delete the module from the PRL Module Suite.

If you delete a module in VSTA without first removing it from Regimes, all Regimes that use that module will stop functioning and a warning will be displayed when you try to open a broken Regime.

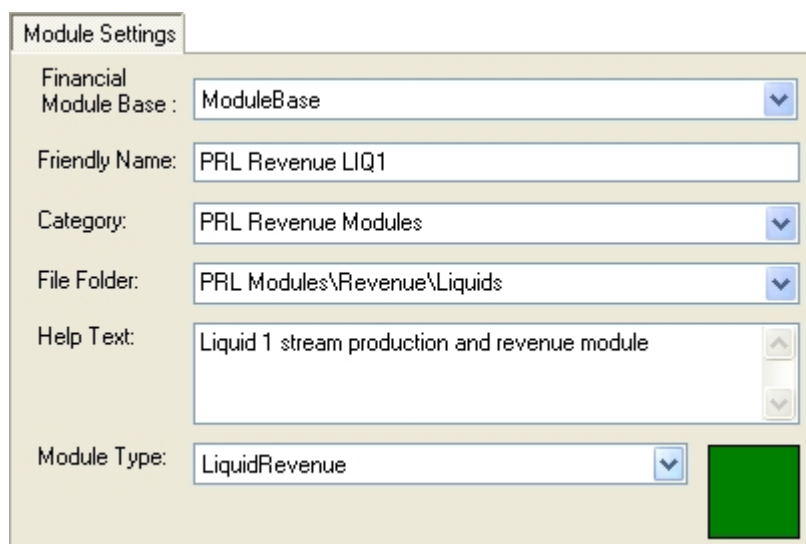


**Note:** All changes made in the Module Editor will not be ported to VSTA until you click on the **Apply** or **OK** button. Changes ported to VSTA will not be saved in the database until the Module Suite has been successfully built (see Section 4.13 "Building, Validating and Applying Projects").



### 3.2.2 Module Settings

The **Module Settings** section determines the standard settings for a module.



Field	Description
<b>Financial Module Base</b>	This list specifies which module class the module inherits from. By default it contains two options: <i>Financial ModuleBase</i> for working with FINANCIALS modules and <i>ModuleBase</i> for working with CASH modules.
<b>Friendly Name</b>	Module name displayed on the module pane (Regime Designer). This can be any alphanumeric string containing spaces or special characters.
<b>Category</b>	Module pane (Regime Designer) group within which the module will be displayed.
<b>File Folder</b>	Relative location of the module within the Module Suite.
<b>Help Text</b>	Tooltip displayed in the Regime Designer when the mouse hovers over a module.
<b>Module Type</b>	Module type determines the module colour in the Regime Designer.

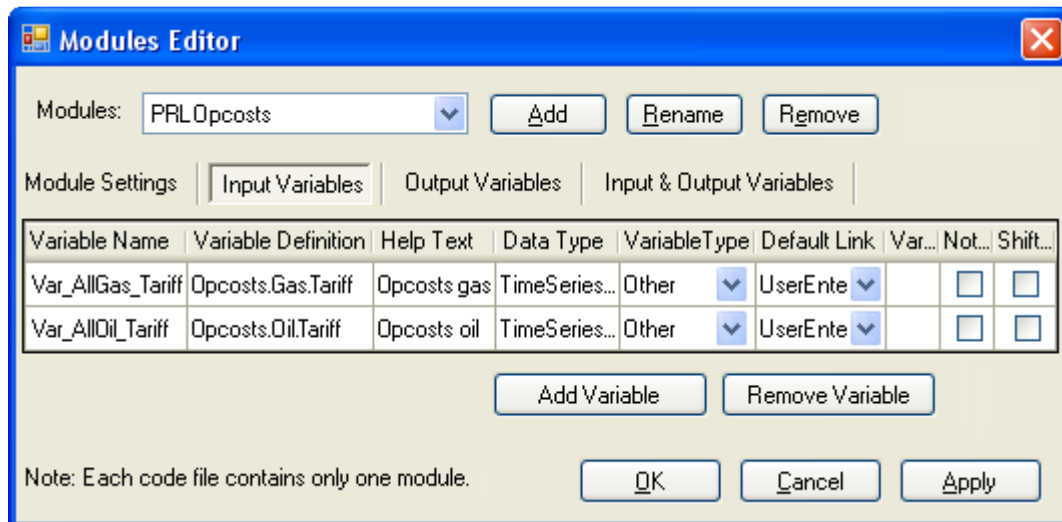
### 3.2.3 Variable Selection

A fundamental component of the Module Editor is the functionality allowing you to select input, output and input/output variables.

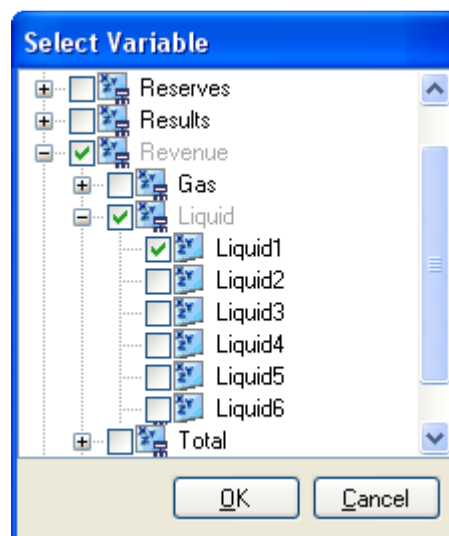
An input variable acts as a source variable for a module, an output variable acts as a result variable for a module, and an input/output variable can act as both source and result variables for a module. Input/output variables are used, for example, when a module takes an input and modifies it for use by other modules.

## Adding a Variable

To add a variable to a module, first select the variable type by switching to the **Input Variables**, **Output Variables** or **Input & Output Variables** tab.



Then, click on the **Add Variable** button and the **Select Variable** dialogue will be displayed.



The **Select Variable** dialogue contains a complete list of variables that can be added to the module. Select as many variables as required and click **OK**. The variables will appear in the Module Editor.

Once the variables have been added to the Module Editor, you can change their settings using the available options.

Field	Description
<b>Variable Name</b>	Name that will be used as an internal module variable name. The name should describe the variable but be short.
<b>Variable Definition</b>	Location of the variable within the variable list. This field is populated automatically and should not be edited.
<b>Help Text</b>	Description of the variable's purpose.
<b>Data Type</b>	Variable data type, for example <i>ScalarString</i> or <i>TimeSeriesDouble</i> . This field is populated automatically from the variable definition.
<b>Variable Type</b>	Determines whether the variable should be made available on the <b>Capital</b> tab of the Regime. Other means the variable will not be displayed on the <b>Capital</b> tab.
<b>Default Link</b>	Determines the default source for the module variable; however this setting can be changed within the Regime Designer. Options include <i>User Entered</i> , <i>Link</i> , <i>Contract Term</i> and <i>None</i> .
<b>Variable To Link</b>	If the <b>Link</b> option is selected in the <b>Default Link</b> column, it is possible to select a variable from another module to link to/from. This functionality can reduce work when building Regimes.
<b>Not Truncate</b>	When the economic limit is determined, the data in this variable will not be truncated after this point if this flag is set.
<b>Shift To EL</b>	This flag is required when abandonment capital variables are to be shifted to the economic limit. If this box is checked, all values within the selected <i>TimeSeriesDouble</i> variable will be shifted to the calculated economic limit.

## Removing a Variable

To remove a variable, select it on the appropriate tab and click on **Remove Variable**.

## 3.3 The Generated Module Code

Although a new module may have been added and configured in the Module Editor, it will only be added to the Module Suite once the **Apply** or **OK** button has been clicked. When the Module Editor is closed, the new module will be available for viewing via VSTA.

The generated module code as specified previously is broken down into two files: the designer file and implementation file. Both files have many common features; the most significant ones are described below.

### 3.3.1 Copyright Information

---

All implementation and designer files in the Module Suite have comment lines at the top, giving standard copyright information. Note that the standard for VSTA is to display comments in green.

```
'COPYRIGHT 2002-2010 PALANTIR ECONOMIC SOLUTIONS LTD. ALL RIGHTS RESERVED.  
'THIS CLASS DECLARATION CONTAINS AUTO GENERATED CODE.  
'MODIFYING THIS DECLARATION OUTSIDE THE MODULE WIZARD MAY CAUSE THE  
'MODULE WIZARD TO STOP WORKING.
```

### 3.3.2 Import Statements

---

All implementation and designer files in the Module Suite by default have a number of `Imports` statements at the top of each file. These statements import/load programming elements defined as the Palantir Framework into each module. These elements allow the module to function as part of CASH at runtime.

```
Imports Palantir.Common  
Imports Palantir.Framework  
Imports Palantir.Framework.Extensibility
```

### 3.3.3 Region Statements

---

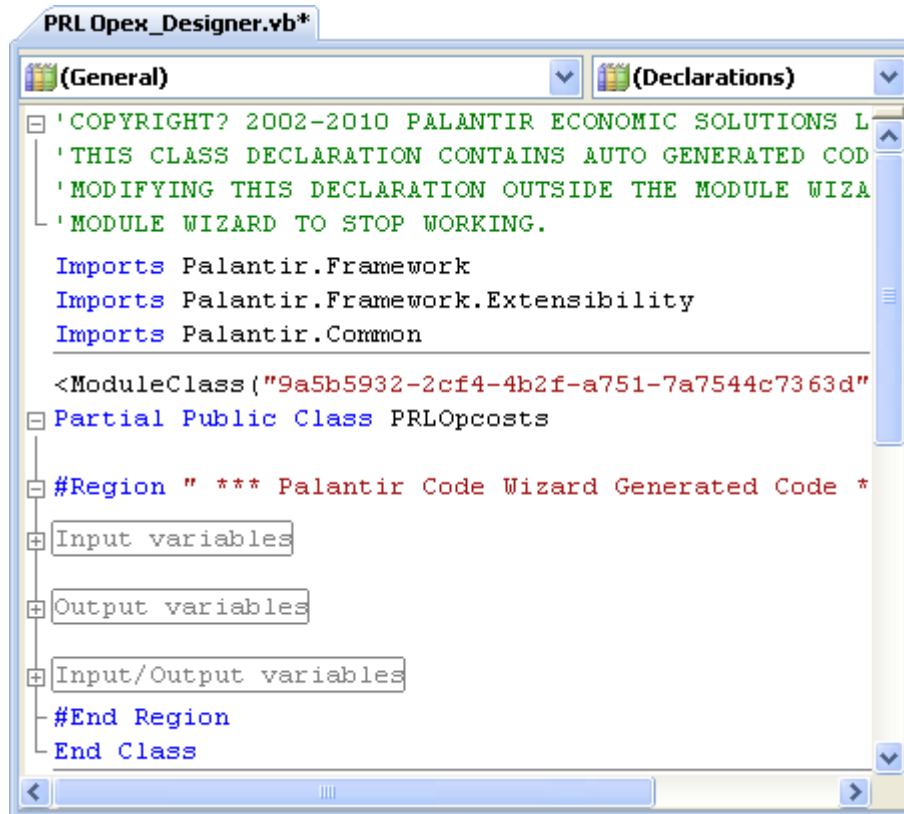
Region statements allow you to define blocks of code that can be expanded or collapsed using the outlining feature of the Editor. Clicking on the plus sign expands a region, while clicking on the minus sign collapses a region.

```
+ *** Palantir Code Wizard Generated Code ***  
- #Region " *** Palantir Code Wizard Generated Code *** "  
  ' Expanded Region  
- #End Region
```

## 3.4 The Designer File

It is highly recommended that the designer file should be considered a READ ONLY file. All changes to it should be made in the Module Editor.

When you open this file, the automatically generated code will look as shown below.



### 3.4.1 Class Definition

The class definition is based on the settings entered in the Module Editor when a new module is created. The first argument of the class definition is known as a GUID (Globally Unique Identifier).

```

<ModuleClass("9a5b5932-2cf4-4b2f-a751-7a7544c7363d", "OpCosts",
            "Opex", "PRL Opex.vb", "009")>
Partial Public Class PRLopcosts

```

The remaining arguments mirror the settings in the Module Editor.

Argument	Description
<b>GUID</b>	Globally Unique Identifier.
<b>Friendly Name</b>	Module name displayed on the Regime Designer.
<b>Category</b>	Group within which the module will be displayed in the <b>Modules</b> pane.
<b>Help Text</b>	Text displayed when hovering over a module in the Regime Designer.
<b>File Name</b>	Name for the module's implementation file.
<b>Module Type</b>	Code determining the module colour on the Regime Designer.

The class definition is finalized by specifying the class name and confirming that the class is a partial class as shown in the example above.

### 3.4.2 Variable Definitions

The regions marked *Input Variables*, *Output Variables* and *Input/Output Variables* contain the module variable definitions as specified in the Module Editor.

```
<ModuleVariable("Production.Liquid1.Rate", "", VariableDirection.Input),
    DefaultVariableLink(VariableLinkAction.UserEntered)>
Public Property LiquidRate() As TimeSeriesDouble
```

The arguments of the module variable definitions are ordered and defined as follows:

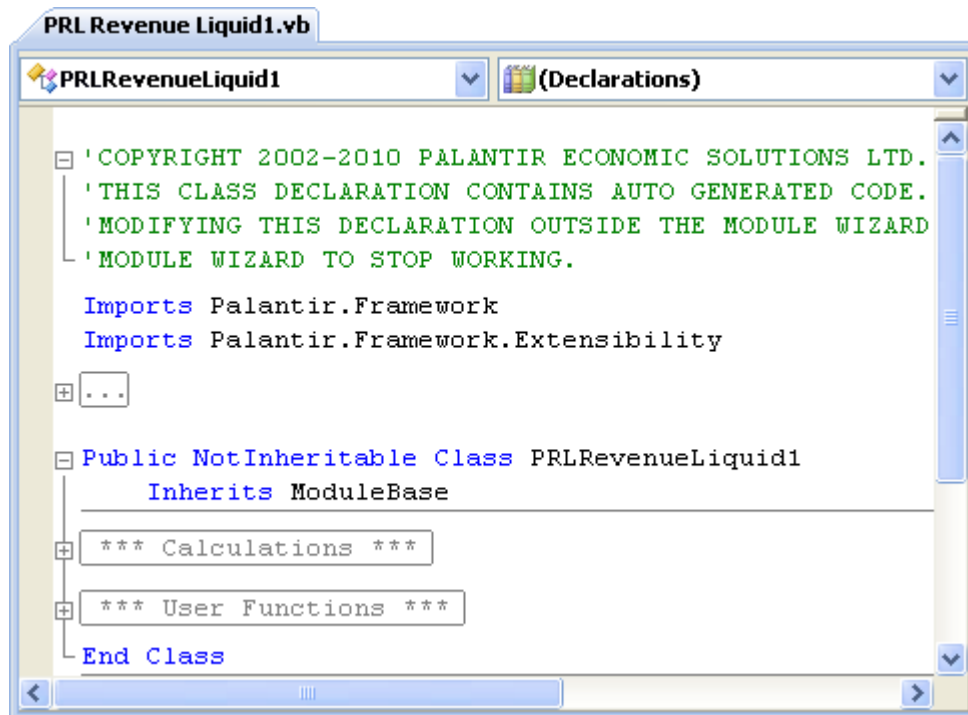
Argument	Description
<b>Variable Definition Name</b>	Location of the variable within the variable list.
<b>Help Text</b>	User-friendly description of the module.
<b>Variable Direction</b>	Direction of the data flow: <i>Input</i> for input variables, <i>Output</i> for output variable or <i>InputOutput</i> for input/output variable).
<b>Default Link</b>	Default source for the module variable.

In the example above, `LiquidRate` is the variable name (the name that is used as an internal module variable name) and `TimeSeriesDouble` is the data type (for more information, see Section 4.6 "Data Types").

## 3.5 The Implementation File

The implementation file is generated to work together with the designer file and allows you to model the fiscal calculations as per design.

When you open the file, the automatically generated code will look as shown below.



The main body of the partial class is enclosed between the `Public Partial Class` and `End Class` statements. The class name specification ensures that all partial classes are bound together to define the entire class.

### 3.5.1 Inheritance

Following the class declaration, the next line clarifies the inheritance settings for the class.

```
Inherits ModuleBase
```

All modules within the Module Suite inherit from the **ModuleBase** class. This setting allows the new class, known as a derived class, to inherit attributes and behaviour of the base class. This helps to reduce code repetition.

### 3.5.2 Calculations

This section describes four stages of the calculation sequence from which module functions may be called. Within the implementation file, these stages (also known as phases) are defined by subroutines.

The subroutines are called in order during project calculation and are named:

1. PreCalc;
2. BeforeEconomicLimit;
3. AfterEconomicLimit;
4. PostCalc;

```
Public Overrides Sub PreCalc(ByVal e As ModulePreCalcArgs)
    MyBase.PreCalc(e)
End Sub

Public Overrides Sub CalcPeriodBeforeEconLimitFound(ByVal
    e As ModuleCalcBeforeLimitArgs, ByVal n As CalcPeriod)
    MyBase.CalcPeriodBeforeEconLimitFound(e, n)
    Call CashFlow(e.Partner, n, "Before")
End Sub

Public Overrides Sub CalcPeriodAfterEconLimitFound(ByVal
    e As ModuleCalcAfterLimitArgs, ByVal n As CalcPeriod)
    MyBase.CalcPeriodAfterEconLimitFound(e, n)
    Call CashFlow(e.Partner, n, "")
End Sub

Public Overrides Sub PostCalc(ByVal e As ModulePostCalcArgs)
    MyBase.PostCalc(e)
End Sub
```

**Note:** A subroutine may be prefixed by the **<DoNotCalcMethod(>** statement. This statement is commonly used to confirm that the specified calculation stage can be ignored as no module functions will be called during this phase. Making use of this statement may significantly speed up the project calculation; however, it is not considered a mandatory requirement.

```
<DoNotCalcMethod(> _
Public Overrides Sub CalcPeriodBeforeEconLimitFound(ByVal
    e As ModuleCalcBeforeLimitArgs, ByVal n As CalcPeriod)
    MyBase.CalcPeriodBeforeEconLimitFound(e, n)
End Sub
```



By default, the four subroutines will be empty upon generation with the developer having the scope to populate and customize as required.

For more information about the calculation phases, see Section 4.8 "Calculation Phases".

### 3.5.3 User Functions

---

All implementation files have an allocated area for user functions. In a newly generated module this area will by default be empty.

```
#Region " *** User Functions *** "  
  
#End Region
```

The functions added to this region give you the ability to model the fiscal calculations. For more information, see Chapter 4 "PRL Programming".

## Chapter 4. PRL Programming

Creating and editing of PRL modules requires basic understanding of Visual Studio and Visual Basic, and a certain level of programming experience. We assume that the person attempting to edit modules has the necessary knowledge and skills.

The following chapters cover the fundamental topics required to customize modules using Visual Basic to carry out fiscal calculations.

### 4.1 Module Programming

Once generated and populated with the required input, output and input/output variables, the implementation file can be tailored to model the core fiscal calculations.

To any user working only with Regimes and projects in CASH, the modules are a series of colour-coded rectangles positioned in a defined sequence. Each module has a set of defined inputs which are fed into the module and used to generate a set of outputs via a sequence of calculations.

The calculations used to generate the outputs (results) are defined within the **User Functions** region of the implementation file. The number of functions defined within this region is determined by the module design and the calculation phase when the function is to be called.

Although it is possible to execute all the required module calculations within one function, good programming practice dictates that to make the code easier to read and maintain, code should be divided into discrete functions which can be called from within another function or by one of the phase subroutines (*PreCalc*, *BeforeEconomicLimit*, *AfterEconomicLimit* and *PostCalc*).

Calling functions from specific phase subroutines also allows the programmer to dictate when specific calculations are to be executed in the calculation sequence. It should be emphasised that calling functions from the correct phase subroutine ensures that calculations are executed at the correct point of the calculation sequence.

It is possible to call a function from within another function and in many instances a function can be called recursively until a required result is obtained. The number of functions and the point at which they are called depends on the module design and implementation of the Regime.

Well-designed, structured and methodical code can drastically reduce the calculation time of a project. Other factors that affect the calculation time are the number of modules in a Regime, unnecessary function calls and efficiency of the code behind each module.

## 4.2 Subroutine Arguments and Local Qualifiers

Each phase subroutine is qualified with phase-specific arguments giving the developer access to a number of calculation settings based on the position in the calculation sequence, the partner being referenced and calculation period specified.

The phase subroutines are discussed later in this chapter. This section covers the common module calculation arguments for each subroutine.

Each of the four phase subroutines accessible via the calculations region is qualified with a module calculation argument specific to the subroutine. Each argument is generally referenced using the qualifier `e` which can be expanded to access calculation-specific settings. The argument will be declared as one of the following:

- `ModulePreCalcArgs` (Pre-Calculation Phase);
- `ModuleCalcBeforeLimitArgs` (Before Economic Limit Phase);
- `ModuleCalcAfterLimitArgs` (After Economic Limit Phase);
- `ModulePostCalcArgs` (Post Calculation Phase).

There are a number of common calculation-specific settings accessible via the `e` qualifier. The common attributes are described in the table below.

Qualifier	Description
<code>e.CalculationCurrency</code>	Returns a 'Currency' object representing the current calculation currency as specified in the project.
<code>e.CalculationCurrencyScenario</code>	Returns a 'CurrencyScenario' object representing the current calculation scenario as specified in the project.
<code>e.CarryInterestShares</code>	Returns a 'CarryInterestShareCollection' object representing the working interest shares for the different partners in the project.
<code>e.EconomicLimit</code>	Returns a 'YearMonth' object representing the year and month of the economic limit. This value can change over calculation sequence.
<code>e.HasEconomicLimitFound</code>	Returns a Boolean confirming whether the economic limit has been determined; if not, value is set to <i>null</i> .
<code>e.OperatingPartner</code>	Returns an 'IWorkingInterestPartner' object which determines the current partner set as the operating partner in the project.
<code>e.Partner</code>	Returns an 'IWorkingInterestPartner' object which determines the current partner being calculated. Returns <i>Nothing</i> for gross calculations.

Qualifier	Description
e.Partners	Returns an 'IWorkingInterestPartner' object which represents all the partners that exist in the project being calculated. This property is set to <i>null</i> when the current calculation is a ringfence or incremental calculation.
e.Product	Returns a 'ProductCategory' object representing the product the project is calculating for.
e.ProductStream	Returns an Integer representing the number of the stream the project is calculating for.
e.TimeframeInfo	Returns a 'CalculationTimeframe' object which represents the timeframe that the current project is calculated over.
e.WIShares	Returns a 'WorkingInterestShareCollection' object representing the working interest shares of all the defined partners in the project.

The second argument referenced in the **Before** and **After Economic Limit** phases is the **Calculation Period** argument. The argument is referenced using the qualifier *n* and can be expanded to determine the current calculation period and project settings. The table below describes the common attributes of this object.

Qualifier	Description
n.MonthsInPeriod	Returns an Integer representing the number of months in a period. This value is based on the periodicity setting.
n.Period	Returns an Integer representing the current period within the current year. This value ranges from 1 to 12 and is based on the periodicity settings: <i>Monthly</i> (1-12), <i>Quarterly</i> (1-4), <i>Semi-Annually</i> (1 or 2) and <i>Annually</i> (1).
n.Periodicity	Returns a 'TimeScale' object representing the calculation setting (number of periods) over which projects are calculated. The Regime periodicity is set either in the Regime itself or in the project document, and can be set as either <i>Monthly</i> , <i>Quarterly</i> , <i>Semi-Annually</i> or <i>Annually</i> .
n.PeriodsInYear	Returns an Integer representing the number of periods in the year given the periodicity of the object.
n.ToDateTime	Returns a 'DateTime' object based on the current year and period settings.
n.Year	Returns an Integer representing the current year the calculation period refers to.

## 4.3 Adding Functions

All newly generated implementation files start with an empty **User Functions** region. In this area you can add variable and functions declarations to customize the file and generate a set of outputs from the given inputs.

```
#Region " *** User Functions *** "  
  
    Private Function TestFunction(ByVal e As ModulePreCalcArgs) As Boolean  
        ' Add function code here!  
  
    End Function  
  
#End Region
```

In the example above, a new function has been added to the **User Functions** region. The empty function declaration has two arguments: *e* and *n*. The first argument *e* contains the arguments associated with the **Before Economic Limit** phase. The second argument *n* defines the exact time period at which the function is called, and is referenced by variables within the function. The importance of choosing the correct phase and further details about time periods will be covered later in this document.

## 4.4 Calling Functions

For the contents of any function to be executed, the function needs to be called from within another function or an appropriate phase subroutine. Function calls require appropriate arguments to be qualified in the function header.

```
Public Overrides Sub CalcPeriodBeforeEconLimitFound(ByVal e  
    As ModuleCalcBeforeLimitArgs, ByVal n As CalcPeriod)  
    MyBase.CalcPeriodBeforeEconLimitFound(e, n)  
  
    Call TestFunction(e, n)  
End Sub
```

In the function call example above, the function name is preceded by the `Call` statement followed by the function name with appropriate arguments. If the number of arguments or argument types does not match the function declaration, the build and validation of the project will fail raising an error.

## 4.5 Variables

A variable can be defined as a portion of memory used to store a value as an application works through an algorithm/calculation. The value of the variable can be assigned on declaration and can change by assignment during the process of calculation.

```
Dim TestVariable As Integer
```

Each newly declared variable requires a unique identifier that distinguishes it from other variables; this is commonly known as the variable name. Upon declaration, the variable data type also requires qualification. In the example above, the variable is declared to be of type *Integer*. Data types are covered in more detail in the next section.

### Variable Scope

Variable scope defines when a variable is accessible. When a variable is declared, it will only be available to certain parts of code.

The variables defined and accessed by module functions have two levels of scope: local within the module function and global through the module partial classes.

The variables declared within a function or phase subroutines are considered local variables and can only be accessed within the function or subroutine where it is defined. Local variables are generally used to aid calculation processes and make code easier to read and understand.

The variables declared within the class statements and externally to functions or subroutines are considered global variables and are accessible throughout the class definition in both the designer and implementation files.

## 4.6 Data Types

Variable declarations contain a variable name and type. The name allows the developer to reference the variable from anywhere within the class, and the type determines the variable's data type.

### 4.6.1 Standard Data Types

Visual Basic has a number of standard data types which the Module Suite uses. The standard data types most commonly used are: *Boolean*, *Char*, *Double*, *Integer* and *String*. Other data types can be used by the Module Suite as well. We assume that the module developer is already familiar with them.

## 4.6.2 PRL Data Types

The Module Suite has its own set of predefined data types which can be divided into three groups: Scalars, Lookup Tables and TimeSeries Arrays.

Scalar Arrays	Time Series Arrays	Lookup Tables
ScalarInteger	TimeSeriesInteger	LookupTableInteger
ScalarDouble	TimeSeriesDouble	LookupTableDouble
ScalarDateTime	TimeSeriesDateTime	LookupTableDateTime
ScalarBoolean	TimeSeriesBoolean	
ScalarString		

When referencing a variable declared to be of a type listed above, you need to specify the correct arguments.

The common argument for the data types listed above is the **Partner** argument. This argument allows CASH to calculate a project recursively generating results for specified partners. The returned value for a variable is based on the value of `e.Partner`. To obtain the gross value for a variable, use `e.Partner = Nothing`.

```
Private Function TestFunction(ByVal e As ModuleCalcBeforeLimitArgs,
                             ByVal n As CalcPeriod) As Boolean

    SDVar(e.Partner) = 10.5      'SDVar declared as ScalarDouble
    TSDVar(n, e.Partner) = 10.5  'TSDVar declared as TimeSeriesDouble

    Dim TempDouble As Double     'LUDTab declared as LookupTableDouble
    TempDouble = LUDTab.Lookup(10.0, e.Partner)

End Function
```

The calculation period (`CalcPeriod`) is commonly used by TimeSeries Arrays. In the example above, `n` is passed to the function and used by the `TSDVar` variable. Similar to the scalar variable `SDVar`, the TimeSeries variable `TSDVar` indicates the value associated to the partner at a specific time period.

## Scalar Data Types

A scalar data type is a one-dimensional array holding values which are accessed by qualifying an index, in this case the partner. The figure below gives a visual representation of the data type structure.

Data Type: ScalarInteger

	Gross	Partner 1	Partner 2	...	Partner n
Value	10	10	0	0	0

`n` = number of specified partners

## TimeSeries Arrays

A TimeSeries Array is a two-dimensional array. The values for this structure can only be accessed by qualifying two indexes; in this case they are the partner and the calculation period. The figure below gives a visual representation of the data type structure.

Data Type: TimeSeriesInteger

	Gross	Partner 1	Partner 2	...	Partner n
Period 1	10	20	30	0	0
Period 2	20	400	600	0	0
...	30	600	900	0	0
Period t	0	0	0	0	0

n = number of specified partners

t = number of calculation periods

The value of both *n* and *t* is determined at runtime and is based on the project settings.

**Note:** Declaring data types such as these may require a lot of memory. Based on the number of partners and time periods, the size of the variable will also grow, which will affect the calculation time due to the number of passes the calculation is forced to run. Runtime settings can be used to reduce the number and length of passes.

## Lookup Tables

Lookup tables are special data types resembling a simple two-dimensional array. They are used to return a value by performing a check on a predefined table using an argument passed to the table. This argument will be used to *look up* a corresponding result value. The data type associated with the look-up table indicates the type of data passed to the lookup, with all lookup tables returning a variable of type Double. The figure below gives a visual representation of the data type structure.

Data Type: LookupTableInteger

Lookup Integer	Lookup Result
10	100.00000
20	200.00000
30	300.00000
40	400.00000

The code sample below demonstrates a typical use of the Scalar and TimeSeries variables.

**Note:** All standard assignment operators (=, +=, -=, \*= and /=) are supported for these variables.



```
Private Function TestFunction(ByVal e As ModuleCalcBeforeLimitArgs,
                             ByVal n As CalcPeriod) As Boolean

    SDVar(e.Partner) = 10.5      'SDVar declared as ScalarDouble
    TSDVar(n, e.Partner) = 10.5 'TSDVar declared as TimeSeriesDouble

    SDVar(e.Partner) += TSDVar(n, e.Partner) 'Now SDVar = 21.0 & TSDVar = 10.5
    TSDVar(n, e.Partner) -= 5.5             'Now TSDVar = 5.0

End Function
```

## 4.7 IntelliSense

As mentioned in Chapter 2 "The VSTA Module Editor", the one of the exciting features of VSTA is the ability to use IntelliSense. IntelliSense provides a quick and convenient way to access variables, functions and parameter lists. Yellow boxes pop up to display help text to aid member selection, and keyboard shortcuts allow for word completion.

The use of the *Dot Operator* also exposes available methods which are accessed via referenced objects. For more information about methods and variables, see documentation for Visual Basic.

**Note:** Right-clicking on the code editor gives access to definitions and references of selected objects.

### 4.7.1 IntelliSense

IntelliSense is a drop-down list that appears when a recognized word is being entered into the code editor. It is a context-sensitive tool which allows you to view useful information, insert methods and variables, and complete statements intuitively.

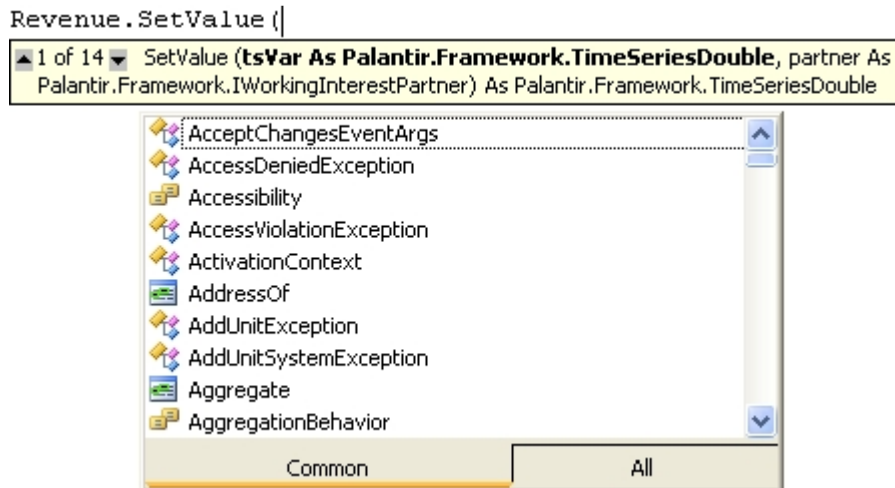


As shown above, when the drop down list is displayed, IntelliSense will also display help text. Help is also available when you hover over the inserted text or any recognized word as shown below.

```
Dim p As IWorkingInterestPartner = e.Partner
```

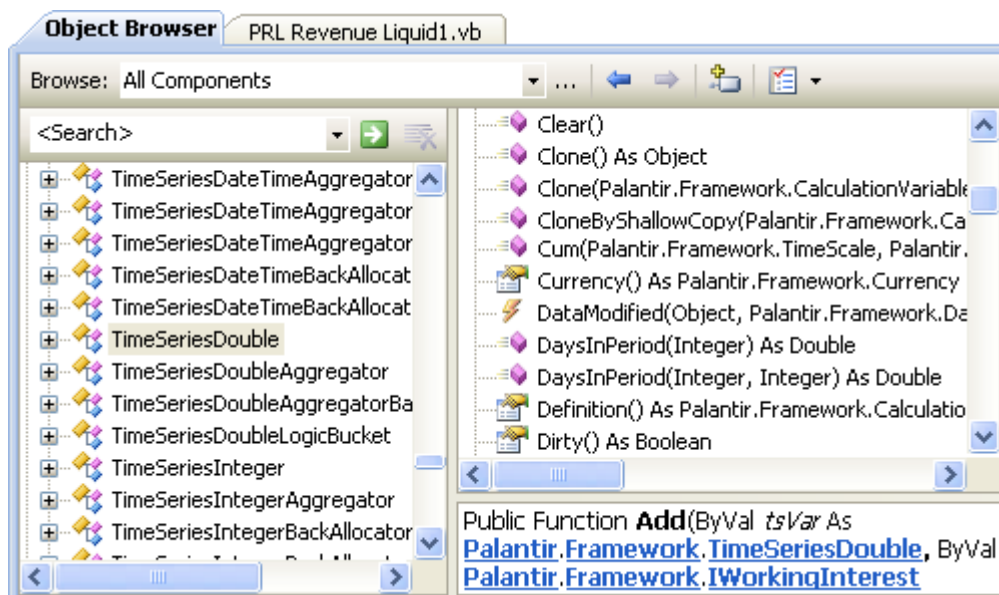
```
Public Property Partner() As Palantir.Framework.IWorkingInterestPartner
```

Another useful feature of IntelliSense is the ability to display a list of parameters for an available function or method. If a method has a number of implementations, use the **Up** and **Down** arrows in the help text to view alternative parameter information as shown below.



## 4.7.2 The Object Browser

The Object Browser allows users to select and examine elements for use in module code within the Palantir Suite. It displays hierarchical structures within the Palantir Framework and more importantly exposes items such as namespaces, libraries, classes, methods and variables.



To launch the Object Browser, go to **View > Object Browser** or press F2.

The Object Browser is a powerful resource giving developers visibility of available methods and functionality. Furthermore, the search facility allows elements to be located quickly without having to drill down to a specific element.

## 4.8 Calculation Phases

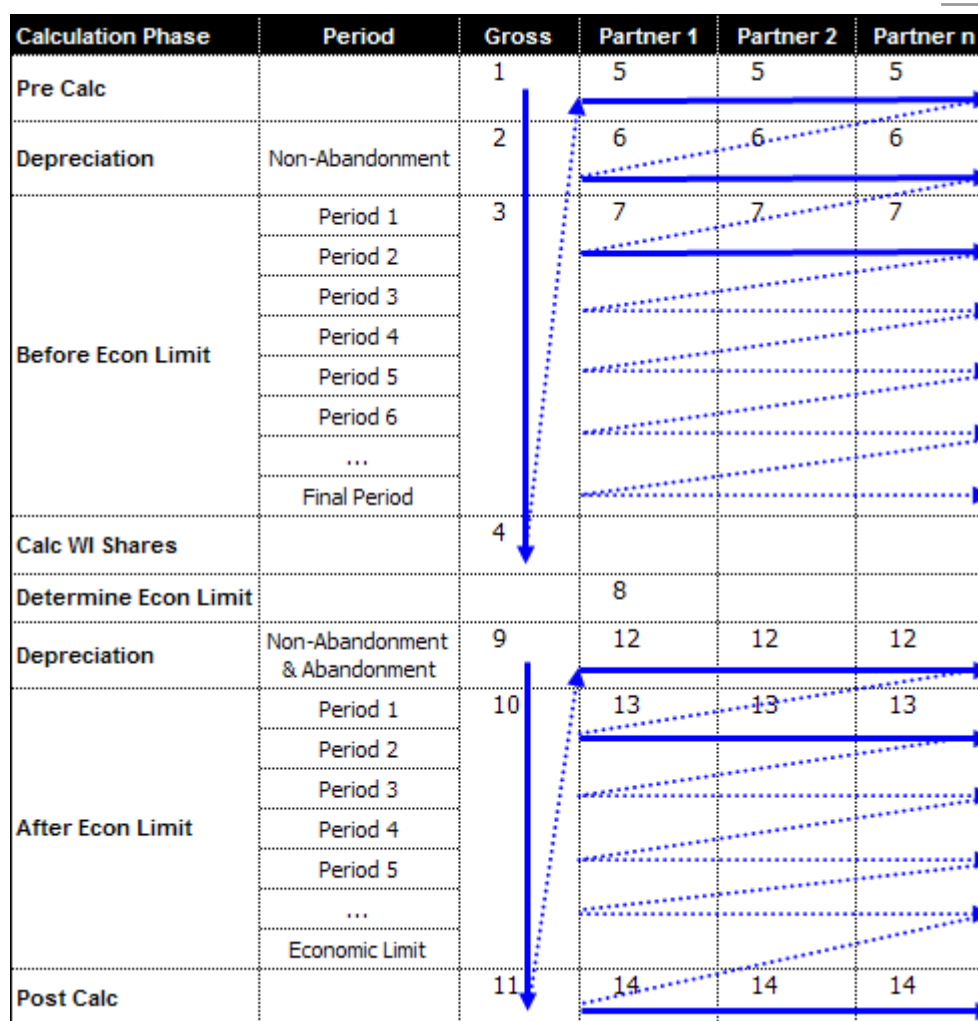
The calculation sequence is divided into eight distinct phases. Of these phases only four are accessible to the developer allowing for the modules to be customized as required.

When the calculate button is clicked within a project, the calculation sequence begins. From here the program runs in order through each phase calling modules sequentially as defined in the Regime. As each module is accessed, the appropriate subroutine within the module is called and executed updating input and output variables as required.

The table below outlines the role of each phase in the calculation sequence. The phases highlighted in grey are available in module code for customisation. The phases are represented by subroutines which are named appropriately.

	Phase	Description
1	<b>Pre Calc</b>	This phase is used to carry out any initial calculation. Values for production volumes and revenues are calculated here in order to obtain totals to feed to other modules in later phases.
2	<b>Depreciation</b>	Here preliminary calculations are carried out to generate depreciation values for use in the next stage.
3	<b>Before Econ Limit</b>	The majority of calculations are executed in this phase. In this phase values such as royalty, cost recovery, profit and taxes are determined.
4	<b>Calc WI Shares</b>	In this phase, CASH can derive working interests for partner share calculations. Phases 1-3 are repeated for all of the partners before moving on to phase 5 below.
5	<b>Determine Econ Limit</b>	The economic limit is derived from the operating partner as defined in the working interest settings in a project. The economic limit date is then applied to all partners and gross.
6	<b>Depreciation</b>	Once determined, the economic limit is used to recalculate the depreciation.
7	<b>After Econ Limit</b>	This phase is used for calculations which do not have an impact on the economic limit and require final depreciation results. In general, corporate tax is calculated in this phase.
8	<b>Post Calc</b>	This phase is commonly used to carry out any final calculations.

Further to the sequence described to this point, CASH additionally calculates results for a number of partners based on project settings. Below you can see a visual representation of this sequence depicting the order in which phases are called and the partners they are being called for.



## 4.9 Periodicity

Periodicity, similar to frequency, determines periods at which Regimes are calculated. Set when a Regime is designed, the periodicity is used during the **Before Economic Limit** and **After Economic Limit** phases and can be set as *Monthly*, *Quarterly*, *Semi Annually* or *Annually*.

As the calculation sequence runs through each phase, the **Before Economic Limit** and **After Economic Limit** phases are set with a loop and called for each period as defined by the Regime periodicity. More specifically, the two phases are called for each period from the start period to the end period which is either determined by the economic limit or defined by the calling project.

Each call of the **Before Economic Limit** or **After Economic Limit** phase is identified by the calculation period which can be fed through to each function called by the subroutine. The calculation period can then be referenced to qualify any TimeSeries variables which are commonly used to define the majority of the input, output and input/output variables in modules.

Although periodicity is set in the Regime and almost exclusively used in the **Before Economic Limit** and **After Economic Limit** phases, it is possible to force a different periodicity setting in any of the accessible phases. This can be used to simulate an alternative periodicity to calculate a value which is either not available in the current phase or cannot be calculated using the Regime periodicity setting.

```
Public Overrides Sub PreCalc(ByVal e As ModulePreCalcArgs)

    Dim i As Integer
    Dim n As CalcPeriod
    Dim StartPrd As CalcPeriod = New CalcPeriod(TimeScale.Monthly,
                                                e.TimeframeInfo.StartYear, 1)
    Dim iLength As Integer = e.TimeframeInfo.Duration * StartPeriod.PeriodsInYear

    For i = 0 To 10
        n = StartPrd.Offset(i)
        Call CalcRate(e.Partner, n)
    Next

End Sub
```

The code extract above demonstrates how a calculation sequence with a specified periodicity can be simulated to generate required results in the **PreCalc** phase. In the example above the periodicity is set to *Monthly*.

## 4.10 Partners and Working Interests

As mentioned earlier, Scalar and TimeSeries variables are both indexed by a partner argument. This argument determines for which partner a value is being assigned/accessed. The working interest for any specified partner is determined in the project document. This value is determined after accounting for any revisions and represented by a percentage which is later applied to determine values for specific partners while working through the calculation sequence.

For the four accessible phase subroutines, the argument *e* declared to be of type associated to the subroutine gives access to the current partner using the qualifier *e.Partner*.

As shown in the calculation sequence diagram in Section 4.8 "Calculation Phases", a gross pass is first executed to determine project values. Beyond this, individual partner values are calculated as CASH runs through the calculation sequence applying the predetermined percentage.

Occasionally, it may be necessary to reference a particular partner by name, for example to override a value or perform a specific calculation. The code extract below demonstrates how to reference and determine the partner name using the *.Name* qualifier. Here we determine whether the current pass is the gross pass and if it is, we increment the global variable *PartnerCount* to determine the number of partners in the project.

```
Public Overrides Sub PreCalc(ByVal e As ModulePreCalcArgs)
    Dim p As IWorkingInterestPartner

    p = e.Partner
    If p.Name <> "Gross" Then
        PartnerCount += 1
    End If
End Sub
```

## 4.11 Subroutine Programming

On execution, the calculation sequence runs through the phase subroutines in order. If the subroutine is not prefixed with `<DoNotCalcMethod()>`, it will be called and in turn will call functions in the **User Functions** region.

Although it is possible to add all required functionality into a phase subroutine, functions are used to define fiscal calculations while subroutines are used to call the functions.

The aim of subroutine programming is to keep things simple. The majority of subroutines defined in the PRL Module Suite use functions to execute fiscal calculations.

```
Public Overrides Sub PreCalc(ByVal e As ModulePreCalcArgs)

    If e.Partner Is Nothing Then
        Call CalcGross(e.Partner)
    Else
        Call CalcPartner(e.Partner)
    End If

End Sub
```

The code extract above demonstrates a simple subroutine. The subroutine confirms the current partner to determine which function to call.

## 4.12 Function Programming

As defined, functions can be called by a phase subroutine or any other function that can feed appropriate arguments into the parameter list.

Function programming allows the programmer to model fiscal calculations as required. The input, output and input/output variables define what is fed into and out of the module.

Those with a basic understanding of Visual Basic will find module development fairly straightforward. The input and output variables are accessible from each declared function where each variable can be initialised, reassigned or used to generate the result for another variable.

To manipulate variables, you can use the standard assignment symbols, operator symbols and methods defined in VSTA and the Module Suite. Ensure you make the appropriate library calls and reference the objects correctly.

```
Private Function CalcTax(ByVal p As IWorkingInterestPartner,
    ByVal n As CalcPeriod, ByVal StartYear As Integer) As Boolean

    TaxableIncome(n, p) = RevenueTotal(n, p) - Deductions(n, p)
    TaxLiability(n, p) = TaxableIncome(n, p) * TaxRate(n, p) / 100

End Function
```

The code extract above demonstrates a very simple tax calculation. In this example, for a specific partner and calculation period `TaxableIncome` is calculated by subtracting `Deductions` from `RevenueTotal`. `TaxLiability` is then determined by taking the product of `TaxableIncome` and `TaxRate` remembering to divide `TaxRate` by 100 to turn it into a fraction.

Although fairly simple, the extract highlights the importance of naming variables appropriately. Doing this allows readers to determine how the module functions without having to reference the designer file to decipher what variables represent.


## 4.13 Building, Validating and Applying Projects

Once the module code has been added / updated in the designer and implementation files, it needs to be compiled to transform it from the source code into an executable format.

In the Palantir Module Suite, this compilation process is broken down into three stages: building, validating and applying. The Palantir toolbar gives developers quick and easy access to these tools, and the standard VSTA functionality is used to monitor and report any errors.

### 4.13.1 Building Projects

The PRL Module Suite is built and tested using the VSTA IDE. Developers generally build a project repeatedly during the development and testing process. In most instances this process begins with compile-time errors which require correction. These errors can include incorrect syntax, references and logic.

The **Build** option is invoked by clicking on  on the Palantir toolbar. This tool takes the source code and runs it through the compiler to identify errors in the code.

### 4.13.2 Validating Projects

The status of the **Build and Validate** process is displayed in the **Output** window. The build process generally completes with a positive or negative result.

If no errors are identified in the code, the process provides positive feedback as shown below.

```

----- Start Building and Validating the Extensibility Code... -----
Palantir (15/09/2009 09:34:27) - Saving the solution.
Palantir (15/09/2009 09:34:27) - Saving the Project.
Palantir (15/09/2009 09:34:27) - The solution has been saved.
Palantir (15/09/2009 09:34:27) - Building the solution for DEBUG...
Palantir (15/09/2009 09:34:33) - The solution has been built successfully.
Palantir (15/09/2009 09:34:47) - Extensibility assembly
<C:\ Temp\clad\LocalPESWorkV35\Modules\bin\Debug\Modules6500.dll> validated successfully.
----- Building and Validating Done Successfully -----

```

If the build process identifies one or more errors, it provides negative feedback identifying a failed build as shown below. Resolving a failed build can be quite straightforward.

```

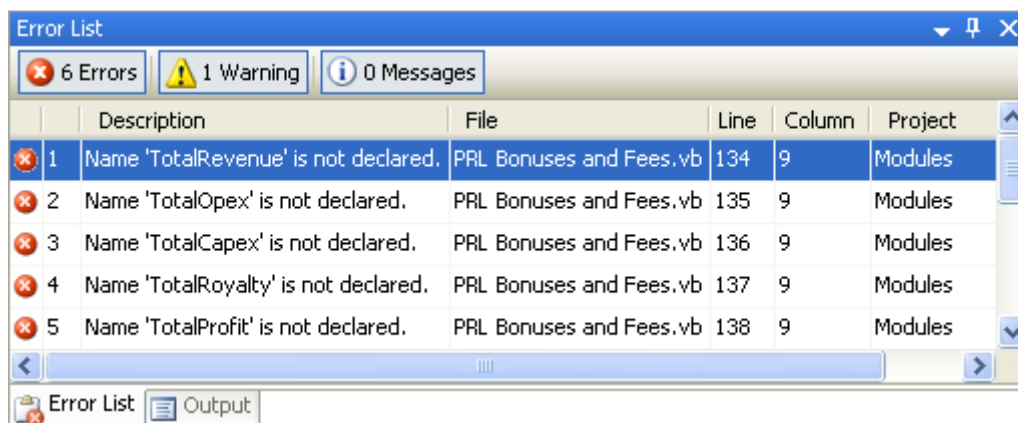
----- Start Building and Validating the Extensibility Code... -----
Palantir (15/09/2009 11:02:39) - Saving the solution.
Palantir (15/09/2009 11:02:40) - Saving the Project.
Palantir (15/09/2009 11:02:45) - The solution has been saved.
Palantir (15/09/2009 11:02:47) - Building the solution for DEBUG...
Palantir (15/09/2009 11:03:33) - The solution built failed.
Palantir (15/09/2009 11:03:33) - Build failed. Please check the build errors.
----- Building and Validating Failed -----

```

## Error List Window


During the development process, the **Error List** window keeps an up-to-date list of identified errors. This speeds up the development process.

The window displays errors, warnings and messages identified as errors during module development. If you double-click on an identified error, VSTA will open the file where the error is and show the error location.



### 4.13.3 Applying Changes to the Database

Once the **Build and Validate** process creates a successful build, the changes need to be applied to the database to be used for calculations.

To apply changes to the database, click on  on the Palantir toolbar. This will invoke the **Build and Validate** function if you have not already called it, and, if the build is successful, save the changes to the database as shown below.



```
----- Start Apply the Changes... -----  
Palantir (15/09/2009 11:45:27) - Extensibility classes and references retrieved.  
----- Completed Apply Changes Successfully -----
```

Once applied, the changes are stored in the CASH database and will form part of the calculation sequence when a project is next calculated.

Now you can safely close VSTA if you don't need to make any more changes to the PRL Module Suite.

**Note:** If changes are not successfully committed to the database, they will be lost. When the **Edit Modules** option is selected from the **Tools** menu, any locally saved changes will be overwritten during initialisation of VSTA.

## Chapter 5. Testing and Debugging


Building, debugging and testing are key tasks when developing any application. This methodology also applies when developing module code for the PRL Module Suite.

Once all build errors are resolved, it is necessary to correct any logic errors which may keep the module from running as intended. This can be achieved by using the VSTA debugging functionality.


Debugging is the process which allows a developer to locate and fix bugs in the source code by pausing and analysing the run-time behaviour of an application, in this case the calculation sequence. The debugger allows you to suspend the execution of the calculation sequence at a defined point and examine the code, evaluate or edit variable values, and jump to different points in the execution path.

The VSTA Debugger provides a number of tools to simplify debugging. Debugger windows, dialogue boxes and pop-ups display runtime information which can be updated to simulate different scenarios.

### 5.1 Attaching the Debugger

To invoke the debugger, click on  on the Palantir toolbar. First, the debugger checks if the modules have been built and validated successfully. If they have, the VSTA debugger will be automatically attached to the running CASH application.


Running in the debug mode allows you to run through the calculation sequence, pause at specified points and analyse/step through the code.

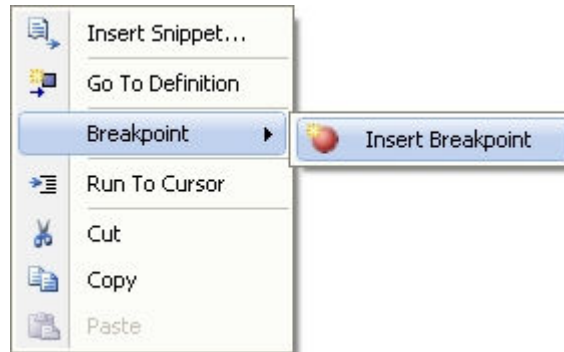
To start a calculation sequence, click on  in an active project. The calculation sequence will start from the beginning and run through each phase in the established order.

### 5.2 Debugging


In the debug mode, the calculation sequence will run through without pausing as we have not flagged any point for the execution path to pause.

#### Break Points










To force the calculation sequence to pause at a specific point, add break points to the module code: navigate to the required place, right-click and select the appropriate option from the context menu (shown below), or press F9. If a break point has been successfully added, a red dot  will be inserted into the margin of the module code.

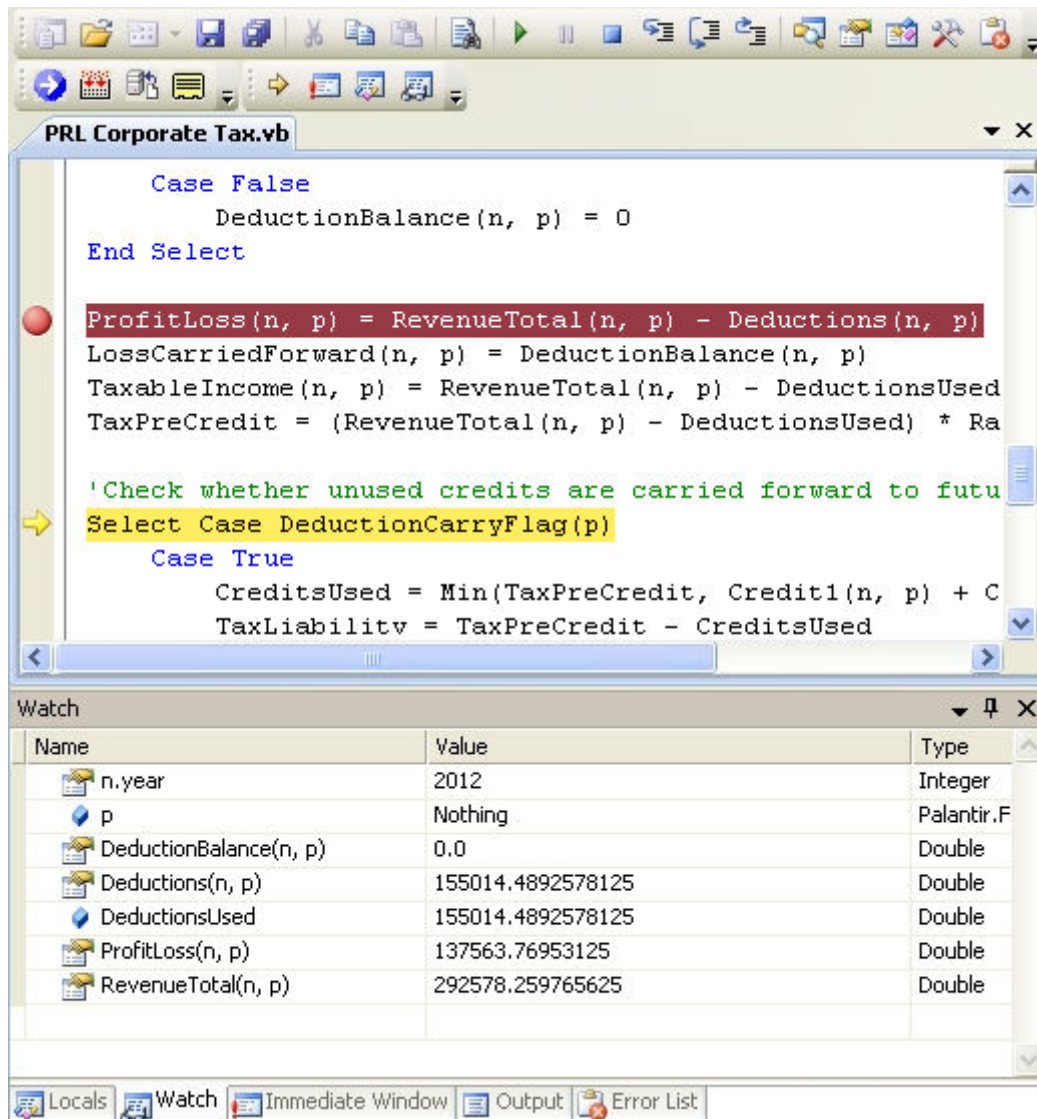


## Debugging Options

Once a break point has been added, click on  to re-run the calculation sequence. However, this time the process will pause at the defined break point. This allows you to view the status of any declared variable and to step through the code analysing how the code processes data.

The table below describes the toolbar options used to debug module code.

Icon	Name	Description
	<b>Continue</b>	Continues to the next break point or logical end.
	<b>Stop Debugging</b>	Stops the debugger.
	<b>Step Into</b>	Executes code one statement at a time, stepping into functions called.
	<b>Step Over</b>	Executes a function or procedure without having to execute each line at a time.
	<b>Step Out</b>	Executes the remainder of the function and moves to the next statement after the function call.
	<b>Next Statement</b>	Moves the cursor to the line that will be executed next.
	<b>Immediate</b>	Displays the <b>Immediate</b> window used to evaluate expressions.
	<b>Locals</b>	Displays the <b>Locals</b> window used to display the information about all the local variables in the current scope.
	<b>Watch</b>	Displays the <b>Watch</b> window used to specify variables and expressions you want to watch while debugging.



The picture above shows a typical debug sequence. Once paused, the debugger has access to the local variables via the **Watch** window and can step through each line of code to determine how the statements change their values.

## Chapter 6. Code Examples

This chapter outlines some methodologies commonly used within the Module Suite but may be considered non-standard for software development.

### 6.1 Declaring Calculation Periods and Periodicity Loops

Calculation periods can be declared locally to simulate a loop with a different periodicity to that of the Regime. This allows you to access members of TimeSeries variables using a different periodicity.

```
Dim StartPeriod As CalcPeriod = New CalcPeriod(TimeScale.Monthly,  
                                                e.TimeframeInfo.StartYear, 1)  
Dim iLength As Integer = e.TimeframeInfo.Duration * StartPeriod.PeriodsInYear  
  
For i = 0 To iLength - 1  
    n = StartPeriod.Offset(i)  
    TempVariable(e, n) = 0
```

Next

The code extract above dimensions a new variable *StartPeriod* which is initialised with periodicity, start year and initial period. The **iLength** variable is similarly dimensioned and initialised with the number of periods in the project.

In this instance, the **FOR** loop steps through each time period in the project allowing you to access members of a TimeSeries variable.

Local periodicity loops are commonly declared in the **PreCalc** subroutine or in instances where a loop needs to reference items in a different periodicity to that of the project.

### 6.2 First Calculated Period Check

A common requirement for many subroutines is to differentiate the first period of a calculation sequence from all other periods.

```
If n.Year = e.TimeframeInfo.StartYear And n.Period = 1 Then  
    p = e.Partner  
    Call Calc Split(p)  
End If
```

The code extract above determines the StartYear from the calculation argument and uses the current calculation periods indicated by 'n' to establish whether the current period is the first period of the calculation sequence.

## 6.3 TimeSeries Object Declaration

You can declare a `TimeSeriesDouble` object within the instance of a class. These variables are generally used for local calculations within the scope of a defined block and cannot be used for any form of external reporting. The use of local `TimeSeriesDouble` variables over regular `Doubles` or `ScalarDoubles` is very much down to the application of the declared variable. The `TimeSeriesDouble` variable permits storage of values over different time periods and for different partners.

The local `TimeSeriesDouble` is commonly set up using two command lines. The first is the common dimensioning of the variable to create a reference to the object to be created. The second uses the `New` statement to create an instance of the `TimeSeriesDouble` object with defined parameters as shown below.

```
Dim TempTSD As TimeSeriesDouble
TempTSD = New TimeSeriesDouble(Royalty.Definition,
                               Royalty.StartYear, Royalty.Duration)
```

The code extract above can be further simplified by dimensioning the reference and creating an instance of the object of the `TimeSeriesDouble` as shown below.

```
Dim TempTSD As New TimeSeriesDouble(Royalty.Definition,
                                     Royalty.StartYear, Royalty.Duration)
```

## Chapter 7. Accessible Framework Attributes

Following on from the module specific calculation settings and the calculation period settings discussed in Section 4.2 "Subroutine Arguments and Local Qualifiers", this chapter covers the common attributes accessible via the Palantir Framework.

In addition to the members of the PRL Module Suite, there are a number of accessible attributes which are available via the **Me.** qualifier which contains calculation-specific attributes.

### 7.1 Me.CalcParm

Returns a 'CalculationParameter' object representing a Calculation Parameters document used for the current calculation.

Qualifier	Description
.DocumentInfo	Returns a 'CalculationParameterInfo' object representing the document information for the current calculation parameter.
.Guid	Returns the GUID – Unique Identifier of the Calculation Parameters document.
.ID	Returns the ID for the Calculation Parameters document.
.Name	Returns a 'String' object representing the name of the Calculation Parameters document.
.Properties	Returns a 'DocumentPropertyCollection' object representing the properties for the current Calculation Parameters document.
.ReadOnly	Returns a 'Boolean' indicating whether the document is read-only or not.
.State	Returns a 'DocumentState' object indicating whether the document is <i>OpenRead</i> , <i>OpenWrite</i> or <i>Closed</i> .

### 7.2 Me.CalculationCurrency

Returns a 'Currency' object representing the currency used for module calculation.

### 7.3 Me.CalculationCurrencyScenario

Returns a 'CurrencyScenario' object representing the currency scenario used for the module calculation.

Qualifier	Description
.Guid	Returns the GUID – Unique Identifier for the currency scenario.
.ID	Returns the ID for the currency scenario.
.Name	Returns a 'String' object representing the name of the currency scenario.
.StartYear	Returns an 'Integer' representing the start year for the currency scenario.

## 7.4 Me.CalculationProjects

Returns a collection of 'Project' objects representing the project to be calculated.

Qualifier	Description
.Count	Returns an 'Integer' representing the number of projects in the collection.
.Item(i)	Returns a 'Project' object referenced by a specified index of the collection.

## 7.5 Me.CurrentPriceScenarioDefinition

Returns a 'PriceScenarioDefinition' object representing the price scenario applied to the current calculation.

Qualifier	Description
.Default	Returns a 'Boolean' indicating whether the current price scenario is the default price scenario.
.EscalationDate	Returns a 'YearMonth' object representing the escalation date for the price scenario.
.Guid	Returns the GUID – Unique Identifier for the price scenario.
.ID	Returns the ID for the price scenario.
.Name	Returns a 'String' object representing the name of the price scenario.
.StartYear	Returns an 'Integer' representing the start year for the price scenario.

## 7.6 Me.CurrentScenario

Returns a 'ProjectScenarioDefinition' object representing the project scenario being calculated.



Qualifier	Description
.Default	Returns a 'Boolean' indicating whether the current project scenario is the default project scenario.
.Guid	Returns the GUID – Unique Identifier for the project scenario.
.ID	Returns the ID for the project scenario.
.IsFailureScenario	Returns a 'Boolean' indicating whether the current project scenario is a failure scenario.
.Name	Returns a 'String' object representing the name of the price scenario.

## 7.7 Me.CurrentScenarioWeighting

Returns a 'Double' representing the weighting of the current project scenario being calculated.

## 7.8 Me.Price

Returns a 'TimeSeriesDouble' object representing the price value based on specified arguments.

Qualifier	Description
.Currency	Returns a 'Currency' object representing the currency used for module calculation.
.Definition	Returns a 'CalculationVariableDefinition' object defining specific parameters of the 'TimeSeriesDouble' object.
.Duration	Returns an 'Integer' representing the duration of the object.
.EscalateOnCalculation	Returns a 'Boolean' indicating whether the current object is to be escalated.
.InflationDate	Returns a 'YearMonth' object representing the inflation date for the current object.
.RealNominal	Returns a 'RealNominalFlag' object indicating whether a value has been entered in real or nominal terms.
.StartYear	Returns an 'Integer' representing the start year for the current object.
.YearMonth	Returns a 'YearMonth' object representing the year and month the variable is defined.

## 7.9 Me.PriceName

Returns a 'String' object representing the price used for module calculation.

## 7.10 Me.ThisCalculation

Returns a 'CalculationBase' object representing the calculation document which the module is currently operating on.

## 7.11 Me.ThisEngine

Returns an 'Engine' object representing the base economic engine which contains all other classes.

### Me.ThisEngine.ApplicationName

Returns a 'String' object indicating the application name. By default *PalantirCash* is returned.

### Me.ThisEngine.CalculationParameters

Returns a 'CalculationParameterCollection' object representing a collection of Calculation Parameters documents.

Qualifier	Description
.CanBeSynchronized	Returns a 'Boolean' indicating whether the current object can be synchronized.
.CanUserModify	Returns a 'Boolean' indicating whether the current user can modify the collection.
.Count	Returns an 'Integer' indicating the number of elements contained within the referenced collection.
.DocumentType	Returns a 'String' object indicating the document type. In this instance the default value is <i>Calculation Parameter</i> .
.ExportDisplayName	Returns a 'String' object indicating the name or category of the items that the collection represents.
.ExportElementName	Returns a 'String' object indicating the name or category of the items that the collection represents.
.ExportMethod	Returns an 'ExportMethod' object specifying how data should be exported for the current collection. Possible values: ENTIRE collection or PART of the collection.

Qualifier	Description
.HasDependantItems	Returns a 'Boolean' indicating whether current object has any dependents.
.IsImportInProgress	Returns a 'Boolean' determining whether the documents are currently being imported.
.Item	Returns an item/object from the collection referenced by a specified index.
.MustImport	Returns a 'Boolean' determining whether the data for this object must be imported.
.Names	Returns a list of 'String' objects representing the names of the documents in the collection.
.OverwritableAttributes	Returns a collection of 'String' and 'Boolean' objects used to determine the attributes that can be selected for overwriting during the import process.
.Parent	Returns an 'Engine' object qualifying the economic engine which contains the class.

## Me.ThisEngine.CurrencyConversion

Returns a 'CurrencyConversion' object containing information and methods allowing for currency conversion.

Qualifier	Description
.CanBeSynchronized	Returns a 'Boolean' indicating whether the current object can be synchronized.
.CanUserModify	Returns a 'Boolean' indicating whether the current user can modify the collection.
.CurrencyScenarios	Returns a 'CurrencyScenarioCollection' object representing the currency scenarios defined in the database.
.ExportDisplayName	Returns a 'String' object indicating the name or category of the items that the 'CurrencyConversion' object represents.
.ExportElementName	Returns a 'String' object indicating the name or category of the items that the object represents.
.ExportMethod	Returns an 'ExportMethod' object specifying how data should be exported. Possible values: ENTIRE object or PART of the object.
.HasDependantItems	Returns a 'Boolean' indicating whether current instance of a currency conversion object has any dependents.
.Item	Returns a 'CurrencyConversion' object referenced by a specified index.
.MustImport	Returns a 'Boolean' determining whether the data for this object must be imported.

Qualifier	Description
.OverriddenCurrencies	Returns a list of 'Currency' objects determining the overridden currencies.
.OverwritableAttributes	Returns a collection of 'String' and 'Boolean' objects used to determine the attributes that can be selected for overwriting during the import process.
.Parent	Returns an 'Engine' object qualifying the economic engine which contains the class.

## Me.ThisEngine.DataBaseName

Returns a 'String' object indicating the current database name.

## Me.ThisEngine.DataSource

Returns a 'DataSource' object representing the class that contains the database connection and database helper methods.

## Me.ThisEngine.DepreciationMethodEngine

Returns a 'DepreciationMethodEngine' object representing the Visual Studio Engine used to calculate capital depreciation.

## Me.ThisEngine.GlobalDataDocuments

Returns a 'GlobalDataDocumentCollection' object representing a collection of Global Data Documents.

All collection objects inherit from a generic collection object and thus have the same qualifiers. For a description of the qualifiers, see the Me.ThisEngine.CalculationParameters.

## Me.ThisEngine.Hierarchies

Returns a 'CalculationHierarchyCollection' object representing a collection of hierarchies.

All collection objects inherit from a generic collection object and thus have the same qualifiers. For a description of the qualifiers, see Section "Me.ThisEngine.CalculationParameters" above.

## Me.ThisEngine.LastSelectedHierarchy

Returns a 'CalculationHierarchy' object representing the last selected hierarchy, a hierarchy containing consolidations and documents.

## Me.ThisEngine.ModuleEngine

Returns a 'ModuleEngine' object representing the Visual Studio module engine used for calculation.

## Me.ThisEngine.Password

Returns a 'String' object representing the password set by the current user.

## Me.ThisEngine.Prices

Returns a 'PriceCollection' object representing a collection of price documents.

All collection objects inherit from a generic collection object and thus have the same qualifiers. For a description of the qualifiers, see Section "Me.ThisEngine.CalculationParameters" above.

## Me.ThisEngine.Regimes

Returns a 'RegimeCollection' object representing a collection of Regime documents.

All collection objects inherit from a generic collection object and thus have the same qualifiers. For a description of the qualifiers, see Section "Me.ThisEngine.CalculationParameters" above.

## Me.ThisEngine.ReportingEngine

Returns a 'ReportingEngine' object representing the engine used for report generation.

Qualifier	Description
.ComparisonReportViewHierarchy	Returns a 'ReportViewHierarchy' object which represents the comparison Result Set for a Sensitivity Run.
.ComparisonSensitivityRun	Returns a 'ResultSetSensitivityRun' object which represents the class for a Result Set's Sensitivity Run.
.CurrentReports	Returns a 'CurrentReportCollection' object which represents a collection of 'CurrentReport' objects.
.CurrentSelectedConsolidations	Returns a list of 'ReportViewProjectNode' objects setting the report view consolidations for the Result Set reporting.
.CurrentSelectedProjects	Returns a list of 'ReportViewProjectNode' objects setting the report view project nodes for Result Set reporting.
.CurrentSelectedProjectScenario	Returns a 'ProjectScenario' object representing the project scenario for scratchpad reporting.

Qualifier	Description
.CurrentSelectedTemplates	Returns an array of 'ReportTemplate' objects used for reporting.
.EnableScratchpadReporting	Returns a 'Boolean' to enable/disable scratchpad reporting after the project calculation.
.IsIncrementalReporting	Returns a 'Boolean' to determine if reporting is incremental.
.Parent	Returns an 'Engine' object qualifying the economic engine which contains the class.
.ReportingFileOptions	Returns a 'ReportingFileOptions' object which represents the reporting file options for a specified report.
.ReportSettings	Returns a 'ReportSettings' object representing the report settings for a specified report.
.ReportTemplates	Returns a 'ReportTemplateCollection' object representing the collection of 'ReportTemplate' objects defined in the database.
.ReportViewHierarchy	Returns a 'ReportViewHierarchy' object which represents the report view hierarchy.
.ReportViews	Returns a 'ReportViewCollection' object which represents the collection of 'ReportView' objects.
.ResultSetProjectProperties	Returns a 'ResultSetProjectPropertyCollection' object representing the collection of 'ResultSet' objects representing project properties of Result Sets which are saved at the time of calculation.
.SelectedCalculationParameter	Returns a 'CalculationParameter' object which represents the Calculation Parameters document used for Result Set reporting.
.SelectedPartners	Returns an array of 'IWorkingInterestPartner' objects representing the companies to be included for reporting.
.SelectedReportView	Returns a 'ReportView' object representing the report view applied for the report hierarchy.
.SelectedResultType	Returns a 'ReportResultType' object which represents the reporting result type for Result Set reporting.
.SelectedSensitivityRuns	Returns an array of 'ResultSetSensitivityRun' objects representing selected Sensitivity Run objects.
.UseAllProjectsForComparison	Returns a 'Boolean' indicating if all the projects (not just matched projects) in a consolidation should be considered for Result Set comparison. This option is used only if <b>IsIncremental</b> is selected.

## Me.ThisEngine.ResultSets

Returns a 'ResultSetCollection' object representing a collection of calculation Result Sets.

All collection objects inherit from a generic collection object and thus have the same qualifiers. For a description of the common qualifiers, see Section "Me.ThisEngine.CalculationParameters" above.

## Me.ThisEngine.Settings

Returns a 'SettingsController' object detailing miscellaneous settings for the engine.

Qualifier	Description
.CalcParmPropertyDefinitions	Returns a 'CalculationParameterPropertyDefinitionCollection' object representing a collection of settings for Calculation Parameters documents.
.CalculationVariableDefinitions	Returns a 'CalculationVariableDefinitionCollection' object representing a collection of settings for calculation variables and their behaviour.
.ContractPropertyDefinitions	Returns a 'ContractPropertyDefinitionCollection' object representing a collection of project document property definitions.
.EscalationTypes	Returns an 'EscalationTypeCollection' object representing a collection of categories that values can be escalated with.
.Miscellaneous	Returns a 'MiscellaneousSettings' object which represents miscellaneous settings used in the application.
.Parent	Returns an 'Engine' object qualifying the economic engine which contains the class.
.Partners	Returns a 'PartnerCollection' object representing a collection of globally defined lists of partners that have a working interest share in one or more projects.
.PricePropertyDefinitions	Returns a 'PricePropertyDefinitionCollection' object representing a collection of price document property definitions.
.PriceScenarios	Returns a 'PricescenarioDefinitionCollection' object representing a collection of all possible scenarios for price documents where the user can enter data.
.ProductCategories	Returns a 'ProductCategoryCollection' object which represents a collection of product streams that economics can be run for.
.ProjectPropertyDefinitions	Returns a 'ProjectPropertyDefinitionCollection' object

Qualifier	Description
	representing a collection of project document property definitions.
.ProjectScenarios	Returns a 'ProjectScenarioDefinitionCollection' object representing a collection of project scenarios that the user can enter data into.
.ProjectTemplates	Returns a 'ProjectTemplateCollection' object which represents a collection of project templates.
.PropertyDictionaries	Returns a 'PropertyDictionaryCollection' object which represents a collection of dictionaries that can be used by the property definitions.
.RegimePropertyDefinitions	Returns a 'RegimePropertyDefinitionCollection' object representing a collection of Regime property definitions.
.ReportGroups	Returns a 'ReportGroupCollection' object representing a collection of report groups.
.VariableCategories	Returns a 'VariableCategoryCollection' object which represents a collection of categories that values can be escalated with.
.WorkingInterestCategories	Returns a 'WorkingInterestCategoryCollection' object representing a collection of working interest category definitions.

## Me.ThisEngine.UnitConversion

Returns a 'UnitConversion' object representing the information necessary to convert values between unit systems and currencies.

Qualifier	Description
.CanBeSynchronized	Returns a 'Boolean' indicating whether the current object can be synchronized.
.CurrencyConversion	Returns a 'CurrencyConversion' object for use by the engine class.
.ExportDisplayName	Returns a 'String' object indicating the name or category of the items that the collection represents.
.ExportElementName	Returns a 'String' object indicating the name or category of the items that the collection represents.
.ExportMethod	Returns an 'ExportMethod' object specifying how data should be exported for the current collection. Possible values: ENTIRE collection or PART of the collection.
.HasDependantItems	Returns a 'Boolean' indicating whether the current object has any dependents.



Qualifier	Description
.Item	Returns an item/object from the collection referenced by a specified index.
.MustImport	Returns a 'Boolean' determining whether the data for this object must be imported.
.OverwritableAttributes	Returns a collection of 'String' and 'Boolean' objects used to determine the attributes that can be selected for overwriting during the import process.
.Parent	Returns an 'Engine' object qualifying the economic engine which contains the class.
.UnitGroups	Returns a 'UnitGroupCollection' object representing a collection of unit groups used to classify units.
.Units	Returns a 'UnitCollection' object representing a collection of units.
.UnitSystems	Returns a 'UnitSystemCollection' object representing a collection of unit systems.

### **Me.ThisEngine.UsedByCalcService**

Returns a 'Boolean' determining whether the application is licensed.

### **Me.ThisEngine.UserName**

Returns a 'String' object representing the current user's username.