2011

Учебник по TestComplete

Практическое руководство

Данное руководство будет полезно всем, кто начинает изучение программы для автоматизации тестирования TestComplete, а также тем, кто уже работал с ним и хочет углубить свои знания

Геннадий Алпаев
http://tctutorial.ru/
25.04.2011

Содержание

Введение

TestComplete — один из самых мощных коммерческих продуктов, предназначенных для автоматизации тестирования программного обеспечения.

Скачать TestComplete можно здесь.

В этом руководстве описаны основы работы с этой программой, которые будут полезны как новичкам, так и опытным пользователям TestComplete.

Мы рекомендуем читателю не просто читать это руководство, но и параллельно выполнять те же действия, которые в нем описаны. Ведь только практикуясь можно достичь наилучших результатов.

Для примеров нашей книги мы выбрали язык JScript. Если вы планируете использовать его и в своем проекте — просто повторяйте задания, которые описаны в книге. Если же вы планируете использовать другой язык, то так даже лучше, так как вы получите больше опыта в написании скриптов.

Совсем необязательно читать все главы подряд. Для начинающих желательно прочитать первые три главы, чтобы получить общее представление о работе с TestComplete, а затем читать те главы, которые необходимы в работе или просто инетересны.

Все примеры, используемые в учебнике, можно скачать здесь.

Удачи!

Чего нет в этом руководстве

В этом пособии мы постарались осветить наиболее важные и часто используемые возможности TestComplete-а и подходы к написанию тестовых скриптов. Тем не менее, вот несколько пунктов, которых вы не найдете здесь:

1. **Руководство по программированию.** Это не учебник по программированию. Для того чтобы писать скрипты, необходимы хотя бы базовые познания в программировании на любом языке. Все примеры кода приведены только на языке JScript для экономии времени создания учебника
2. **Полный справочник по TestComplete.** Наиболее полным руководством по TestComplete является справочная система, поставляемая вместе с ним (на английском языке). Данное руководство не претендует на полное описание всех возможностей TestComplete-а.
3. **Учебник по тестированию.** При написании этого руководства мы используем знания из разных областей, в частности из области тестирования. Однако это — не учебник по тестированию.
4. **Руководство по языкам программирования, используемым в TestComplete.** TestComplete позволяет использовать один из пяти языков программирования. В этом руководстве нет описания этих языков, так как это выходит за рамки учебника. Если вы хотите изучить тот или иной язык, вам придется обратиться к другим источникам. В частности:

- Описание языка DelphiScript есть в справочной системе TestComplete-а (см. раздел справки DelphiScript Description)
- Описание зыка JScript можно найти в онлайн руководстве от Майкрософт, либо скачать его оффлайн версию
- Описание зыка VBScript также можно найти в онлайн руководстве от Майкрософт, либо скачать его оффлайн версию
- Языки C++Script и C#Script по сути являются тем же JScript с немного измененным синтаксисом. В нем используются те же самые объекты, свойства и методы. Для изучения этих языков вам понадобится изучать руководство по JScript (см. выше).

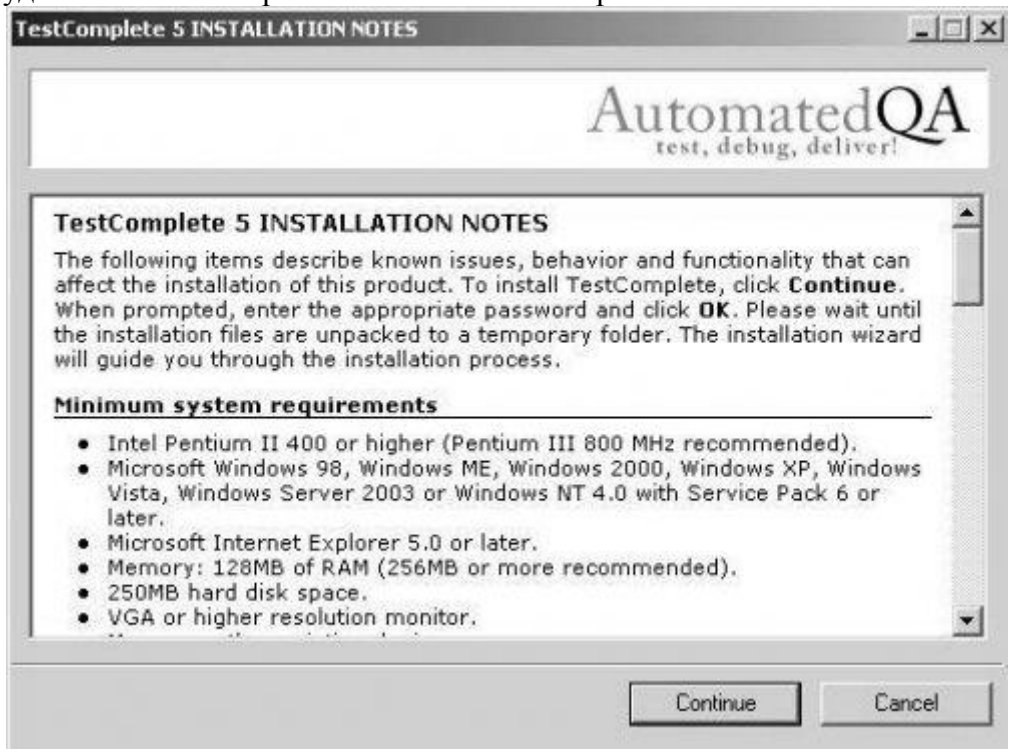
1 Инсталляция и первое знакомство

1.1 Инсталляция

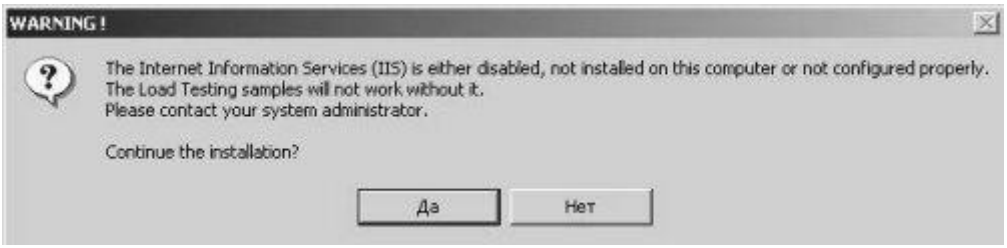
Инсталляция TestComplete не представляет особых сложностей и не отличается от инсталляции многих продуктов.

Для установки TestComplete вы должны войти в систему с правами администратора.

После запуска инсталляционного файла вы увидите первое окно Installation Notes, в котором перечислены минимальные системные требования, необходимые для работы TestComplete. Проверьте, что ваш компьютер удовлетворяет им и продолжайте установку. Кроме того, в этом окне находятся дополнительные указания для проведения нагрузочного тестирования (Load Testing), тестирования Java и .NET приложений, особенности установки TestComplete для системы Windows 98, а также указания по удалению TestComplete с вашего компьютера.



Далее последуют окно лицензионного соглашения, выбор, для кого будет доступно приложение после инсталляции (только для текущей сессии или для всех пользователей данного компьютера), выбор папки для установки TestComplete и имени ярлыка в меню программ. После этого может появиться предупреждение о неактивном сервисе IIS (Internet Information Services).

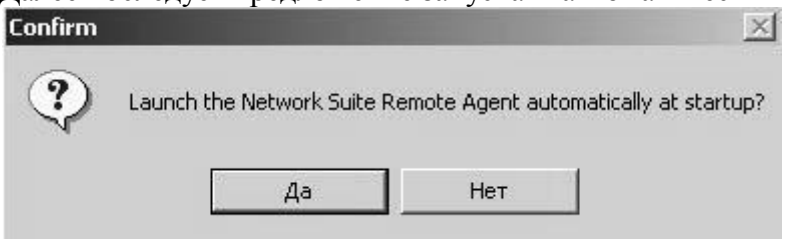


Если вы планируете проводить нагрузочное тестирование, то вам придется прервать установку, нажав кнопку Нет, и включить этот сервис (Панель управления – Установка и удаление программ – Установка компонентов Windows). Если же проведение нагрузочного тестирования не предполагается, нажмите Да и продолжайте установку.

В следующем окне вам будет предложено выбрать, какие компоненты и примеры устанавливать. Здесь рекомендуется оставить все по умолчанию, так как TestComplete отключает те компоненты, которые в данный момент не нужны (например, интеграция с MS Visual Studio отключена, если последний не установлен и т.д.).

Обратите внимание: не рекомендуется включать интеграцию с TestComplete 3, если у вас нет скриптов, которые были разработаны в третьей версии TestComplete. Данная опция предназначена лишь для обратной совместимости скриптов. В противном случае у вас могут возникнуть проблемы с написанием и воспроизведением скриптов на разных компьютерах.

Далее последует предложение запускать автоматически Network Suite Remote Agent.



Remote Agent – это утилита для обмена данными между TestComplete проектами, которые будут запускаться совместно в сети. Если вы планируете подобные запуски нажмите Да, иначе - Нет. Если вы не знаете в данный момент, будет ли проводиться подобное тестирование, нажмите Нет: вы всегда сможете добавить эту утилиту (tcrea.exe) в автозапуск вручную.

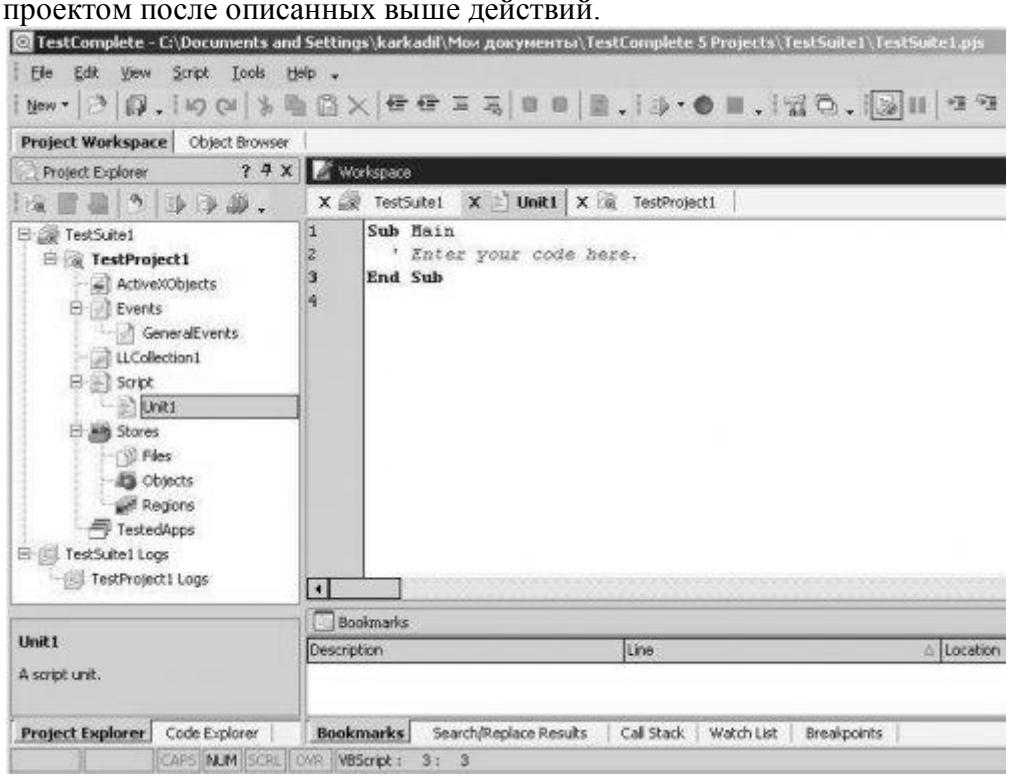
На этом рассмотрение инсталляции можно завершить. Теперь перейдем к изучению непосредственно приложения TestComplete.

1.2 Первое знакомство

Запустите TestComplete, выберите пункт меню *File - New - New Project Suite*. В появившемся окне Create Project Suite введите имя для нового набора проектов (Name), укажите его расположение на диске (Location) и нажмите OK. Теперь выберите пункт меню *File - New - New Project*. В открывшемся окне в разделе templates выберите пункт General-Purpose Test Project (он выбран по умолчанию) и так же, как и для Project Suite введите его имя и расположение на диске. Нажмите OK, затем Finish.

В данный момент мы не будем подробно разбирать создание нового проекта, сделаем лишь небольшой обзор среды TestComplete.

На рисунке показан пример того, как выглядит TestComplete с открытым пустым проектом после описанных выше действий.



В левой части окна TestComplete находится Project Workspace (рабочее пространство проекта). Из него вы получаете доступ ко всем элементам ваших проектов: файлы скриптов (units), тестируемые приложения (TestedApps), хранилища (Stores), результаты работы скриптов (Logs) и т.п. Все они будут рассмотрены подробно чуть позже. Project Workspace имеет две вкладки: Project Explorer, через который вы открываете элементы ваших проектов, и Code Explorer, в котором в удобном виде отображаются скрипты и содержащиеся в них процедуры и функции.

Рядом с Project Workspace находится панель Object Browser (браузер объектов). В нем отображаются все запущенные процессы, их внутренние объекты, а также свойства процессов и объектов.
В правой части окна находится ваше рабочее пространство (Workspace), в котором вы можете переключаться между всеми открытыми элементами проекта и просматривать их содержимое. В нижней правой части находятся несколько полезных вкладок, в которых вы можете просмотреть закладки (Bookmarks), результаты поиска и замены (Search/Replace Results), точки останова (Breakpoints) при отладке скриптов и т.п.

Все эти панели (Project Explorer, Code Explorer, Workspace, Bookmarks и т.п.) можно перемещать и комбинировать, как вам угодно. Для этого необходимо щелкнуть левой кнопкой мыши на заголовке соответствующей панели и, не отпуская кнопку, перетаскивать панель. При этом на экране будут появляться указатели, позволяющие прикрепить панель в нужное место. Либо вы можете оставить панель как отдельное окошко (наподобие плавающих панелей инструментов в MS Office).
Для того чтобы вернуть все панели в первоначальное состояние, необходимо выбрать пункт меню *View - Desktop - Restore Default Docking*. Кроме того, можно сохранить

понравившееся вам расположение панелей во внешний файл (*.tcDock) с помощью меню *View - Desktop - Save Docking to File*, либо загрузить из имеющегося файла, используя пункт меню *View - Desktop - Load Docking From File*.

Если вы закрыли какую-либо панель, вы можете открыть ее снова, воспользовавшись меню *View - Select Panel*.

2 Начало работы с TestComplete

В этом разделе даются основы работы с TestComplete, необходимые новичкам для того, чтобы ознакомиться с приложением.

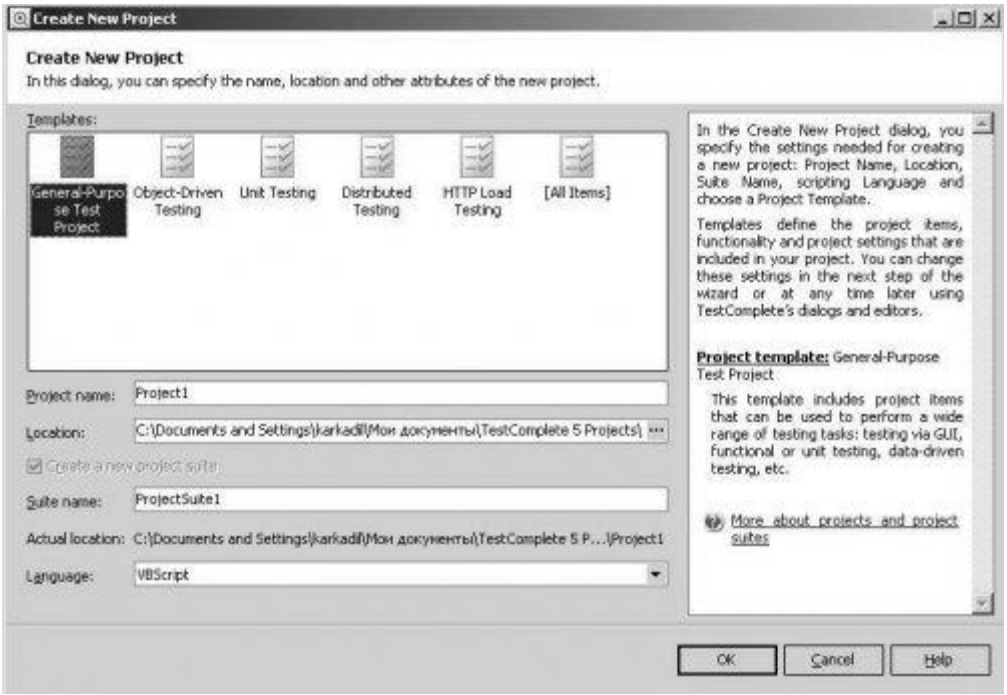
Мы рассмотрим запись и воспроизведение скриптов, чтение логов, работу с Object Browser-ом, выбор языка программирования, а также некоторые другие базовые понятия, которыми необходимо владеть, если вы хотите работать с TestComplete.

Если вы знакомы с этими вопросами, можете смело приступать к чтению главы 3 Основы разработки тестовых скриптов.

2.1 Создание первого проекта, выбор языка программирования

Основным понятием в среде TestComplete, как и во многих средах разработки, является проект (Project). В нем хранятся все данные, которые вы создаете в процессе работы. Проекты объединяются в наборы проектов (Project Suite). Вы можете поместить в один набор проектов, например, несколько проектов, которые предназначены для тестирования определенной функциональности большого проекта, или же все проекты для тестирования какого-то конкретного приложения. В отдельный проект можно также выделить общие функции, константы, которые будут использоваться в других проектах.

Давайте создадим новый проект, выбрав пункт меню File – New – New Project и изучим процесс создания более детально.



В появившемся окне Create New Project находится список шаблонов (templates). При выборе одного из них вы получите готовый шаблон для того или иного вида тестирования (например, при выборе элемента Unit Testing – шаблон для модульного тестирования, HTTP Load Testing – шаблон для нагрузочного тестирования, и т.п.). В данный момент достаточно выбрать первый пункт: General Purpose Test Project (проект общего назначения). Далее необходимо указать имя проекта и его местоположение на диске

(Project name и Location соответственно). Далее в зависимости от того, был ли создан набор проектов ранее, или нет, может оказаться необходимым указать также имя набора проектов (Suite name). Если же набор уже создан, то можно либо оставить существующий, либо, включив опцию “Create a new project suite” (создать новый набор проектов), создать новый набор и включить новый проект в созданный набор.

Далее следует опция Language (Язык), на которой необходимо остановиться подробнее. TestComplete поддерживает 5 скриптовых языков: VBScript, JScript, C++Script, C#Script и DelphiScript. Язык проекта выбирается в момент его создания и не может быть изменен позже. Немного о поддерживаемых языках.

- VBScript – скриптовый язык от Microsoft. Достаточно легок для изучения. На сайте Microsoft доступна как онлайн-версия справки по нему, так и оффлайн-версия.
- JScript – скриптовый язык от Microsoft, аналог языка JavaScript. Как и в случае с VBScript справочную информацию по этому языку можно найти на сайте Microsoft.
- C++Script, C#Script – это аналоги языка JScript, имеют тот же набор функций, что и JScript, различаются лишь синтаксисом.
- DelphiScript – скриптовый язык с синтаксисом, похожим на язык Pascal. Поддержка DelphiScript встроена в TestComplete.

Выбор языка для проекта TestComplete не зависит от того, на чем написано тестируемое приложение. То есть вы можете выбрать, например, язык VBScript для тестирования Delphi-приложения, или язык JScript для тестирования приложения, написанного на Visual C++. Есть лишь одна рекомендация: если вы планируете создать Connected (присоединенное) или Self-Testing (самотестирующееся) приложение, то желательно выбрать язык, совпадающий с языком, на котором написано тестируемое приложение (подробнее об этих приложения см. далее). Если вы знакомы с одним из этих языков, рекомендуем выбрать именно его. Если вы новичок, то рекомендуем для начала попробовать VBScript, так как он наиболее простой в освоении.

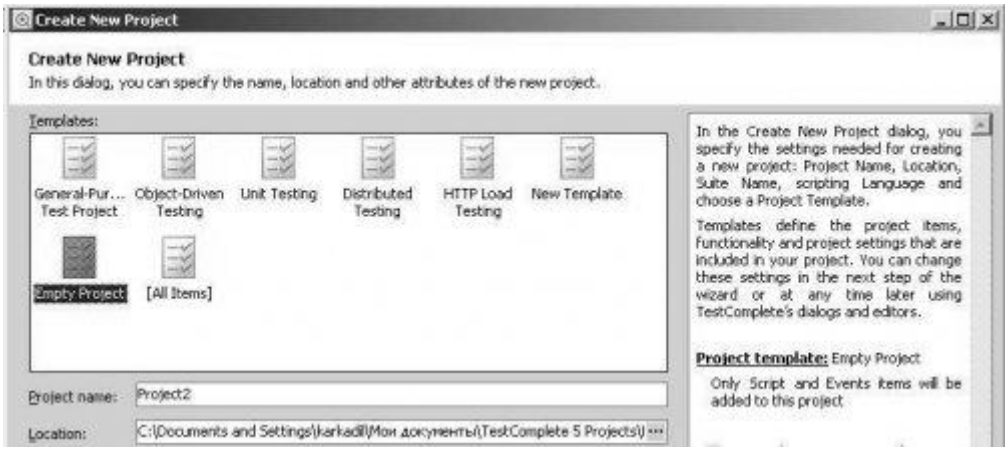
Для нашей книги мы выбрали язык JScript и сделали это по одной простой причине: он наиболее компактен с точки зрения написания кода. В остальном же в любом языке можно написать то же самое, что и в другом, тем или иным способом. Вообще, наиболее правильным было бы приводить примеры на всех языках, как это сделано в справочной системе TestComplete, однако это привело бы к увеличению объема книги.

Итак, определившись с языком, который мы будем использовать, выбираем его из списка Language и нажимаем OK.

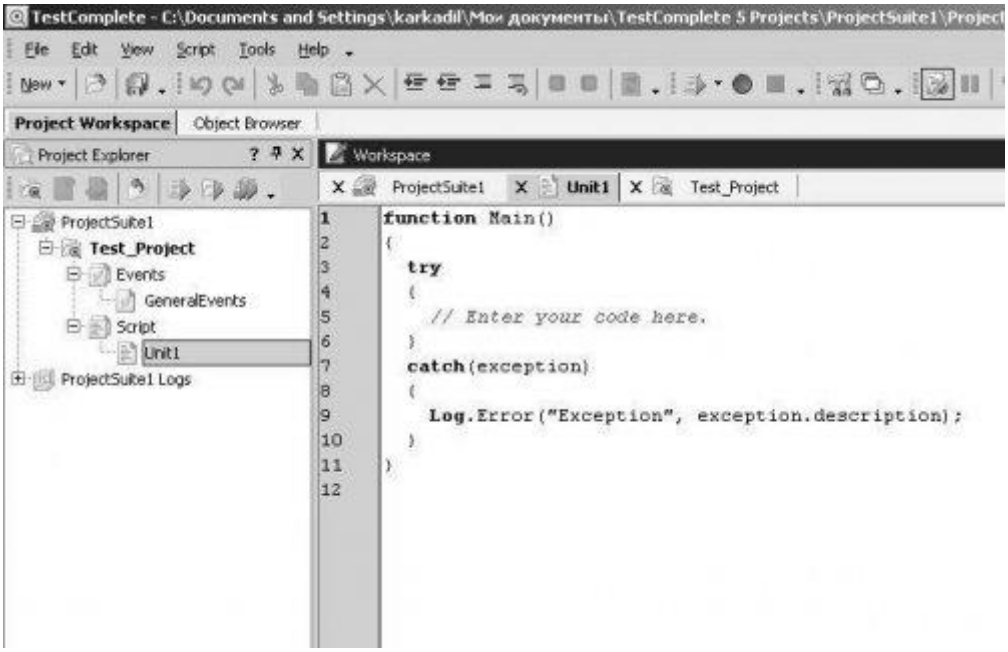
Теперь у нас на экране появляется окно Project Wizard, в котором нам необходимо выбрать нужные нам элементы проекта. Обязательными среди них являются Events (События) и Script (Скрипт).

Отключите все остальные пункты, оставив лишь эти два.

Обратите внимание: ниже списка элементов проекта находятся кнопки Select All, Unselect All и Save As... Первые две из них включают/выключают все элементы проекта, а последняя служит для сохранения текущего вида проекта как шаблона. Например, после отключения всех элементов проекта вы можете сохранить текущий вид как шаблон “Empty Project”, который будет появляться в списке шаблонов при создании нового проекта.



Теперь нажмем кнопку Finish и TestComplete сгенерирует новый проект с теми элементами, которые мы выбрали: Events и Scripts.



Элемент Scripts содержит файлы скриптов, которые вы будете либо записывать автоматически, либо писать вручную. Элемент Events содержит события, которые происходят во время запуска скриптов.

2.2 Запись и воспроизведение скрипта

Работая с TestComplete вы можете создавать скрипты двумя способами: автоматическая запись скрипта средствами TestComplete и ручное написание. Кроме того, можно комбинировать эти два метода. С одной стороны, автоматическая запись скрипта легче, чем написание вручную. Однако при помощи автоматической записи возможно создать лишь простые скрипты, что не всегда является достаточным.

Давайте попробуем записать скрипт в автоматическом режиме и посмотрим, что из этого получится.

Прежде всего необходимо определиться, действия в каком приложении мы будем записывать. В поставке TestComplete есть много приложений, на которых можно

потренироваться (они находятся в папке *"C:\Program Files\Automated QA\TestComplete X\Samples"*; здесь X - это версия TestComplete). Однако приложения, идущие в поставке, необходимо скомпилировать, поэтому для первого знакомства мы ограничимся программой Notepad (Блокнот), идущей в стандартной поставке Windows. Для запуска Блокнота нажмите Пуск - Выполнить, и в открывшемся окне в поле ввода впишите notepad.exe и нажмите ОК.

Для начала записи необходимо выбрать пункт меню Script - Record, или нажать соответствующую кнопку (с изображением красного кружочка) на панели инструментов Test Engine. При этом TestComplete перейдет в режим записи и свернется, на экране в правом верхнем углу появится индикатор (значок TestComplete с подписью Recording), а также панель инструментов Recording.

Далее произведем в Блокноте следующие действия:

- Напишем строку "Test string", затем нажмем Enter и введем текст "Second string"
- Выберем пункт меню Правка – Выделить все, затем Правка – Удалить, затем Правка – отменить
- Щелкнем правой кнопкой мыши по тексту и из контекстного меню выберем пункт "Копировать"

После всего этого нажмем на панели Recording кнопку Stop (с изображением синего квадрата) для остановки записи скрипта и посмотрим на текст скрипта, который сгенерировал для нас TestComplete:

```
function Test1()
{
  var  w1;
  var  w2;
  w1 = Sys.Process("notepad").Window("Notepad", "*");
  w2 = w1.Window("Edit");
  w2.Click(20, 12);
  w2.Keys("Test string[Enter]Second string");
  w1.MainMenu.Click("Edit|Select All");
  w1.MainMenu.Click("Edit|Delete");
  w1.MainMenu.Click("Edit|Undo");
  w2.ClickR(19, 15);
  w2.PopupMenu.Click("Copy");
}
```

Сначала объявляются две переменные w1 и w2. Далее переменной w1 присваивается значение Sys.Process("notepad").Window("Notepad", "*") – это собственно окно Блокнота, а переменной w2 значение w1.Window("Edit") – это текстовое поле, в которое мы вводим текст. Далее следует вызов метода Click() в поле ввода (w2) – это щелчок левой кнопкой мыши для активации этого поля. Затем трижды вызывается метод Click() для трех пунктов главного меню *Правка – Выделить все, Правка – Удалить и Правка – Отменить*. И, наконец, щелчок правой кнопкой мыши ClickR() и выбор пункта контекстного меню Копировать с помощью метода Click() для объекта PopupMenu.

Примечание: в TestComplete версии 7 и выше имена новых переменных более осмысленные и похожи на имена окон, что более удобно при работе с записанным скриптом.

Теперь обратим внимание на объявление главного окна Блокнота. Оно состоит из трех частей:

1. Sys – это глобальный объект, через который вы получает доступ к системе. Например, через этот объект можно получить доступ к объектам OLE (Object Linking and Embedding, Sys.OleObject), к буферу обмена (Sys.Clipboard), информации об операционной системе (Sys.OSInfo), а также ко всем процессам, работающим в данный момент.
2. Для доступа к процессам используется метод Process(), в качестве параметра которому передается имя процесса (в нашем случае это "notepad").
3. Собственно главное окно Блокнота Window("Notepad", "*"). Первый параметр этого метода – это класс окна, второй параметр – его заголовок. Звездочка в имени класса или заголовке служит заменой для любой последовательности символов. В нашем случае это говорит о том, что в заголовке окна Блокнота может быть абсолютно любой текст. Мы можем заменить его на заголовок вида "*Блокнот". Это будет указывать на то, что заголовок окна может начинаться с любой последовательности символов, однако заканчиваться должен обязательно словом "Блокнот".

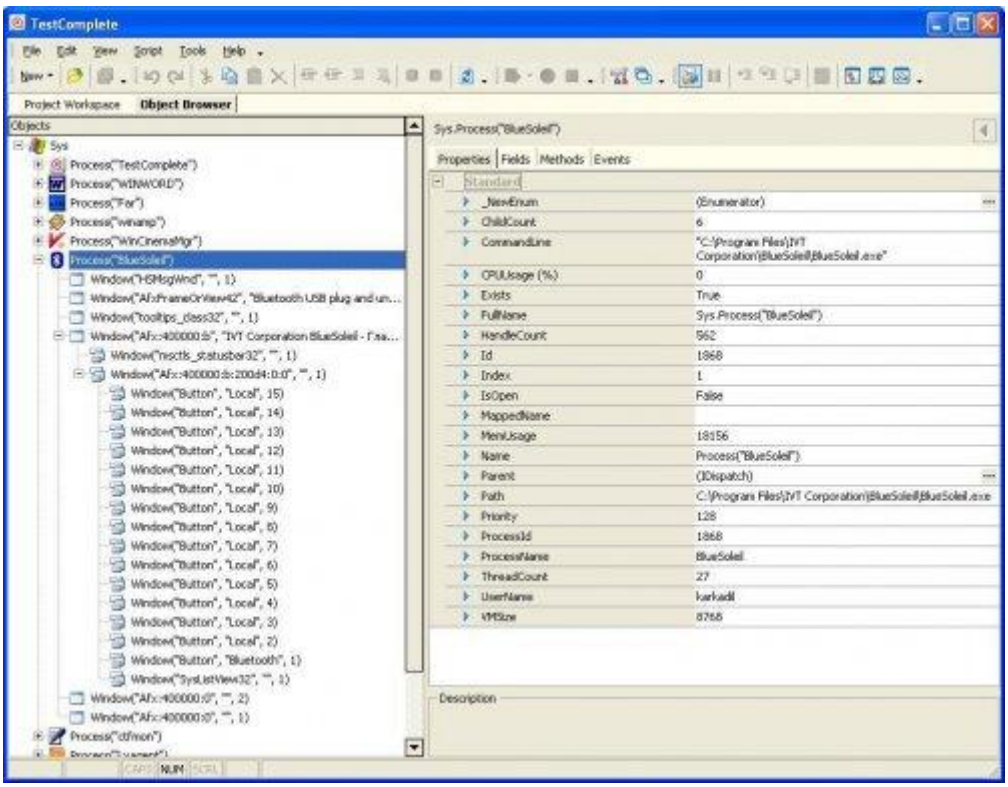
Для запуска записанного скрипта необходимо щелкнуть правой кнопкой мыши на записанной функции и выбрать из выпадающего меню пункт Run Current Routine. При этом TestComplete запустит функцию, которая повторит все записанные действия.

В следующей главе мы более подробно рассмотрим работу с объектом Sys и процессами.

2.3 Использование Браузера Объектов

Панель Object Browser предоставляет информацию о существующих процессах, окнах, а также о действиях, которые TestComplete позволяет делать со всеми этими объектами. Для того чтобы открыть браузер объектов, необходимо выбрать пункт меню *View – Select Panel* и в открывшемся окне Select Panel выбрать элемент Object Browser.

Общий вид браузера показан на рисунке.



Слева находится панель Дерева объектов (Object Tree), справа – панель Свойства объекта (Object Properties). При выборе какого-либо объекта в панели дерева информация в правой части меняется в зависимости от того, какие именно свойства доступны для выбранного объекта.

Значок слева от свойства показывает тип этого свойства: значок в виде стрелочки вправо используется для свойств, которые доступны в режиме «Только для чтения», стрелка вправо используется для свойств, доступных в режиме «Только для записи», для свойств, которые можно считывать и модифицировать, значок не используется.

Если в качестве значения свойства указано IDispatch, это значит, что оно составное и содержит в себе другие свойства. Для их просмотра необходимо дважды щелкнуть левой кнопкой мыши по имени этого свойства. Для возврата обратно к списку свойств родительского объекта достаточно нажать на кнопку со стрелкой влево, которая расположена справа от полного имени свойства в правой верхней части браузера объектов.

В правой части браузера, кроме страницы со свойствами, есть также страницы методов (Methods), полей (Fields) и событий (Events).

Если найти нужный объект в браузере объектов трудно (например, на одном уровне находится много похожих объектов), то можно воспользоваться инструментом Свойства объекта (Object Properties). Для его открытия необходимо щелкнуть правой кнопкой мыши на дереве объектов и выбрать пункт Object Properties в появившемся контекстном меню. При этом окно TestComplete свернется, а вместо него откроется окно Object Properties, похожее на правую часть браузера объектов.

В нижней части этого окна мы видим кнопку с изображением значка цели (Finder Tool). Если щелкнуть по этой кнопке и перетащить ее на интересующий нас объект, то сначала

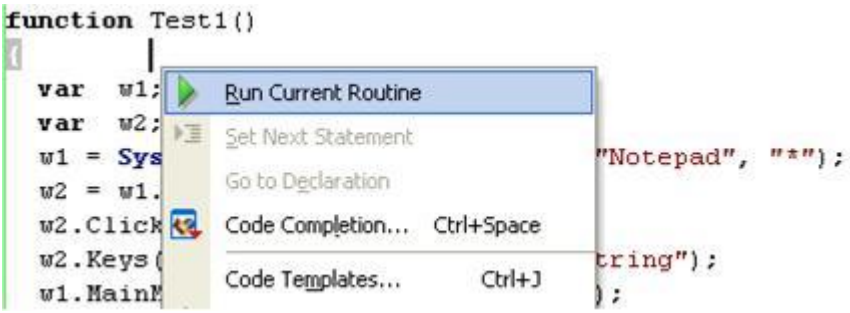
он будет подсвечен красным цветом, а если отпустить кнопку мыши, то в окне Object Properties отобразится список свойств этого объекта.

Кроме того, можно настроить окно Object Properties. Для этого необходимо выбрать пункт меню *Tools - Options* и в открывшемся окне Properties выбрать *Panels – Object Browser*. Если на этой вкладке отключить опцию Capture Mouse, то нам не придется каждый раз перетаскивать кнопку Finder Tool на нужный объект: теперь в окне Object Properties отображаются свойства того объекта, над которым в данный момент находится курсор мыши. Для того, чтобы зафиксировать информацию о текущем объекте в окне Object Properties, необходимо нажать комбинацию клавиш Ctrl-Shift-A.

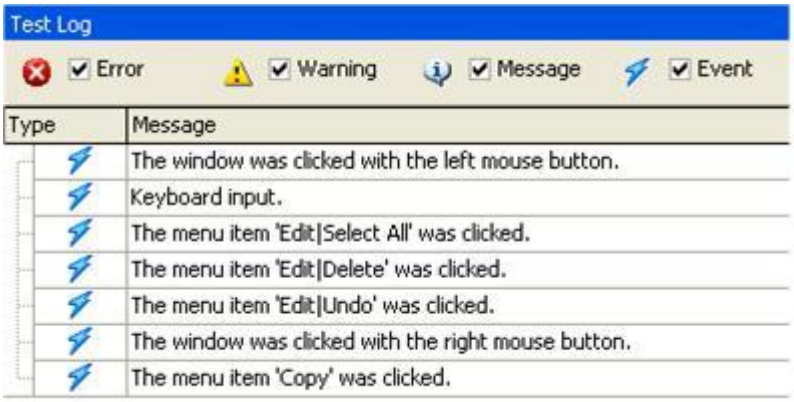
ПРИМЕЧАНИЕ: обратите внимание, что если на вашем компьютере установлена программа ICQ, то данная комбинация клавиш может не работать.

2.4 Запуск скрипта и анализ результатов

Теперь пришло время запустить наш записанный скрипт и посмотреть на результаты. Для запуска скрипта необходимо щелкнуть правой кнопкой мыши на нашей функции и выбрать пункт меню Run Current Routine. Убедитесь, что Блокнот запущен перед запуском скрипта.



После того, как скрипт отработает, TestComplete сформирует лог, в котором будут отображены все произведенные действия, и выведет его на экран.



Если в момент выполнения скрипта произойдут ошибки, то в логе они будут отображены красным цветом. Кроме того, в лог можно записывать свою информацию. Подробнее о работе с логом можно прочитать в разделе 3.8 Использование логов и анализ результатов

2.5 Понятие «Открытое приложение»

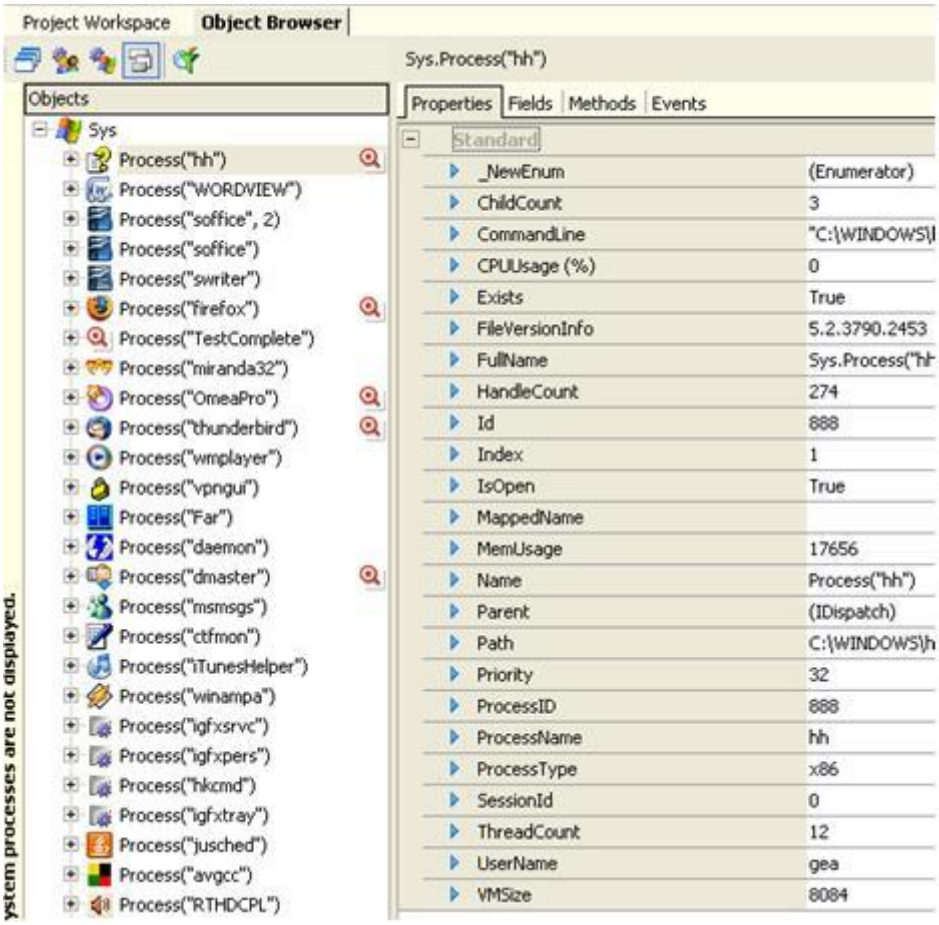
TestComplete позволяет работать практически со всеми типами приложений (Web, Win32, .NET, Delphi, Java и т.п.), однако некоторые приложения придется перекомпилировать

специальным образом, чтобы получить доступ ко всем свойствам и методам. К таким приложениям относятся, например Delphi-приложения (которые необходимо компилировать с включенной Debug информацией и дополнительными модулями, поставляемыми с TestComplete) и Visual C++ приложения, которые необходимо компилировать с включенной Debug информацией.

Приложения, которые не требуют специальной компиляции (например, .NET, Java), а также приложения, скомпилированные нужным образом, называются Открытыми приложениями (Open Applications).
Имейте ввиду, что если ваше приложение требует перекомпиляции, однако сделать это не представляется возможным, вы все равно сможете использовать TestComplete для автоматизации тестирования такого приложения, правда с некоторыми ограничениями.

Отдельного внимания заслуживают Web-приложения. Полный доступ к ним как к открытым приложениям будет только в том случае, если вы используете браузеры Internet Explorer версии 5 или выше (либо любой браузер, созданный на его основе, например MyIE или Avant Browser); Mozilla Firefox версии 1.5.0.1 и выше; Netscape Navigator 8.1.2 (ограниченная поддержка). Для всех остальных браузеров придется ограничиться работой как с не открытым приложением.

Чтобы узнать, распознет ли TestComplete ваше приложение как открытое, необходимо посмотреть на имя процесса в Браузере Объектов. У открытых приложений рядом с именем процесса находится значок с логотипом TestComplete-а, а также свойство IsOpen установлено в True.

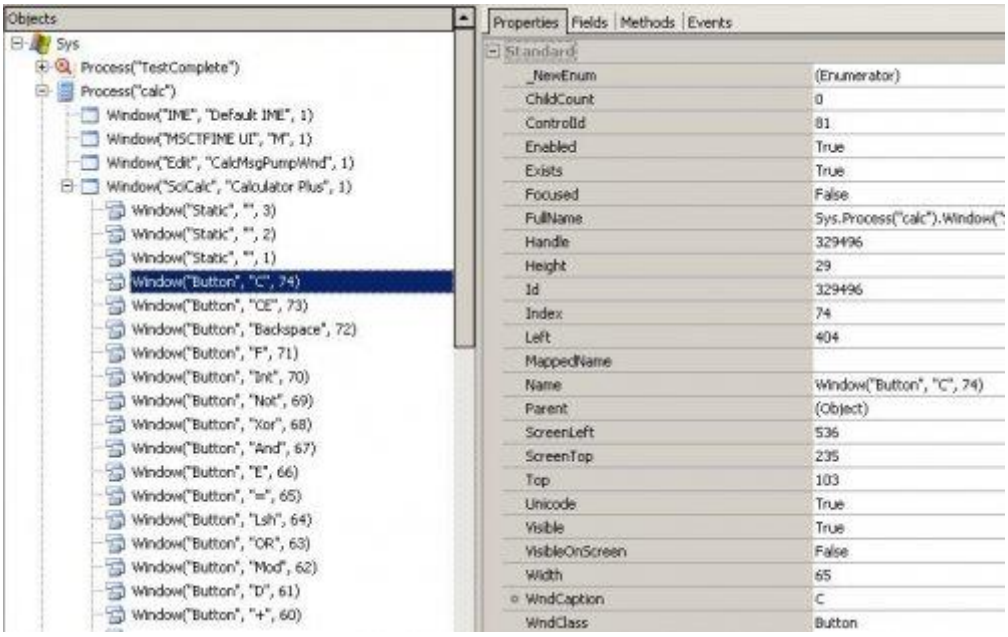


Для более подробной информации о различных типах открытых приложений см. раздел «Open Applications» справочной системы TestComplete-а.

Примечание. В TestComplete версии 7 и выше уже нет понятия "открытое приложение". Все приложения являются "открытыми" сразу и нет необходимости перекомпилировать их с дополнительными файлами (даже если значок TestComplete-а отсутствует в дереве Object Browser-а и свойство IsOpen=false). Однако для того, чтобы иметь доступ ко многим полезным свойствам и методам, необходимо перекомпилировать тестируемое приложение с включенной debug-информацией. Как это сделать – можно прочитать в справке по TestComplete в соответствующих разделах (например, раздел справки " Using Debug Info Agent With Delphi Applications " для Delphi-приложений и т.п.).

2.6 Разные типы приложений

В зависимости от того, на чем написано тестируемое приложение, TestComplete по-разному будет видеть его окна и элементы управления. Для начала возьмем обычное Win32 приложение (Калькулятор) и посмотрим на него через Object Browser:

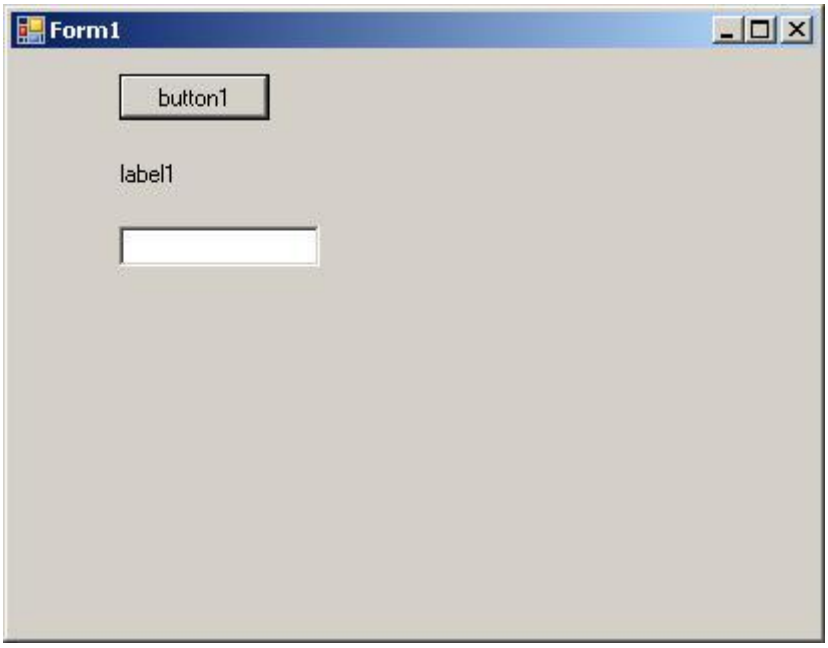


У процесса calc есть несколько объектов Window. Один из этих объектов (Window("SciCalc", "Calculator Plus", 1)) является главным окном программы Калькулятор. У главного окна есть куча дочерних объектов, например:

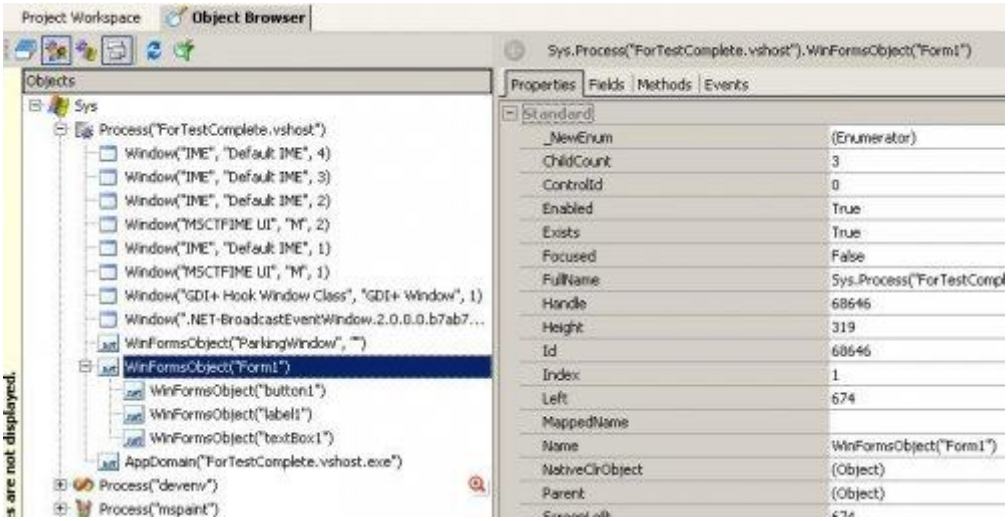
- Window("Static", "", 1) – это статический текст (или label)
- Window("Button", "CE", 73) – это кнопка
- Window("Edit", "", 1) – это текстовое поле

Обратите внимание на то, что все они являются объектами типа Window, различается лишь их класс (Static, Button, Edit).

Теперь возьмем простое .NET приложение, которое выглядит вот так:



и посмотрим на него в Браузере Объектов:



Несмотря на то, что здесь имеется точно такое же главное окно и точно такие же элементы управления (кнопка, статический текст и текстовое поле), для TestComplete они являются совершенно разными!

Для доступа к объектам .NET приложений используется объект WinFormsObject.

Точно так же совершенно разные объекты будут использоваться для Java, Delphi, Web и прочих приложений. Очень важно понимать эту разницу и не пытаться использовать неправильные объекты для работы с разными типами приложений.

В этом учебнике мы чаще всего работаем с обычными Win32 приложениями (Калькулятор, Блокнот и пр.) и поэтому используем объекты Window. Ваше же тестируемое приложение может потребовать работы с другими объектами.

Это особенно важно будет учитывать при чтении главы Синхронизация выполнения скриптов, где используются методы WaitWindow. Для других типов приложений эти методы будут называться иначе (WaitWinFormsObject, WaitVCLObject и т.д.).

Кроме того, при записи скриптов TestComplete может иногда давать краткие имена

элементам управления, тип которых ему известен. В этом случае при чтении скрипта вы не сможете понять, какой тип элемента управления используется, так как обращение к нему будет выглядеть примерно так:

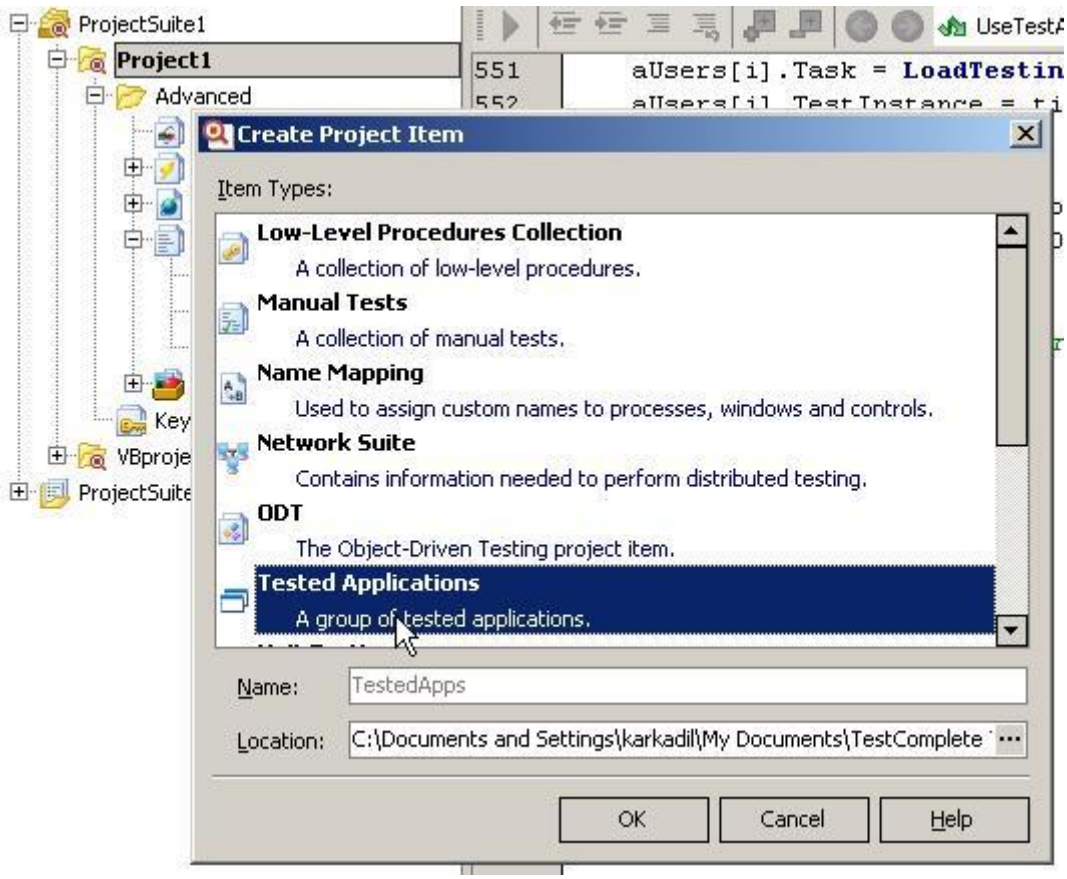
```
Sys.Process(...).frmMain.wndWindow.btnOK.Click()
```

Запись кратких имен можно отключить в настройках *Tools – Options – Engines – General – Object Naming – Use short names when possible.*

2.7 Запуск тестируемого приложения (TestApp)

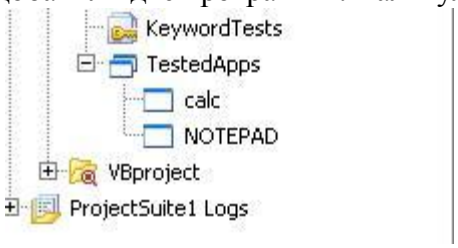
Мы рассмотрели запись и воспроизведение скриптов, используя уже запущенное приложение. В остальных главах мы также будет работать с уже запущенными приложениями, чтобы акцентировать внимание на рассматриваемой теме, а в этой главе узнаем, как запустить тестируемое приложение в TestComplete перед тем, как начинать с ним работать.

Для хранения тестируемых приложений в TestComplete есть специальный объект **TestedApps**. Чтобы добавить его в проект, щелкните правой кнопкой мыши на имени проекта и выберите пункт меню *Add – New Item* и в появившемся окне Create Project Item выберите элемент Tested Applications.



После этого в проекте появится новый элемент – TestedApps, в который вы можете добавлять приложения, которые планируется запускать из скриптов. Для добавления приложения необходимо щелкнуть правой кнопкой мыши на элементе TestedApps, выбрать пункт меню *Add – New Item* и выбрать необходимый файл. В нашем случае мы

добавили две программы: Калькулятор и Блокнот.



Отсюда можно запустить приложение (щелкнув по нему правой кнопкой мыши и выбрав пункт меню *Run*), а также посмотреть и изменить параметры запуска, дважды щелкнув по элементу приложения. При этом появится окно параметров приложений:

Здесь можно изменить путь к приложению, параметры, которые передаются приложению через командную строку и количество запускаемых копий приложения.

Кроме того, здесь можно изменить способ запуска (Run Mode, по умолчанию Simple, т.е. приложение запускается так, как будто пользователь просто запустил приложение) таким образом, чтобы приложение запускалось от имени другого пользователя (например, у другого пользователя могут быть другие привилегии). Теперь если нажать на кнопку с троеточием в колонке Parameters, окно Parameters будет иметь немного другой вид, который позволяет кроме параметров командной строки передать также логин и пароль пользователя, от чьего имени будет запускаться программа.

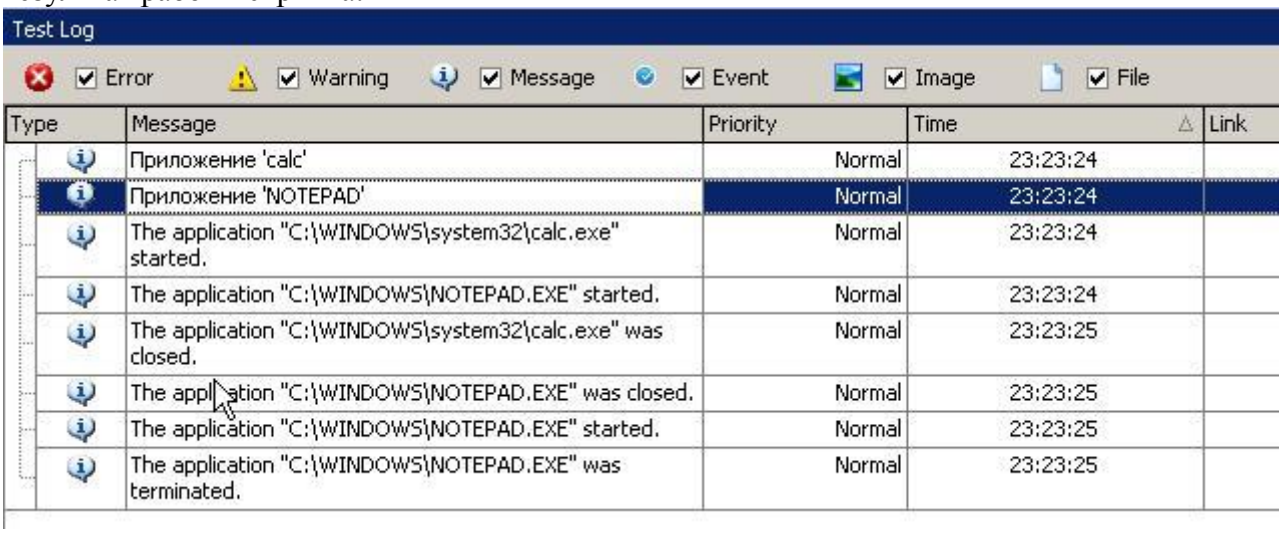
Теперь разберемся, как работать с элементами из TestedApps в скриптах.
Для запуска всех приложений из TestedApps используется метод *RunAll* (при этом запустятся только те приложения, для которых включена опция Launch).

Для доступа к конкретному элементу из списка TestedApps используется свойство Items, куда в качестве параметра надо передать порядковый номер или имя приложения, как оно сохранено в TestedApps. Это свойство возвращает объект типа TestedApp, с которым можно дальше работать (например, получить или изменить его свойства, запустить, закрыть и т.д.).

В качестве примера напишем функцию, которая выводит в лог некоторые параметры приложений из списка TestedApps, а также отрывает и закрывает приложения.

```
function UseTestApp()
{
  var i, oApp;
  for(i = 0; i < TestedApps.Count; i++)
  {
    oApp = TestedApps.Items(i);
    Log.Message("Приложение '" + oApp.ItemName + "'", "Path: " +
oApp.Path + "\nSize (bytes): " + oApp.Size);
  }
  TestedApps.RunAll();
  TestedApps.CloseAll();

  TestedApps.Items("notepad").Run();
  TestedApps.Items("notepad").Terminate();
}
```

Результат работы скрипта:



3 Основы разработки тестовых скриптов

В этой главе рассматриваются темы, без которых невозможно создание хороших тестовых скриптов, а также функциональность, которая может существенно облегчить создание и модификацию тестов в TestComplete.

Вопросы, рассмотренные в некоторых главах (например, Использование фреймворков и Синхронизация выполнения скриптов), актуальны при использовании любого инструмента тестирования, а не только TestComplete-а.

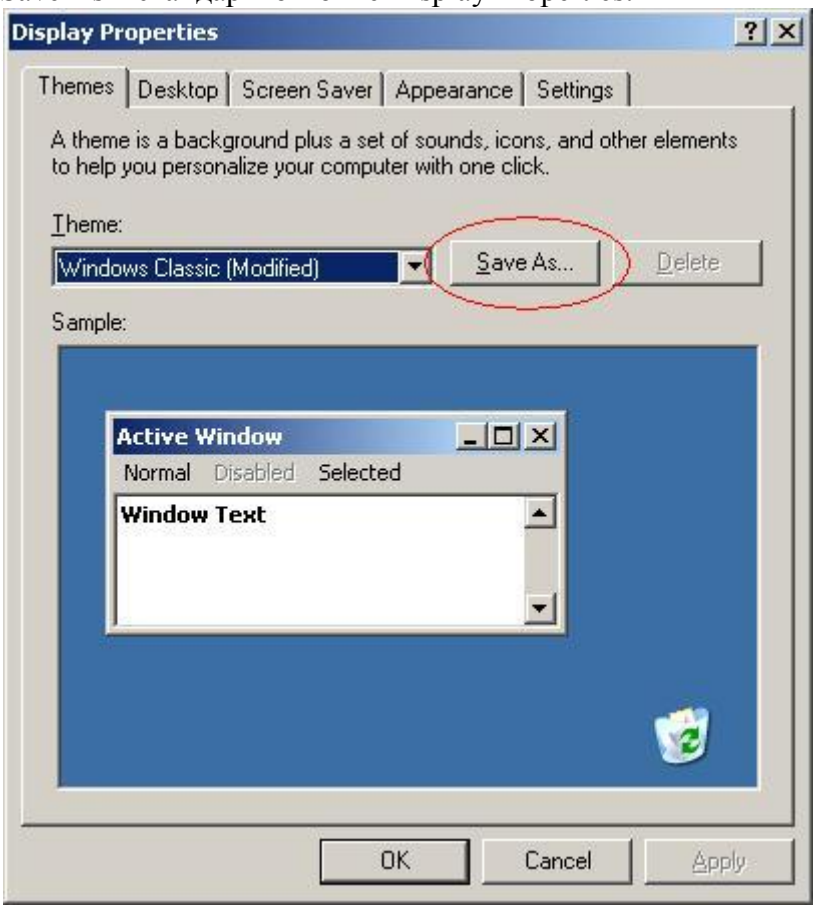
3.1 Выбор модели объектов

Это, пожалуй, один из наиболее важных моментов при работе с TestComplete. В зависимости от того, какую модель представления объектов вы выберете, будет зависеть вся ваша дальнейшая работа.
Элементы управления в любом приложении расположены в строгой иерархии. Например, есть диалоговое окно, внутри него расположен элемент Набор закладок (Tab control), каждая закладка – это также объект, в котором находятся другие элементы управления (кнопки, поля, списки и т.д.). Любой из перечисленных объектов также может быть составным (например, выпадающий список может состоять из двух частей: собственно список и поле ввода).
TestComplete позволяет работать с двумя моделями: Flat и Tree.

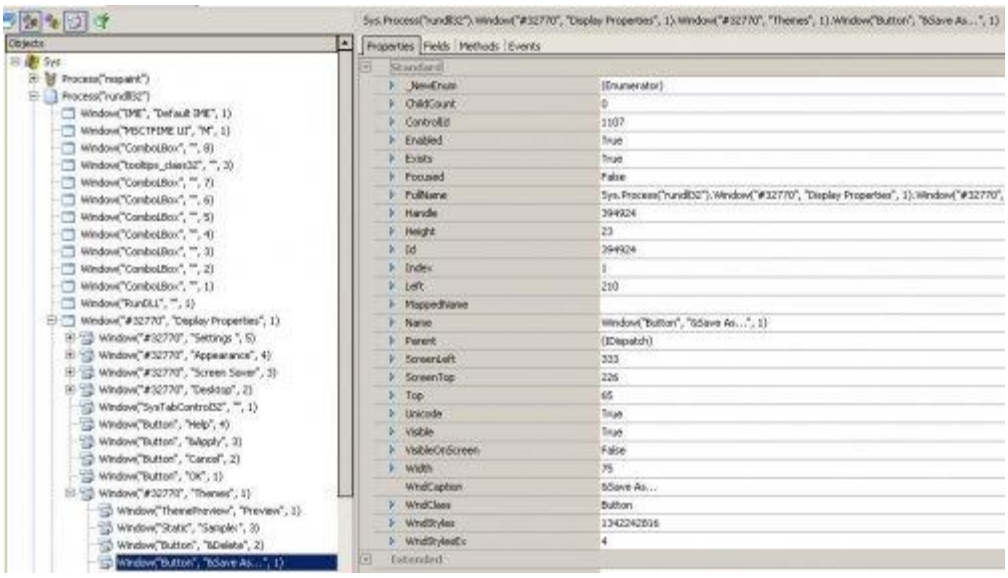
- При использовании Flat модели все элементы управления внутри окна будут находиться на одном уровне независимо от того, как они расположены в приложении. Например, в описанной выше иерархии (окно – набор вкладок – вкладка – поля ввода) и набор вкладок, и каждая вкладка, и каждое поле внутри вкладки будут потомками диалогового окна
- При использовании Tree модели все объекты будут находиться на том же уровне, как они расположены в тестируемом приложении

В качестве примера предлагаем вам посмотреть на то, как TestComplete распознает кнопку

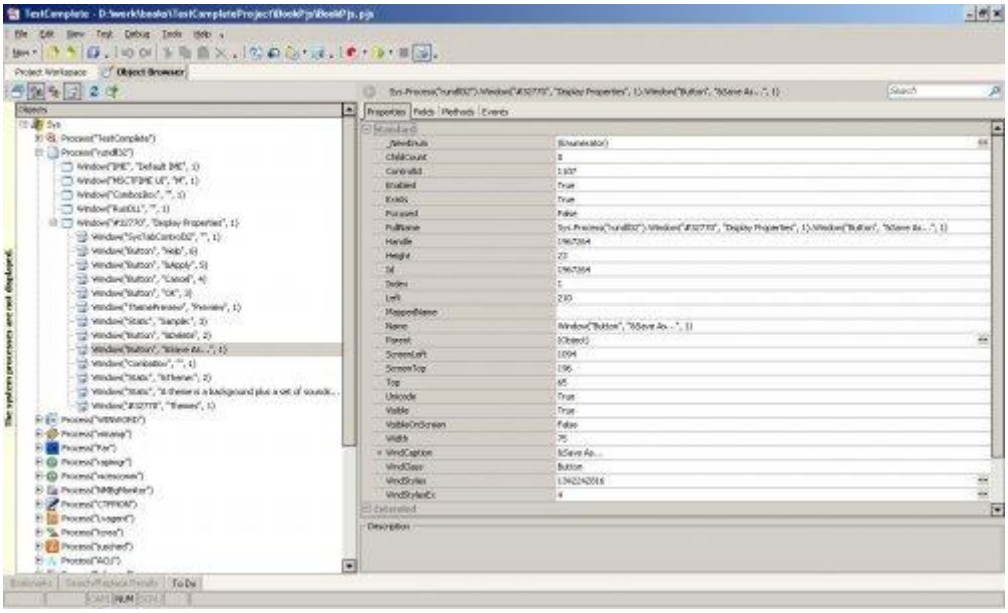
Save As в стандартном окне Display Properties:



Tree model:



Flat model:



В правом верхнем углу находится полное имя объекта. Из этих примеров видно, что доступ к кнопке во втором случае будет более простым, так как во втором случае кнопка Save As является прямым потомком окна Display Properties, а в первом случае у нас имеется дополнительное окно Themes.

У обеих моделей есть свои достоинства и недостатки, а также рекомендации по использованию.

Модель Flat является более простой. В случае ее использования все элементы управления окна являются непосредственными потомками окна, независимо от того, как устроено приложение. Эту модель обычно рекомендуется использовать для небольших и средних приложений, где количество элементов управления сравнительно невелико.

Модель Tree фактически повторяет структуру приложения и является более «точной». Однако при ее использовании возникают дополнительные проблемы: зачастую внутреннее устройство приложения очень сложное и уровни иерархии могут достигать нескольких десятков! Обычно Tree модель рекомендуют использовать для больших приложений, где количество элементов управления очень большое и могут встречаться элементы с одинаковыми именами.

На практике мы не сталкивались с приложениями, в которых использование Tree модели было бы оправдано. Те немногие недостатки, которые появляются в случае использования модели Flat, обычно решаются достаточно легко, а доступ к элементам управления несравнимо более простой и удобный.

В любом случае, мы рекомендуем вам попробовать самим использование обеих моделей на вашем приложении и самим решить, какую модель использовать.

Если вы остановили свой выбор на модели Tree, вам стоит обязательно ознакомиться с главой «Использование пространства имен (Namespaces) и псевдонимов (Aliases)», где вы сможете узнать, как упростить доступ к элементам управления и повысить читабельность ваших скриптов.

Если вы тестируете веб-приложение, то изменение модели объектов – это не все, что вам

нужно сделать. Обратитесь к главе 4.1 Функциональное тестирование Web-приложений чтобы узнать больше о настройках проекта для тестирования веб-приложений.

3.2 Использование стандартов именования

Независимо от того, какое средство разработки вы используете, вам необходимо использовать стандарты именования (или стандарты кодирования, Coding Standards). Это необходимо как при работе в команде, так и в том случае, если вы – единственный разработчик тестовых скриптов, так как после вас кому-то, возможно, придется работать с этими скриптами.

Стандарты можно как разработать самому, так и найти готовые решения в интернете.

В стандарты кодирования обычно входит всё, что касается кода: имена переменных и функций, оформление комментариев, отступы в коде, именование модулей и т.д. В случае ТестКомплита необходимо также определиться, какой язык программирования используется, как проекты взаимодействуют между собой (есть ли общие проекты, в которых хранятся функции, которые можно вызывать из других проектов), какой формат файлов используется для хранения изображений, какая объектная модель используется, какие настройки параметров воспроизведения должны быть указаны и т.п.

В качестве примера мы разработали простые стандарты кодирования, которым придерживаемся в этом учебнике. С этими стандартами можно ознакомиться в приложении Стандарты кодирования.

3.3 Запись, модификация и написание скриптов

Средства записи скриптов, которыми обладают все современные средства автоматизации, обычно недостаточны для создания качественных скриптов, которые будет удобно модифицировать и поддерживать в дальнейшем.

Обычно средства записи полезны в самом начале, на этапе изучения инструмента, чтобы понять, как именно происходит работа с элементами управления в скриптах. Затем, по мере накопления опыта, вы начнете изменять скрипты таким образом, чтобы они работали стабильнее или были более читабельны. Часть кода будет выноситься в отдельные функции для дальнейшего использования, часть будет перемещаться в циклы, будут добавляться условные операторы и т.п. Зачастую приходится писать довольно сложные функции для дальнейшего использования (например, при работе с базами данных). В конце-концов весь код будет писаться вручную и это будет занимать меньше времени, чем запись и последующая модификация записанного скрипта.

Давайте в качестве примера сделаем такое упражнение. Предположим, нам необходимо пять раз открыть и закрыть окно справки в Блокноте. Вот как такой скрипт будет выглядеть при записи:

```
function Test1()

{

  var  notepad;

  var  wndNotepad;
```

```
  notepad = Sys.Process("notepad");

  wndNotepad = notepad.Window("Notepad", "*");

  wndNotepad.MainMenu.Click("Help|About Notepad");

  notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

  wndNotepad.MainMenu.Click("Help|About Notepad");

  notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

  wndNotepad.MainMenu.Click("Help|About Notepad");

  notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

  wndNotepad.MainMenu.Click("Help|About Notepad");

  notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

  wndNotepad.MainMenu.Click("Help|About Notepad");

  notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

}
```

А что делать, если в дальнейшем нам нужно будет открывать окно справки не пять, а пятьдесят раз? Снова перезаписывать скрипт или скопировать имеющийся участок кода 10 раз? Естественно, такой подход является неправильным! Вот как будет выглядеть измененный скрипт:

```
function Test1()

{

  var  notepad;

  var  wndNotepad;

  var i;

  notepad = Sys.Process("notepad");

  wndNotepad = notepad.Window("Notepad", "*");



  for(i = 0; i < 5; i++)

  {

    wndNotepad.MainMenu.Click("Help|About Notepad");
```

```
    notepad.Window("#32770", "About Notepad").Window("Button",
"OK").ClickButton();

  }

}
```

Теперь если нам понадобится открыть окно справки 50 раз, нам необходимо лишь изменить одну цифру в цикле.

Это лишь очень простой пример того, как запись усложняет дальнейшую работу со скриптами. В настоящих проектах все может быть гораздо сложнее.

Кроме того, с помощью записи можно создавать лишь очень простые скрипты. Если вы собираетесь создавать набор хорошо спроектированных скриптов, которые будут постепенно дополняться, то вам придется использовать какой-либо фреймворк. А в случае использования фреймворков возможность записи скриптов обычно отпадает, так как скрипты имеют более сложную структуру, чем просто набор действий с приложением.

3.4 Использование именования (NameMapping) и псевдонимов (Aliases)

Именования (NameMapping) и псевдонимы (Aliases) используются для более удобного доступа к элементам управления в скриптах. Сравните, например, два способа нажатия на кнопку CE в Калькуляторе.
Обычный способ:

```
Sys.Process("calc").Window("SciCalc", "Calculator
Plus").Window("Button", "CE").ClickButton();
```

С использованием Name Mapping:

```
NameMapping.Sys.calc.wndSciCalc.btnCE.ClickButton();
```

И с использованием Aliases:

```
Aliases.calc.wndSciCalc.btnCE.ClickButton();
```

Во всех трех случаях мы делаем одно и то же действие: нажимаем на кнопку CE калькулятора. Однако в первом случае нам необходимо написать гораздо больше кода, чем во втором и, тем более, в третьем. В нашем примере мы экономим не так уж и много места, однако представьте себе, насколько меньше кода придется писать для тестирования всего приложения, и насколько проще читать и понимать код, приведенный во втором и третьем случаях!

Дело в том, что вы первом случае мы используем обычный способ доступа к элементам управления в приложении. Именно так, как видит их ТестКомплит. Мы указываем, какой тип приложения используется (Window), какой заголовок окна, с которым мы работаем (Calculator Plus), какой тип элемента управления, на который мы нажимаем (Button) и его заголовок (CE).

NameMapping позволяет нам создать псевдонимы для любого элемента любого уровня вложенности. Например, мы можем переименовать объект Sys в S, и тогда нажатие на кнопку будет выглядеть так:

```
NameMapping.S.calc.wndSciCalc.btnCE.ClickButton();
```

Мы также можем переименовать остальные объекты в нашем приложении и, в конце концов, добиться следующего обращения к кнопке:

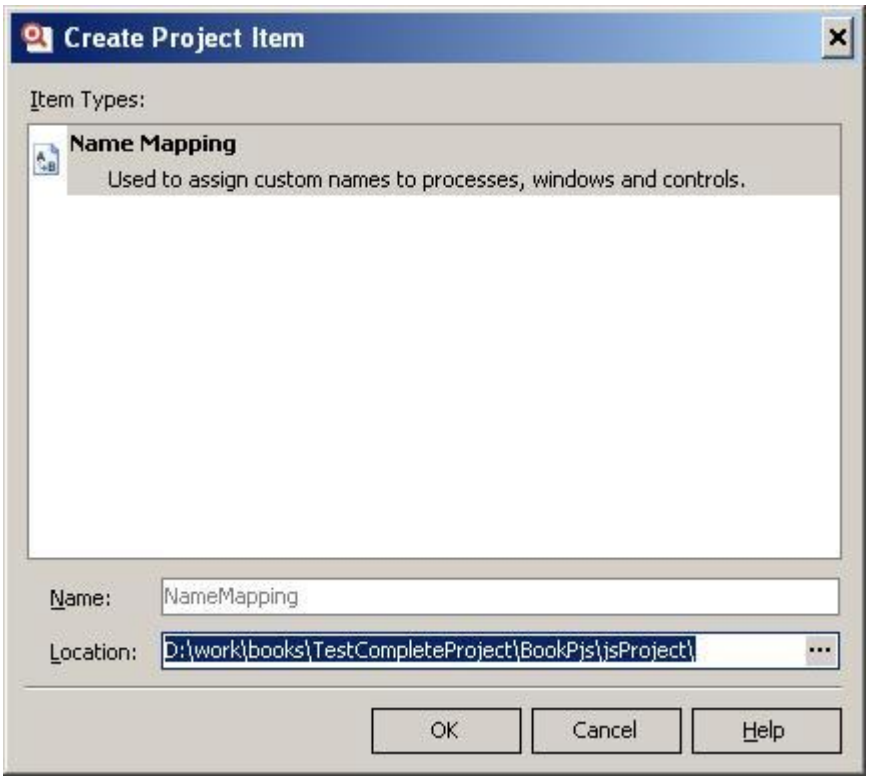
```
NameMapping.S.calc.Calc.CE.Click();
```

Однако, как бы мы не переименовывали объекты в NameMapping, мы так и остаемся привязаны к структуре приложения. То есть мы можем как угодно изменять имена объектов в NameMapping, но иерархия остается такой же, какая она есть в приложении. Это не всегда удобно, поэтому в подобных случаях нам на помощь приходят Aliases. Aliases – это тот же NameMapping, который, однако, позволяет установить собственную иерархию объектов. Например, в нашем случае мы можем сделать так, чтобы кнопка CE стала прямым потомком процесса calc (это совершенно бессмысленное и даже вредное действие, однако мы приводим этот пример для того, чтобы показать возможности Aliases):

```
Aliases.calc.btnCE.ClickButton();
```

Теперь рассмотрим процесс создания NameMapping и Aliases.

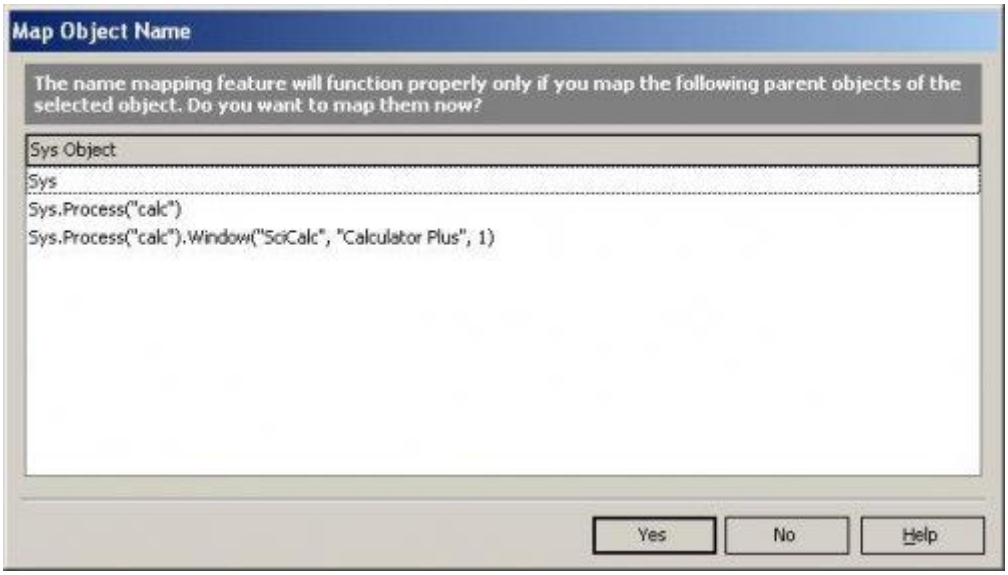
Если вы используете ТестКомплит версии 7 и выше, то при записи скрипта ТестКомплит создаст и предложит сохранить созданный NameMapping.



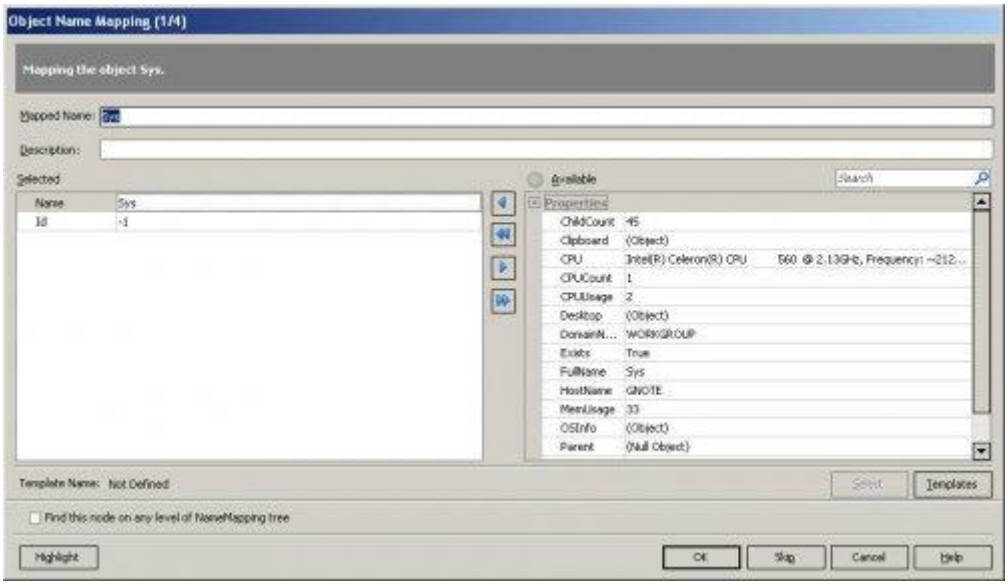
Если же у вас ТестКомплит более ранней версии, то вам придется создавать NameMapping самим.

Для этого прежде всего необходимо добавить элемент NameMapping в проект. Щелчок правой кнопкой мыши на имени проекта, затем *Add – New Item – Name Mapping*. После этого выбираете необходимый вам объект в Object Browser, щелкаете на нем правой кнопкой мыши и в появившемся контекстном меню выбираете пункт Map the Object

Name. Если один или несколько родительских объектов не были примапплены, ТестКомплит выдаст вам сообщение



Если в этом окне нажать кнопку Yes, вам необходимо будет именовать все неименованные объекты приложения по предложенным или выбранным вами свойствам.



Обратите внимание на список Selected слева. Именно по этим свойствам и их значениям ТестКомплит будет в дальнейшем распознавать объекты приложения, поэтому здесь необходимо тщательно продумывать, по каким именно свойствам вы хотите идентифицировать объект. Набор свойств должен быть уникальным для данного элемента управления.

По мере того, как вы добавляете объекты в NameMapping, они появляются и в разделе Aliases. Изменять порядок объектов иерархии в списке Aliases можно перетаскивая объекты мышью.

Запомните: можно использовать NameMapping без Aliases, однако Aliases могут быть использованы только в том случае, если у вас имеется NameMapping!

Если вы не хотите использовать NameMapping в своих скриптах и не хотите, чтобы окно NameMapping появлялось каждый раз, когда вы записываете скрипт, вам необходимо

отключить эту опцию в *Tools – Options – Engines – Name Mapping*, чекбокс *Map object names automatically*.

3.5 Синхронизация выполнения скриптов

При работе практически с любым приложением иногда возникают ситуации, которые могут выполняться разное время в зависимости от внешних факторов (скорость сети, объем передаваемых данных, быстродействие компьютера и т.п.).

Например, вы можете записать скрипт, в котором при нажатии на какую-то кнопку выполняется выборка данных из базы с небольшим количеством записей. После того, как выборка сделана, появляется окно с результатами поиска. Допустим, что на выборку было потрачено 5 секунд. После этого этот же скрипт запускается на другой базе, с гораздо большим количеством записей, и процесс выборки занимает 20 секунд. TestComplete не может знать о том, что используется другая база, и через 5 секунд, не дождавшись появления окна (как это произошло при записи), он сообщит об ошибке и прекратит выполнение скрипта (или попытается продолжить выполнение скрипта дальше, в зависимости от настроек TestComplete, однако так как окно не появилось, работать с ним некак, а это, в свою очередь, вызовет новое сообщение об ошибке).

Возможна и другая ситуация. Предположим, что у вас в окне есть десяток элементов управления с почти одинаковыми свойствами (например, 10 кнопок без текста), однако только одна из них доступна (Enabled, т.е. может быть нажата). Мы ничего не знаем о том, какая именно из этих 10ти кнопок доступна, но нажать нам надо именно на нее. При первом запуске приложения (и записи скрипта) может быть доступна первая кнопка, а при следующем запуске – пятая или седьмая. Каким образом нам выбрать именно ту кнопку, которая нужна нам в момент выполнения скрипта?

Для решения подобных проблем в TestComplete есть несколько методов:

- Wait... методы окон и элементов управления (WaitWindow, WaitVCLWindow, WaitWinFormsObject, WaitVBObject и т.п.). Обратите внимание, что методы Wait существуют для любых объектов (например, объекту Panel соответствует методWaitPanel, объекту Button – метод WaitButton и т.д.)
- Wait... методы для свойств и дочерних элементов окон (WaitChild, WaitProperty)
- Методы для поиска объектов (FindChild, FindAll, FindAllChildren, FindId)

Правильно комбинируя использование этих методов можно добиться практически идеальных запусков скриптов, однако использовать эти методы всегда и везде не очень разумно, так как в некоторых случаях их работа может быть довольно долгой (например, если в окне очень много элементов управления, то метод FindAllChildren будет работать долго).

Теперь рассмотрим несколько примеров использования этих методов на примере тестирования Калькулятора. Для начала запишем скрипт, который выбирает в Калькуляторе пункт меню Help – About и затем закрывает окно About.

```
function Test1()
{
  var  calc;
  calc = Sys.Process("calc");
```

```
  calc.Window("SciCalc", "Calculator
Plus").MainMenu.Click("Help|About Calculator Plus");
  calc.Window("#32770", "About Calculator Plus").Close();
}
```

Теперь предположим, что окно About появляется не мгновенно, а может появиться как через секунду, так и через 5 или 7 секунд и нам необходимо как-то ждать его появления. Для этого мы воспользуемся методом WaitWindow.

```
function Test1()
{
  var  calc;
  calc = Sys.Process("calc");
  calc.Window("SciCalc", "Calculator
Plus").MainMenu.Click("Help|About Calculator Plus");

  if( !calc.WaitWindow("#32770", "About Calculator Plus", -1,
10000).Exists)
  {
    Log.Error("Окно About не открылось!");
  }
  else
  {
    calc.Window("#32770", "About Calculator Plus").Close();
  }
}
```

Теперь в случае если окно не откроется, в лог будет выведено сообщение об ошибке, а если откроется – оно будет закрыто. Давайте подробнее разберем теперь внесенные изменения.

Объекту Window мы передавали 2 параметра: класс окна (“#32770”) и его заголовок (“About Calculator Plus”). В метод WaitWindow, в дополнение к этим двум параметрам, мы передаем еще два: порядковый номер окна (или его позиция, так называемый Z-порядок) и таймаут в миллисекундах (то есть время ожидания окна). В нашем случае порядковый номер равен -1 (если номер окна меньше единицы, то он просто игнорируется TestComplete-ом), а время ожидания равно 10 тысячам миллисекунд, т.е. 10ти секундам.

Метод WaitWindow возвращает объект, аналогичный объекту Window. У этого объекта есть свойства и методы, которые можно использовать, и в данном случае мы как раз воспользовались одним из свойств – Exists, который возвращает true, если объект существует, и false в противном случае.

Таким образом, если в течение 10ти секунд окно About не появится, то отработает первый блок IFa, в котором выдается сообщение об ошибке, а если окно появится – то отработает второй блок IFa и окно About будет закрыто.

Обратите внимание на некоторые особенности, которые обычно вызывают сложности у начинающих:

1. Метод WaitWindow возвращает объект, а не проверяет, существует ли окно. То есть, следующий код будет неправильным:
   if(!calc.WaitWindow("#32770", "About Calculator Plus", -1, 10000))

Даже если окно не существует, метод WaitWindow все равно вернет объект, у которого необходимо проверить свойство Exists

2. Вместо метода WaitWindow для проверки существования окна можно использовать и объект Window (if(!calc.Window("#32770", "About Calculator Plus").Exists)). Однако в этом случае если окно не откроется, TestComplete выдаст в лог сообщение об ошибке (“Window not found”), что не всегда бывает удобно, если необходимо только проверить наличие окна и в зависимости от результата выполнить разные действия
3. Не используйте метод WaitWindow для объектов другого типа. Для каждого типа элементов управления есть соответствующие методы Wait (например, WaitWinFormsObject для .NET элементов)
4. Хотя вы и можете использовать объекты, которые возвращают методы Wait, для дальнейшей работы, на практике их лучше использовать только для проверки существования. Например, следующий код вполне работоспособен:

```
var obj = calc.WaitWindow("#32770", "About Calculator Plus", -1, 10000);
if( obj.Exists)
  {
       obj.Activate();
obj.Close();
  }
```

Вместо этого лучше заново инициализировать объект и дальше с ним работать. То есть пример выше лучше переписать так:

```
if (calc.WaitWindow("#32770", "About Calculator Plus", -1, 10000).Exists)
{
var obj = calc.Window("#32770", "About Calculator Plus");
obj.Activate();
obj.Close();
}
```

При использовании первого варианта TestComplete может иногда «терять» объект, возвращенный методом Wait и выводить в лог сообщение об ошибке (Window not found). Подобное поведение, собственно говоря, не является нормальным, однако выявить его причину практически невозможно, а потому приходится просто мириться с этой особенностью.

Объект Window по умолчанию ждет появления окна в течение 10ти секунд. Чтобы изменить этот параметр, необходимо щелкнуть правой кнопкой мыши на имени проекта и выбрать пункт меню *Edit – Properties*, затем в открывшейся панели выбрать элемент Playback и установить необходимый таймаут в поле Auto-wait timeout, ms.

Теперь рассмотрим следующий пункт раздела синхронизации: методы WaitChild и WaitProperty. Метод WaitChild используется для ожидания появления какого-либо дочернего объекта у того элемента управления, для которого он вызывается.

Метод WaitChild принимает 2 параметра:

- имя объекта, который следует ждать (его можно скопировать из Object Browser-а, выделив объект в дереве объектов и скопировав значение поля Name в правой части)
- таймаут в миллисекундах, сколько ожидать объект

В качестве примера возьмем все тот же Калькулятор и напишем скрипт, который будет бесконечно ожидать появления окна About. Для того, чтобы проверить правильность

работы скрипта, запустите этот пример и TestComplete будет работать, постоянно ожидая появления окна About. Чтобы прекратить выполнение скрипта, выберите в Калькуляторе пункт *Help – About.*

```
function TestWaitChild()
{
  var  calc;
  calc = Sys.Process("calc");
  calc.Window("SciCalc", "Calculator Plus").Activate();
  while(true)
  {
    if(calc.WaitChild("Window(\"#32770\", \"About Calculator
Plus\", 1)", 1).Exists)
    {
      Log.Message("Окно About открылось!");
      break;
    }
    aqUtils.Delay(100);
  }
}
```

Обратите внимание, что бесконечные циклы вида while(true) не рекомендуется использовать при написании рабочих скриптов, так как подобный цикл может навсегда завесить компьютер. Мы используем подобные примеры только лишь для упрощения примеров кода.

Кроме того, если вы делаете подобный цикл, в нем рекомендуется вставлять задержку (мы воспользовались для этого методом Delay объекта Utils), чтобы не загружать процессор на 100% и не мешать выполнению других программ.

В имени объекта можно использовать символы групповой автозамены (? для одного символа и * для нескольких символов). Теперь строку ожидания объекта можно существенно упростить:

```
calc.WaitChild("*About Calculator Plus*", 1).Exists
```

Однако при использовании подобных приемов нужно быть осторожным, иначе можно слишком сильно упростить имя ожидаемого объекта и дождаться совсем другого элемента управления. Например, может оказаться так, что в окне будет кнопка с надписью «Help» и при ее нажатии будет открываться окно «Help». В таком случае нельзя будет просто сократить имя так, как мы сделали в нашем примере, иначе скрипт может «найти» кнопку вместо окна.

Если вы используете в скриптах NameMapping и Aliases, вам понадобится использовать методы WaitNamedChild и WaitAliasChild соответственно.

Метод WaitProperty по принципу работы похож на WaitChild, однако вместо ожидания появления объекта он ожидает, пока какое-то конкретное свойство объекта не станет равным заданному значению. Например, если у нас есть невидимый объект (т.е. его можно увидеть в Object Browser-е, но не в приложении), то можно ожидать, пока его свойство VisibleOnScreen не станет равным True. Или можно ждать, пока в текстовом поле не появится какой-то текст. Вот пример скрипта, который ожидает появления значения «123» в поле Калькулятора. Как и в предыдущем примере, чтобы прекратить выполнение

скрипта, просто введите в Калькуляторе цифры 123 и TestComplete прекратит работу.

```
function TestWaitProperty()
{
  var  calc;
  calc = Sys.Process("calc");
  calc.Window("SciCalc", "Calculator Plus").Activate();
  var edit = calc.Window("SciCalc", "Calculator Plus",
1).Window("Edit", "", 1);
  while(true)
  {
    if(edit.WaitProperty("wText", "123. "))
    {
      Log.Message("Текст '123' введен!");
      break;
    }
    aqUtils.Delay(100);
  }
}
```

И последняя группа функций, использующихся для синхронизации скриптов, функции Find... . В некотором роде эти методы самые сложные в использовании, так как работа с ними осуществляется по-разному в разных языках программирования. Это связано с тем, что методы Find... работают с массивами, а в языках VBScript и Jscript используются разные типы массивов.
Принцип работы всех этих методов одинаков: вы передаете два массива (один с именами свойств, второй со значениями этих свойств), а также в некоторых случаях дополнительные параметры, а TestComplete по этим наборам свойств и значений находит соответствующие объекты в приложении.

Сначала рассмотрим простой пример. Допустим, нам нужно в Калькуляторе найти кнопку C (Clear), которая обнуляет текстовое поле результатов вычисления.

Проблема в том, что у нас две таких кнопки: вторая кнопка с текстом C находится в ряде кнопок внизу, которые предназначены для шестнадцатеричных вычислений. Так как в данный момент вторая кнопка C недоступна (disabled), мы можем найти нужную нам

кнопку C (Clear) по двум свойствам: текст = “C” и свойство Enabled = true.

Вот как поиск будет выглядеть на языке VBScript:

```
Sub TestFindChild

  Dim PropArray, ValuesArray, btn, wnd

  PropArray = Array("WndCaption", "Enabled")
  ValuesArray = Array("C", True)

  Set wnd = Sys.Process("calc").Window("SciCalc", "Calculator
Plus")
  Set btn = wnd.FindChild(PropArray, ValuesArray, 5)

  wnd.Activate

  If btn.Exists Then
    btn.Click
  Else
    Log.Error "Кнопка 'C' не найдена!"
  End If

End Sub
```

Как видите, ничего сложного. Если хотите проверить, будет ли эта процедура работать когда кнопки C действительно нету, просто замените во втором массиве ValuesArray значение C на F (кнока с заголовком F в калькуляторе только одна и она недоступна) и снова запустите скрипт.

В случае использования языка DelphiScript код будет таким же простым, так как массивы в DelphiScript аналогичны массивам VBScript. Однако в случае использования языка Jscript (а также C++Script и C#Script) нам придется сначала конвертировать массивы Jscript в формат VBArray. Для этого воспользуемся функцией ConvertJScriptArray, которую можно найти в справочной системе TestComplete:

```
function ConvertJScriptArray(JScriptArray)
{
  // Uses the Dictionary object to convert a JScript array
  var objDict = Sys["OleObject"]("Scripting.Dictionary");
  objDict["RemoveAll"]();
  for (var i in JScriptArray)
    objDict["Add"](i, JScriptArray[i]);
  return objDict["Items"]();
}
```

Теперь напишем JScript функцию, которая, как и пример выше, будет искать доступную снопку C и кликать по ней:

```
function TestFindChild()
{
  var  wnd;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate()

  var PropArray = ConvertJScriptArray(new Array("WndCaption", "Enabled"));
  var ValuesArray = ConvertJScriptArray(new Array("C", true));
```

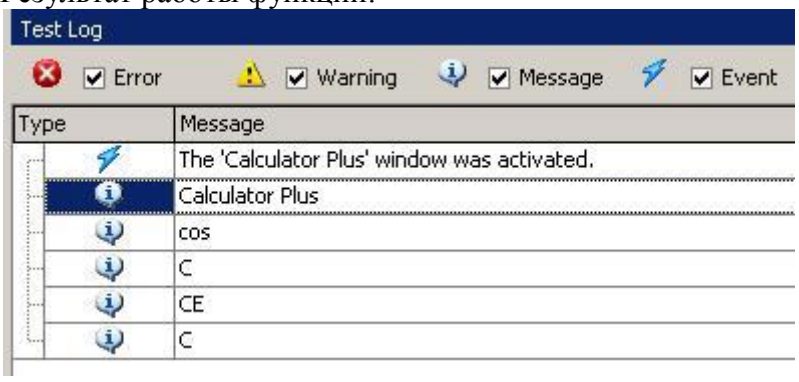
```
  var btn = wnd.FindChild(PropArray, ValuesArray);

  if(btn.Exists)
    btn.Click();
  else
    Log.Error("Кнопка C не найдена!");
}
```

Метод FindChild предназначен для поиска одного объекта и возвращает первый найденный объект, который удовлетворяет условиям поиска (свойствам и их значениям). Если же нужно найти несколько объектов, то для этого можно использовать методы FindAll и FindAllChildren. Они практически идентичны, разница лишь в том, что метод FindAllChildren перебирает только дочерние объекты, а метод FindAll проверяет также тот объект, для которого вызван метод. Методы FindAll и FindAllChildren также работают с массивами типа VBArray и возвращают именно такой массив. Для преобразования массива типа VBArray в массив типа JScript используется метод toArray.

В качестве примера работы с методом FindAll мы найдем все кнопки в Калькуляторе, заголовок которых начинается с символа C, а затем выведем в отчет заголовки всех кнопок, которые удовлетворяют этому условию.

```
function TestFindAll()
{
  var  wnd, i;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate()

  var PropArray = ConvertJScriptArray(new Array("WndCaption"));
  var ValuesArray = ConvertJScriptArray(new Array("C*"));

  var res = wnd.FindAll(PropArray, ValuesArray).toArray();

  for (i in res)
  {
    Log.Message(res[i].WndCaption);
  }
}
```

Результат работы функции:



Как видите, кроме кнопок C, CE и cos в результат вывелся и сам Калькулятор (первый результат Calculator Plus), так как его заголовок тоже начинается с символа C. Если заменить в нашем примере метод FindAll на FindAllChildren, то в результат выведутся только кнопки, так как само окно проверяться не будет.

Последний метод, используемый для поиска элементов управления, метод FindId,

используется для нахождения контрола по его уникальному идентификатору (id). Мы предоставим читателям самим опробовать работу с ним, так как он очень простой по сравнению с остальными методами Find...

И еще несколько слов по использованию методов Find... . Кроме параметров-массивов, в которых передаются имена и значения свойств искомых объектов, у этих методов есть и другие параметры:

- **Depth (глубина поиска).** Этот параметр может оказаться важным в том случае, если при создании проекта вы решили использовать модель объектов Tree. При использовании этой модели иерархия элементов управления в Object Browser-е может быть очень сложной и при поиске объектов необходимо указывать «глубину» поиска, т.е. поиск будет проводиться не только в объекте, для которого вызван метод, но и в его дочерних объектах. С помощью параметра Depth можно указать глубину поиска по иерархии. По умолчанию Depth=1. Чтобы искать во всех дочерних объектах, необходимо установить значение Depth очень большим, точно превышающим реальную глубину вложенности элементов (например, 10000).
- **Refresh (обновление).** По умолчанию TestComplete работает с кешированной версией иерархии объектов, которая иногда может отличаться от реальной иерархии (например, после определенных действий добавились или удалились некоторые элементы управления). Если установить параметр Refresh=true (как установлено по умолчанию), то в случае, если искомый объект не найден в кешированной версии приложения, TestComplete обновит ее и осуществит поиск еще раз.

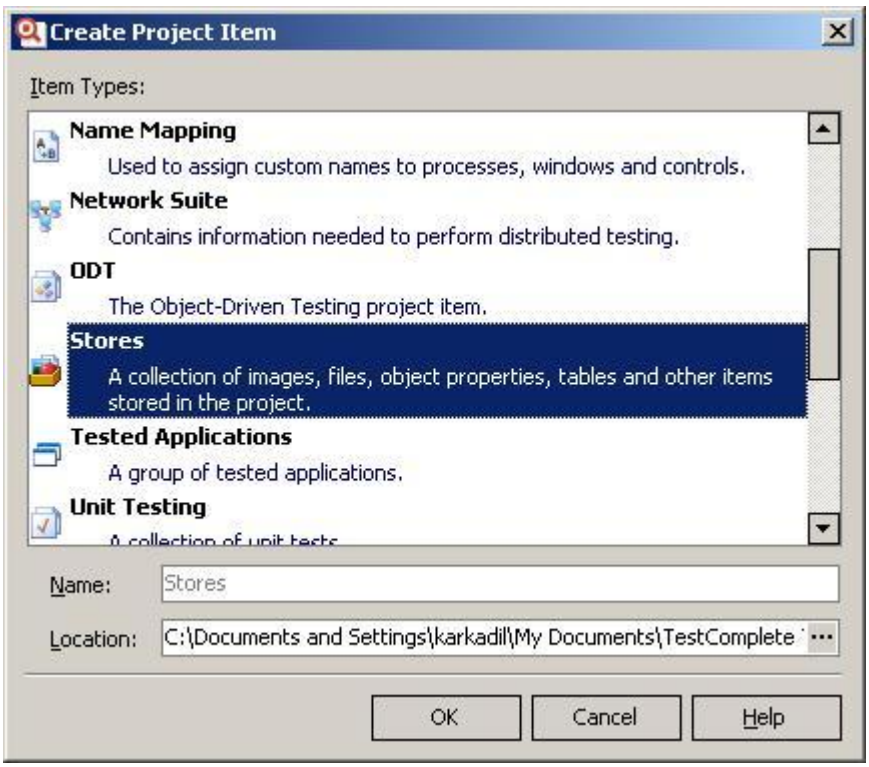
Многие приложения имеют очень сложную иерархию (гораздо сложнее, чем это кажется конечному пользователю) с большим количеством элементов управления (в том числе невидимых). В таких случаях поиск может производиться долго, поэтому всегда ищите компромисс между быстрой работой скриптов (когда методы Find не используются) и качеством их работы. Используйте методы Find только там, где это действительно необходимо и оправдано.

3.6 Использование хранилищ (Stores) и контрольных точек

Stores – это хранилище в TestComplete, куда вы можете складывать файлы, картинки, объекты и т.д. для дальнейшего их сравнения в скриптах. Например, в результате работы программы генерируется файл результатов, который необходимо проверить. Для простой проверки достаточно просто сравнить полученный файл с неким эталоном, который был один раз создан и проверен вручную. То же самое касается картинок, таблиц и прочих элементов Stores.

Checkpoint – это процесс сверки объекта, картинки и т.д. с неким эталонным объектом. В качестве эталонных объектов используются объекты, хранящиеся в Stores, а в качестве проверяемых объектов – элементы тестируемого приложения.

Прежде, чем начать работать со Stores, необходимо добавить соответствующий элемент в проект TestComplete. Для этого щелкнем правой кнопкой мыши на имени проекта, выберем пункт меню *Add – New Item* и в появившемся окне Create Project Item выберем Stores.



Теперь в нашем проекте появился пункт Stores с несколькими вложенными элементами:

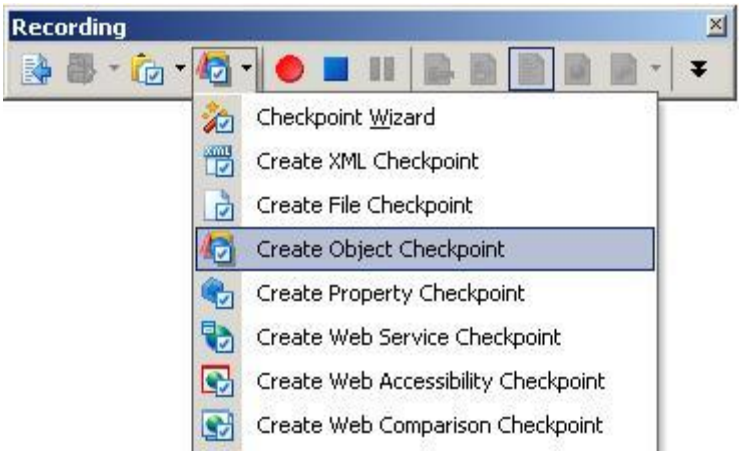
- **DBTables** – используется для хранения информации, хранящейся в базе данных
- **Files** – используется для хранения файлов
- **Objects** – используется для хранения информации об элементах управления (контролах)
- **Regions** – используется для хранения изображений (в частности скриншотов приложения и его элементов управления)
- **Tables** – здесь можно сохранять информацию из таблиц (гридов)
- **Web-Testing** – специальные элементы для проверки и сравнения веб-страниц
- **XML** – элементы для хранения информации в формате XML

Давайте рассмотрим несколько примеров использования хранилищ.

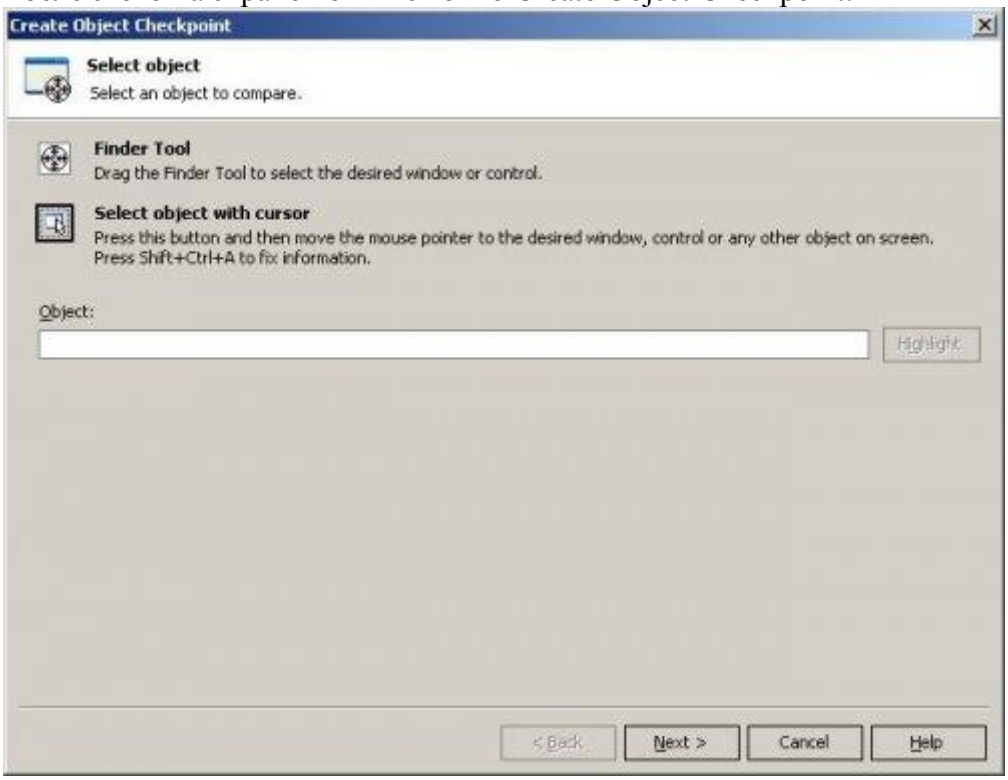
Objects

В раздел Objects помещается информация о различных объектах, которые используются в приложении (например, списки, диалоговые окна, кнопки и т.п.). Хранимая информация – это любые свойства, которые можно увидеть в Object Browser-е (например, WndCaption, Width, Class, Id и пр.).

Самый простой способ создания Object Checkpoint – это начать запись скрипта и затем на панели Recording выбрать пункт *Add checkpoint from list – Create object checkpoint*.

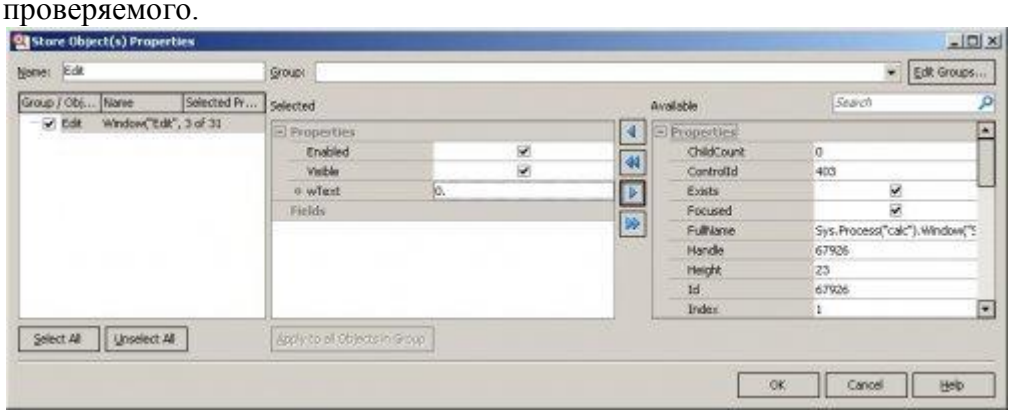


После этого на экране появится окно Create Object Checkpoint.



Теперь необходимо выбрать объект, для которого мы собираемся делать проверку. Для этого надо либо перетянуть картинку с изображением мишени (Finder Tool) на проверяемый объект, либо выбрать Select object with cursor, после чего установить указатель мыши над объектом и нажать *Ctrl-Shift-A*. При этом объект будет подсвечен красным контуром, а его полный путь появится в строке Object. При этом также может появиться окно Map object, где будет предложено сохранить объект в NameMapping. Если вы не используете NameMapping, просто нажмите Cancel.
В нашем примере мы будем делать проверку для единственного текстового поля в Калькуляторе.

Нажимаем Next, затем Edit и у нас появляется окно Store object(s) properties, где мы можем выбрать свойства, по которым будет выполняться проверка объекта. Если перед нажатием на Edit выбрать галочку Store properties of child objects, и в нашем контроле есть дочерние объекты, то нам будут предложены для проверки все объекты, которые находятся внутри проверяемого.



Как видно из этого примера, мы решили делать проверку по трем свойствам: Enabled (доступен), Visible (видимый) и wText (текст в поле). Жмем OK, Next, Finish, затем кнопку остановку записи на панели Recording.

Теперь в разделе Objects проекта появился новый элемент Edit, а в редактор кода вставлен код проверки:

```
function Test1()
{
  Objects.Compare(Sys.Process("calc").Window("SciCalc",
"Calculator Plus").Window("Edit"), "Edit", true);
}
```

Теперь можно запустить полученный скрипт и в логе мы получим сообщение «The objects are the same.». Для проверки того, что код работает корректно, можно ввести в Калькуляторе какое-то число, запустить функцию снова и тогда мы в результате получим в логе предупреждение «Objects.Compare error», а в поле Remarks под логом полное объяснение того, какая именно проверка не прошла и почему:

```
The "wText" property of the "Window("Edit", "", 1)" object holds
a value that differs from the "wText" property of the stored
"Edit" object.
Current value = "12. ".
Stored value = "0. ".
```

Конечно, совсем необязательно каждый раз для создания чекпоинта запускать запись скрипта, есть и более простой способ. Для этого необходимо щелкнуть правой кнопкой мыши в дереве проекта на элементе Objects и выбрать пункт меню Add – New Item, после чего пройти по шагам процесс добавления объекта (он аналогичен процессу добавления объекта во время записи), а затем вручную вставить код проверки в код скрипта.

Процесс создания других типов чекпоинтов аналогичен чекпоинту для объекта, поэтому мы не будем их рассматривать подробно. Разница лишь в том, что для других типов проверок во время добавления элементов в хранилище надо выбирать и настраивать другие параметры, специфичные для этого типа. Например, при создании чекпоинта для региона мы можем выбрать точность проверки (Tolerance), следует ли учитывать положение курсора при проверке и т.п.

Единственное, о чем хотелось бы еще упомянуть в этой главе: используйте проверки скриншотов (Regions) только в самом крайнем случае, когда другие способы не подходят. Проверка регионов является, с одной стороны, наиболее точной, но с другой стороны, малейшие изменения в интерфейсе влияют на результаты проверок. И если уж вы приняли решение об использовании проверки регионов, то проверяйте отдельные элементы управления, а не окна целиком, иначе в дальнейшем вам придется очень часто обновлять картинки, хранящиеся в Regions.

Работа с изображениями более подробно рассматривается в главе 12 Особенности работы с графическими объектами

3.7 Запуск скриптов

Теперь пришло время подробнее поговорить о запуске тестовых скриптов. Из предыдущей главы вы знаете один способ запуска: щелкнуть правой кнопкой мыши на функции и выбрать пункт меню Run Current Routine.

Однако что делать, если нам необходимо запустить несколько функций подряд? самым простым способом будет создать одну функцию, из которой вызывать все необходимые функции. Например, если у нас есть функции Test1, Test2 и Test3, то мы можем создать новую функцию:

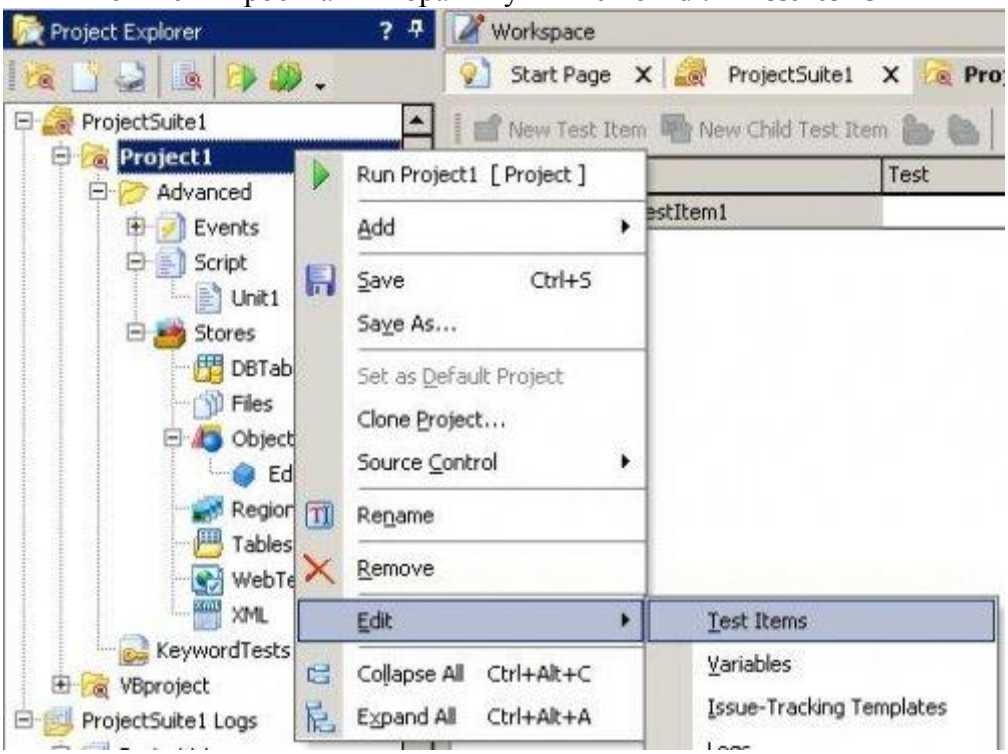
```
function Test()
{
  Test1();
  Test2();
  Test3();
}
```

Если теперь запустить эту функцию, то запустятся по очереди функции Test1, Test2, Test3.

В TestComplete предусмотрена специальная возможность для запуска нескольких функций подряд, которая называется Test Items. Test Items – это список функций в проекте или наборе проектов. У каждого Test Item-а есть имя и он связан с какой-то функцией в проекте.

Чтобы открыть список Test Item-ов для проекта, необходимо щелкнуть правой кнопкой

мыши по имени проекта и выбрать пункт меню Edit – Test Items



При этом в правой части окна TestComplete откроется список Test Items (в самом начале он, естественно, пустой).

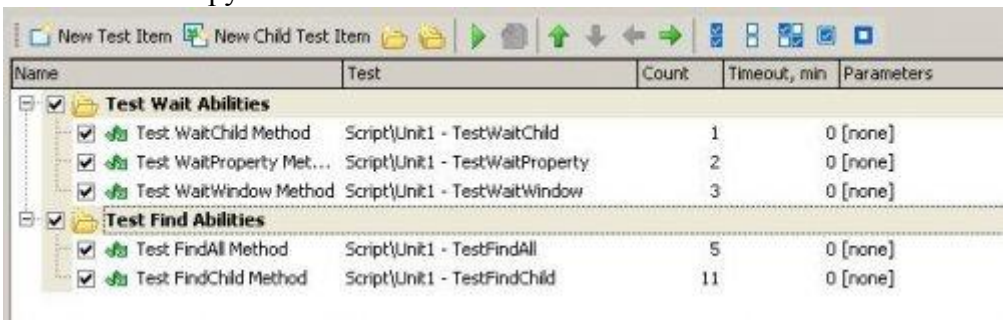
Чтобы добавить Test Item, необходимо нажать на кнопку New Test Item, которая находится в верхней части панели Test Items. При этом в список добавится новая строка:

В колонке Name указывается имя Test Item-а (оно необязательно должно совпадать с именем функции, которая будет связана с этим Test Item-Ом). В поле Test указывается имя запускаемой функции; для этого необходимо нажать на кнопку с троеточием в правой части ячейки и в открывшемся окне Select test выбрать нужную функцию. В поле Count указывается количество раз, сколько будет запущен этот тест. В поле Timeout указывается максимальное время в минутах, сколько может работать тест. Если время прошло, а тест не закончил работу, считается, что он завис, его выполнение прекращается и начинается выполнение следующего теста. В колонку Parameters вносятся параметры, передаваемые функции (если функция, конечно, принимает какие-то параметры). И в поле Description вносится описание теста или комментарий к нему.
Чекбокс слева в каждой строке указывает, будет ли запущен этот тест при пакетном запуске (т.е. когда запускаются сразу все Test Item-ы).

В этом примере мы добавили два Test Item-а:

1. "Test WaitChild Method" связан с функцией TestWaitChild из модуля Unit1, будет выполнен 1 раз без таймаута (что означает, что чисто теоретически он может выполняться вечно)
2. "Test WaitProperty Method" связан с функцией TestWaitProperty из модуля Unit1, будет выполнен 2 раза с таймаутом 5 минут (что означает, что если его выполнение не закончится через 5 минут, оно будет прервано принудительно)

Test Item-ы можно также объединять в группы для большей наглядности. Для этого используются кнопки New Child Test Item, Add New Group и Add New Subgroup на панели инструментов окна Test Items. На скриншоте ниже показан пример подобной организации Test Item-ов в группы.



После того, как Test Item-ы добавлены, их можно передвигать вверх-вниз (тем самым меняя очередность их выполнения), а также перемещать по уровням влево и вправо (меняя их принадлежность к группам).

Естественно, одна функция может быть добавлена в список Test Items сколько угодно раз.

Для того чтобы запустить все Test Item-ы, необходимо выбрать пункт меню *Test – Run Project* или нажать комбинацию клавиш Ctrl-F5.

Если в вашем наборе проектов есть несколько проектов, то можно также использовать Test Item-Ы на уровне набора проектов. Для этого надо щелкнуть правой кнопкой мыши на имени Project Suite-а и выбрать пункт меню *Edit – Test Items*. Каждый Test Item на уровне Project Suite-а – это набор Test Item-ов в каждом проекте, т.е. при запуске test Items набора проектов фактически будут запущены все Test Item-ы из всех проектов. Для запуска всего Project Suite-а необходимо выбрать пункт меню *Test – Run Project Suite* или нажать *Ctrl-Alt-F5*.

Обратите внимание! В более ранних версиях TestComplete (до версии 7.0) существовало понятие Main Routine. Main Routine – это некая "главная" функция в проекте, которая запускалась по нажатию клавиши F5. Для задания main Routine необходимо было дважды щелкнуть на элементе Script в дереве проекта. Начиная с TestComplete 7.0 и выше эта функциональность убрана, и нажатие клавиши F5 запускает Test Items проекта (как и Ctrl-F5).

Если в вашем проекте (или проектах) написано очень много скриптов и они выполняются долго, TestComplete может постепенно занимать все больше и больше памяти, замедляя работу системы вообще и тестируемого приложения в частности. В этом случае вам, возможно, придется отказаться от использования Test Items и запускать скрипты из командной строки по частям, каждый раз стартуя TestComplete заново (для каждого скрипта или набора скриптов). Более подробно о запуске из командной строки можно прочитать в главе 11.2 Запуск TestComplete из командной строки

3.8 Использование логов и анализ результатов

Вы уже имеете небольшой опыт работы с логами, которые генерирует TestComplete во время работы тестовых скриптов. Теперь мы рассмотрим работу с логом подробнее.

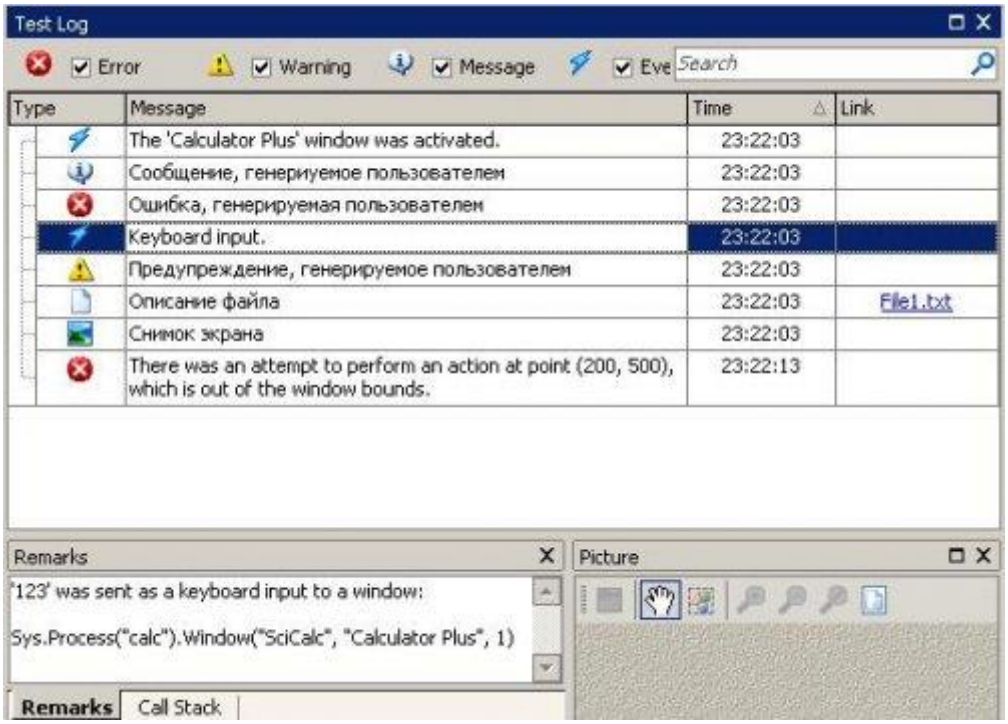
Если открыть любой лог, то в его верхней части вы увидите 6 типов сообщений, которые могут быть использованы:

- **Error** – ошибки. Это могут быть как ошибки, автоматически генерируемые TestComplete-ом (например, когда TestComplete не находит объект или скрипт пытается выполнить щелчок мышью за пределами экрана), так и генерируемыми вручную (например, если выполняется какая-то проверка и необходимо сообщить о ее неправильных результатах).
- **Warning** – предупреждения. Это тоже ошибки, но не критичные. Они также могут генерироваться TestComplete-ом или писаться непосредственно в скриптах.
- **Message** – сообщения. Любые сообщения, которые могут быть полезны при просмотре логов. Сообщения вставляются в текст скрипта вручную и не генерируются TestComplete-ом.
- **Events** – события. Происходят каждый раз, когда скрипт вводит текст (с помощью метода Keys) или щелкает мышью. События генерируются TestComplete-ом автоматически, но сообщения такого типа также можно создавать в скрипте.
- **Image** – изображения. Позволяют вставлять изображения в лог (например, снимки экрана).
- **File** – файл. Позволяет вставить в лог файл с возможностью открыть его непосредственно из лога. Это может быть полезно, например, если тестируемое приложение генерирует файл с данными и при просмотре лога есть необходимость в открытии этого сгенерированного файла.
- **Link** – ссылка на файл или другой ресурс

Прежде чем изучать другие возможности системы логгирования TestComplete, рассмотрим простой пример работы с логом. Для генерации событий мы воспользуемся все тем же приложением Калькулятор.

```
function UseLog()
{
  var  wnd, i;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate();
  Log.Message("Сообщение, генериуемое пользователем", "Дополнительный текст
сообщения");
  Log.Error("Ошибка, генерируемая пользователем", "Дополнительный текст
ошибки");

  wnd.Keys("123");
  Log.Warning("Предупреждение, генерируемое пользователем", "Дополнительный
текст предупреждения");
  Log.File(Project.Path + "\\Stores\\Files\\testfile.txt", "Описание файла",
"Дополнительный текст");
  Log.Picture(Sys.Desktop.Picture(), "Снимок экрана", "Дополнительный
текст");
  wnd.Click(200, 500);

}
```

Если запустить этот скрипт, мы получим следующий результат:



Здесь мы можем видеть как сначала генерируется событие при активации окна, затем мы выводим в лог сообщение и ошибку, затем имитируем нажатие клавиш (что также приводит к генерации события и соответствующей записи в логе), затем мы вставляем предупреждение, файл, снимок экрана и в конце делаем попытку щелкнуть мышью за пределами окна, что приводит к генерации TestComplete-ом ошибки.

В нижней части окна находятся секции Remarks (дополнительный текст, комментарии) и Picture (здесь отображается картинка, когда в логе выделено сообщение типа Image).

Теперь рассмотрим более сложные примеры работы с логом.

У каждой записи в логе (ошибки, сообщения, предупреждения и т.п.) есть атрибуты (цвет текста, цвет фона, шрифт), которые можно изменять. Вот пример использования атрибутов.

```
function UseLogAttr()
{
  var attr = Log.CreateNewAttributes();
    attr.BackColor = BuiltIn.clBlue;
    attr.FontColor = BuiltIn.clWhite;
    attr.Bold = true;
    attr.Italic = true;
    attr.Underline = true;

  Log.Message("Message without attributes");
  Log.Message("Message with attributes", "", pmNormal, attr);
    attr.BackColor = BuiltIn.clRed;
    attr.Bold = false;
    attr.Italic = false;
    attr.Underline = false;
  Log.Message("Message with attributes", "", pmHighest, attr);
  Log.Message("Message without attributes", "", pmHighest);
}
```

Результат работы функции:

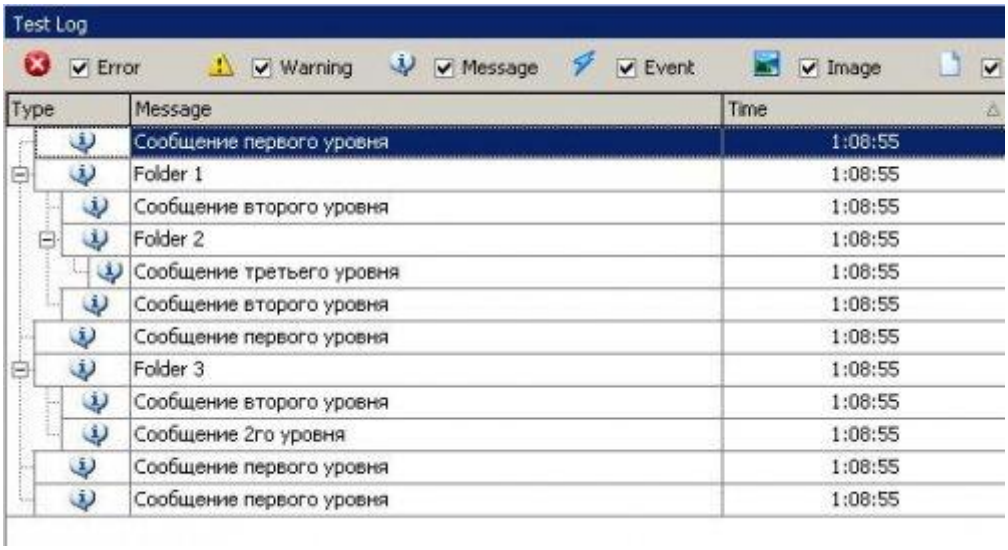
Также в логе можно использовать папки (folders). Папка – это элемент лога, который содержит внутри себя дочерние элементы, в результате чего лог имеет древовидную структуру. Эта возможность может быть полезна, например, для того, чтобы все события, сообщения, ошибки и пр. для конкретного шага тесткейса не выводились в общий лог, а были скрыты до тех пор, пока не понадобятся для анализа. Это позволяет создавать более удобочитаемые логи.

Для работы с папками используются методы CreateFolder, AppendFolder, PushLogFolder, PopLogFolder. Метод AppendFolder создает новую папку и делает ее активной, в результате чего все следующие сообщения будут помещаться в нее до тех пор, пока она не будет закрыта с помощью метода PopLogFolder.

Метод CreateFolder создает папку, но не делает ее активной. При этом метод CreateFolder возвращает идентификатор созданной папки. С помощью этого идентификатора позже можно активировать эту папку методом PushLogFolder. Кроме того, во всех методах объекта Log (Message, Error, Warning и т.д.) есть необязательный параметр FolderID, с помощью которого можно помещать сообщения или ошибки прямо в необходимую папку.

Ниже показан пример работы с папками и результат работы скрипта.

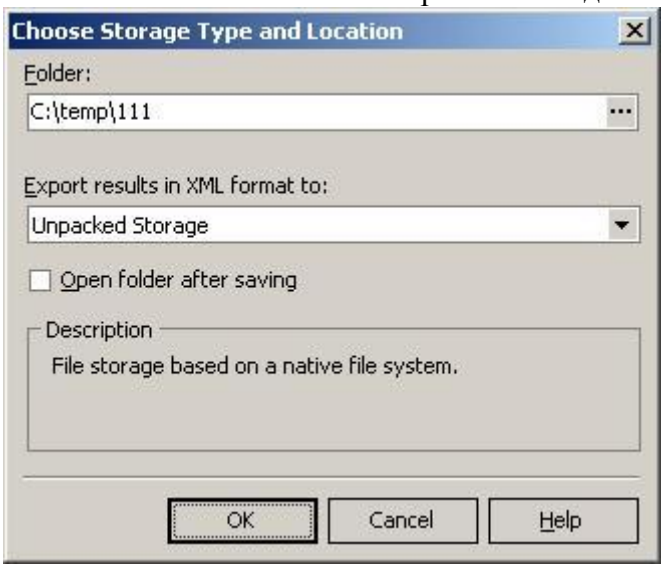
```
function UseLogFolder()
{
  Log.Message("Сообщение первого уровня");
  Log.AppendFolder("Folder 1");
  Log.Message("Сообщение второго уровня");
  Log.AppendFolder("Folder 2");
  Log.Message("Сообщение третьего уровня");
  Log.PopLogFolder();
  Log.Message("Сообщение второго уровня");
  Log.PopLogFolder();
  Log.Message("Сообщение первого уровня");

  var folderID = Log.CreateFolder("Folder 3");
  Log.Message("Сообщение первого уровня");
  Log.PushLogFolder(folderID);
  Log.Message("Сообщение второго уровня");
  Log.PopLogFolder();
  Log.Message("Сообщение первого уровня");

  Log.Message("Сообщение 2го уровня", "", pmNormal, null, null, folderID);
}
```



При просмотре лога в верхней части панели можно отключать с помощью соответствующих чекбоксов те типы сообщений, которые вам в данный момент не нужны (например, можно отключить вывод на экран событий, а можно и вовсе отображать только ошибки). Если вы ищете какое-то конкретное сообщение и вам недостаточно этих фильтров для поиска нужного сообщения – воспользуйтесь строкой поиска в верхнем правом углу панели лога.

Полученные в результате работы скриптов логии можно экспортировать в HTML либо MHT формат. Для этого на панели инструментов лога необходимо нажать кнопку Export Test Results

затем в появившемся окне выбрать необходимые параметры и нажать OK.



Результаты работы скриптов также можно экспортировать непосредственно из самих скриптов (например, в самом конце их работы). Для этого нужно воспользоваться методом Log.SaveResultsAs.

Кроме рассмотренных возможностей, в TestComplete-е также есть и другие полезные свойства и методы, полезные для работы с логом.

- **ErrCount, WrnCount, EvnCount, FileCount, ImgCount, LinkCount** – количество ошибок, предупреждений и т.д. в логе
- **FolderErrCount, FolderWrnCount, FolderEvnCount, FolderFileCount, FolderImgCount, FolderLinkCount** – количество ошибок, предупреждений и т.д. в конкретной папке лога
- **LockEvents/UnlockEvents** – блокировка/разблокировка событий. После использования метода LockEvents TestComplete не будет помещать в лог события (Events) до тех пор, пока не отработает метод UnlockEvents. Это может быть полезно в том случае, когда при тестировании генерируется очень много событий, что влечет за собой большой размер лога и долгое время его открытия

3.9 Отладка скриптов

Отладка – это такой режим запуска скриптов, когда на любой строке кода можно поставить точку прерывания (breakpoint) и работа скрипта будет остановлена в этом месте до тех пор, пока пользователь не решит, что делать дальше. Режим отладки используется в тех случаях, когда скрипты по какой-то причине работают не так, как хотелось бы, и необходимо проверить значения переменных на момент выполнения скрипта.

В TestComplete есть все необходимые инструменты для отладки скриптов. Если вы используете язык DelphiScript, то вам не нужно ничего дополнительно, однако для доступа к отладочным функциям при использовании других языков необходимо установить MS Script Debugger.

Кроме того, убедитесь, что опция *Debug – Enable Debugging* включена. Еще желательно включить опции *Tools – Options – Engines – General – Debug – Highlight execution point* и *Tools – Options – Engines – Log – Show Log on Pause.*

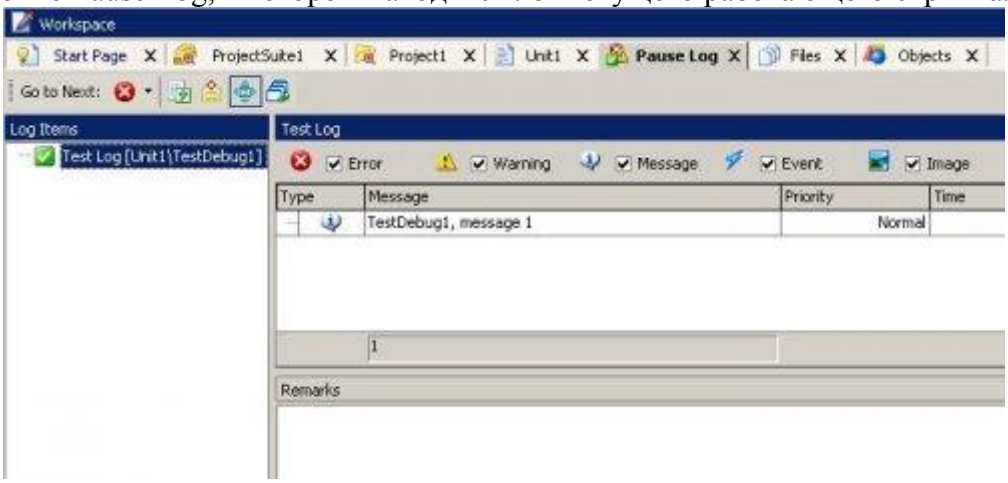
Для изучения возможностей режима отладки мы создадим 2 функции TestDebug1 и TestDebug2, каждая из которых будет помещать в лог сообщения. Кроме того, функция TestDebug1 будет вызывать функцию TestDebug2.

```
function TestDebug1()
{
  Log.Message("TestDebug1, message 1");
  TestDebug2();
  Log.Message("TestDebug1, message 1");
}

function TestDebug2()
{
  Log.Message("TestDebug2, message 1");
  Log.Message("TestDebug2, message 2");
}
```

Теперь поставим точку прерывания в функции TestDebug1 на строке вызова функции TestDebug2. Для этого необходимо щелкнуть мышью на этой сроке и нажать клавишу F9. После этого строка станет красной.

```
172  function TestDebug1()
173  {
174    Log.Message("TestDebug1, message 1");
       TestDebug2();
176    Log.Message("TestDebug1, message 1");
177  }
178
179  function TestDebug2()
180  {
181    Log.Message("TestDebug2, message 1");
182    Log.Message("TestDebug2, message 2");
183  }
```

Если теперь запустить функцию TestDebug1, то ее выполнение будет приостановлено непосредственно перед вызовои функции TestDebug2, а в панели закладок появится новое окно Pause Log, в котором находится лог текущего работающего скрипта.



Как видно из приведенного скриншота, первое сообщение уже отправлено в лог и TestComplete ждет наших дальнейших действий. Вернемся на вкладку модуля скрипта.

Здесь мы можем сделать следующие действия:

- продолжить выполнение скрипта, нажав F5. При этом выполнение будет продолжено до следующего брекпоинта или до конца, если других брекпоинтов нет
- продолжить выполнение построчно с помощью клавиши F10. При этом будет выполнена следующая строка текущей функции (в нашем случае полностью отработает вызов функции TestDebug2)
- продолжить выполнение построчно функции, вызываемой в данной строке, с помощью клавиши F11. При этом мы переместимся на первую строку вызываемой функции TestDebug2, но ее выполнение не начнется, его тоже надо будет проходить по шагам
- продолжить выполнение скрипта до того места, где находится текстовый курсор (F4)

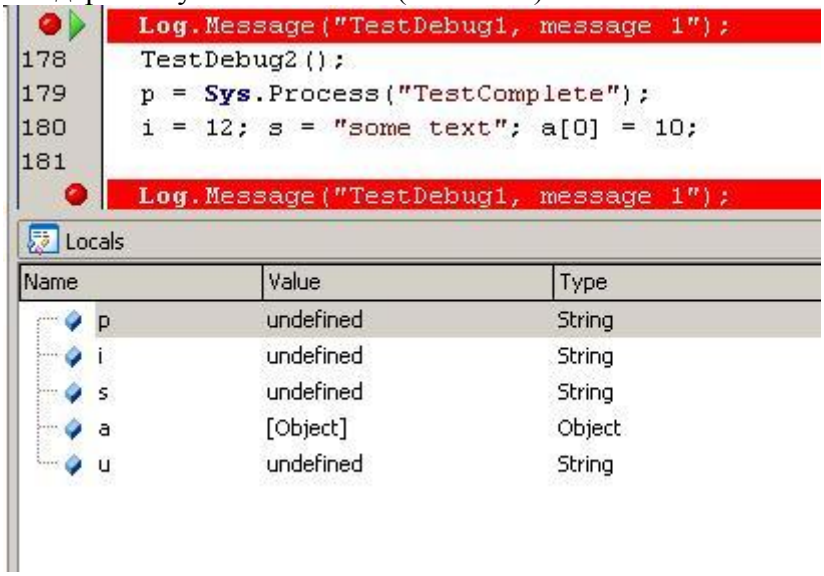
Теперь добавим в наши функции объявления переменных и ознакомимся с другими возможностями отладочного режима.

```
function TestDebug1()
{
  var p, i, s, a = new Array(), u;

  Log.Message("TestDebug1, message 1");
```

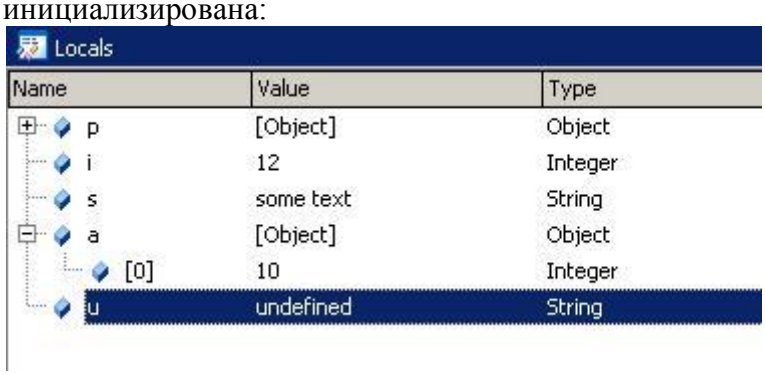
```
  TestDebug2();
  p = Sys.Process("TestComplete");
  i = 12; s = "some text"; a[0] = 10;

  Log.Message("TestDebug1, message 1");
}
```

Поставьте брекпоинты на строках Log.Message и запустите функцию. После остановки на первом брекпоинте обратите внимание на нижнюю часть окна TestComplete, где находятся вкладки Call Stack, Locals, Watch List и Breakpoints.

На вкладке Locals отображаются все локальные переменные (т.е. переменные, доступные из текущей функции), их значения и тип. В данный момент все они не инициализированы и содержат пустые значения (undefined).



Теперь нажмите F5 и после того, как выполнение скрипта остановится на следующем брекпоинте, снова обратите внимание на вкладку Locals. Теперь мы видим, какое значение в данный момент имеют все переменные кроме переменной u, которая так и не была инициализирована:

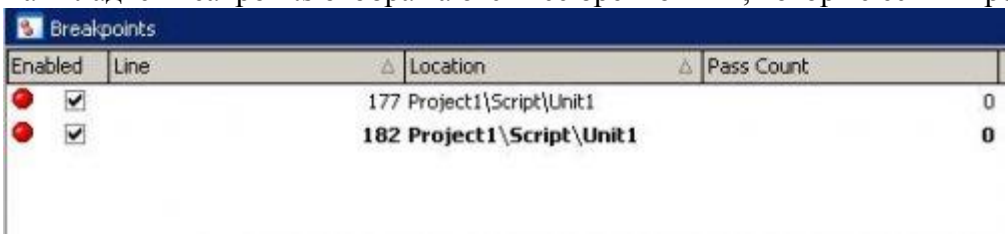


Если щелкнуть по значку + возле переменной p – раскроется список доступных свойств этого объекта (в данный момент это свойства процесса "TestComplete"). Если щелкнуть на нем правой кнопкой мыши и выбрать пункт меню Inspect, откроется окно, аналогичное Object Browser-у, в котором можно посмотреть все свойства и методы объекта.

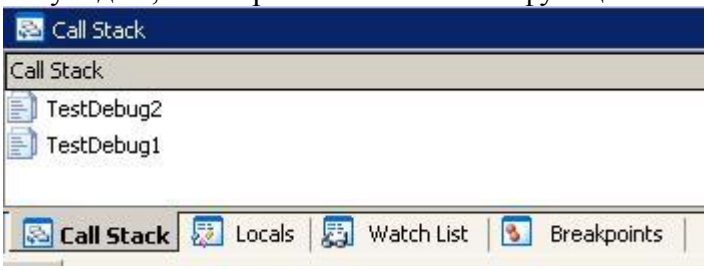
Теперь перейдите на вкладку Watch List. Здесь вы можете как смотреть значения любых переменных (локальных и глобальных), так и вычислять целые выражения. Ниже показан

пример проверки существования процесса и нескольких переменных.

На вкладке Breakpoints отображаются все брекпоинты, которые есть в проекте.



На вкладке Call Stack отображается порядок вызова вложенных функций. Например, в нашем случае, если остановить выполнение скрипта в середине функции TestDebug2, то мы увидим, что первой была вызвана функция TestDebug1, а затем из нее TestDebug2.



3.10 Работа с несколькими модулями (Units)

До сих пор мы все действия делали в одном модуле (Unit), однако обычно схожие по назначению функции выделяются в отдельные модули и уже из этих модулей вызываются функции.

Для того, чтобы добавить новый модуль, необходимо щелкнуть правой кнопкой мыши на элементе проекта Scripts и выбрать пункт меню *Add – New Item*, а затем в появившемся окне ввести имя нового модуля.

Предположим, что у нас есть два модуля: **Unit1** и **Unit2**, и мы хотим из модуля Unit2 вызвать функцию, которая находится в модуле Unit1. Это можно сделать двумя способами:

1. С помощью метода Runner.CallMethod
2. Подключив модуль Unit1 в модуле Unit2

Пример использования первого метода:

```
function MyFunction1()
{
  Runner.CallMethod("Unit1.Sleep", 10);
}
```

Обратите внимание, что имя вызываемой функции содержит имя модуля и имя функции, разделенные точкой. Дальше через запятую задаются параметры, которые передаются в вызываемую функцию.

Если функции из какого-то модуля используются часто, есть смысл один раз подключить этот модуль и затем вызывать функции из него напрямую. Для подключения модуля используется ключевое слово USEUNIT, перед которым ставится символ комментария без пробела (или для DelphiScript ключевое слово uses). Вот как будет выглядеть подключение модуля для разных языков:

JScript, C++Script, C#Script
```
//USEUNIT Unit1
```

VBScript
```
'USEUNIT Unit1
```

DelphiScript
```
uses Unit1
```

Пример использования:
```
//USEUNIT Unit1

function MyFunction2()
{
  TestIndicator();
}
```

Если же у вас подключены 2 модуля и в них есть функции с одинаковыми именами, можно воспользоваться полным именем для доступа к функции из конкретного модуля:

```
//USEUNIT Unit1

function MyFunction1()
{
  Unit1.TestIndicator();
}
```

Кроме того, можно подключить модуль визуально, щелкнув правой кнопкой мыши на имени модуля и выбрав пункт меню *Add Unit References*. Тогда директивы подключения будут вставлены в модули автоматически.

Обратите внимание на некоторые особенности использования модулей:

1. Директива подключения (uses или USEUNIT) должна быть в самом начале модуля
2. Для подключения нескольких модулей в DelphiScript просто перечислите их имена через запятую. Для остальных языков программирования подключайте каждый модуль отдельной строкой USEUNIT

3. Перед символом комментария, а также между символом комментария и ключевым словом USEUNIT не должно быть пробелов
4. Взаимное подключение модулей (т.е. Unit1 подключается в Unit2, а Unit2 подключается в Unit1) невозможно при использовании языков JScript, C++Script и C#Script
5. В случае подключения нескольких модулей друг из друга (например, Unit1 подключается в Unit2, а Unit2, в свою очередь, подключается в Unit3), нельзя использовать более трёх подключений, иначе на самом нижнем уровне будут недоступны функции из самого верхнего уровня (т.е. в нашем примере если создать модуль Unit4, то из него будет невозможно вызвать функции из модуля Unit1)

3.11 Использование фреймворков и организация кода

Самым простым способ создания тестовых скриптов является их запись с помощью встроенных средств TestComplete. Однако записанные скрипты являются самыми неэффективными, так как их приходится потом очень часто изменять, а иногда и полностью переписывать (если изменения в тестируемом приложении существенны).

Для того чтобы снизить количество подобных изменений или вообще избавиться от них, при написании автоматических скриптов обычно применяются так называемые фреймворки (Frameworks).

Фреймворк – это такая организация проекта, которая позволяет упростить разработку, поддержку и модификацию программного кода. На данный момент в автоматизации тестирования существует несколько широко используемых фреймворков:

- **Functional Decomposition (Функциональная декомпозиция)** – разнесение кода в разные функции или модули в зависимости от их назначения. Например, функции для работы с файловой системой могут храниться в одном файле, а функции для работы с окнами – в другом. С функциональной декомпозицией тесно связано понятие Рефакторинга (внесение изменений в существующий код, при котором функциональность остается без изменений, а сам код изменяется для более удобного восприятия и модификации)
- **Data-driven Testing (тесты, управляемые данными)** – при котором тестовые данные выносятся в отдельные файлы (например, в файл Excel или в базу данных), а тестовые скрипты считывают их оттуда по мере надобности. Этот подход обычно используют в сочетании с другими подходами. Более подробно о нем можно почитать в главе 7 Data Driven Testing
- **Keyword-driven Testing (тесты, управляемые ключевыми словами)** – при этом подходе тесты выглядят как некие ключевые слова, каждое из которых отвечает за выполнение какого-то блока кода. В TestComplete есть специальная надстройка, позволяющая записывать подобные тесткейсы, более подробно о них можно почитать в главе 6 Keyword Driven Testing. Другой пример подобного подхода к тестированию – это создание специального кода-обертки, который принимает какие-то ключевые слова и в зависимости от того, какие ключи были переданы, выполнится тот или иной программный код
- **Object-driven Testing (тесты, управляемые объектами)** – при этом подходе функциональность приложения представляется в скриптах в виде объекта с собственными свойствами и методами. Более подробно об этом подходе в TestComplete можно прочитать в главе 9 Object Driven Testing.

Если у вас нет опыта в проведении рефакторинга и/или создании проекта с нуля, вот вам несколько простых советов, которые могут сэкономить вам время в будущем:

1. **Не записывайте скрипты.** Средства записи полезны только на начальном этапе изучения средства автоматизации. Наиболее эффективные и удобные для поддержки скрипты пишутся вручную
2. **Не дублируйте код.** Если вы скопировали какой-то код больше одного раза – это повод выделить его в отдельную функцию или метод
3. **Не усложняйте написание скриптов.** Иногда встречаются задачи, требующие на реализацию много времени и/или изучения новых технологий. Если это разовая задача, то не стоит заниматься "программированием ради программирования" и тратить на решение отдельной задачи кучу времени. Обычно для таких задач можно найти более простой и быстрый обходной путь, который, возможно, покажется вам менее красивым и изящным, но если он будет хорошо работать, то можно оставить и так. Конечно, если подобная задача встречается часто, то лучше один раз потратить много времени и решить задачу хорошо
4. **Ни одно приложение невозможно автоматизировать на 100%,** а некоторые задачи настолько сложны для автоматизации, что лучше оставить их для ручного тестирования
5. **Старайтесь не писать очень длинные функции и функции с огромным количеством параметров,** так как это усложнит понимание написанного кода другим сотрудникам (даже если вы работаете один, всегда есть вероятность того, что в будущем с вами или вместо вас будут работать другие люди)
6. **Возьмите готовые или напишите свои стандарты кодирования и придерживайтесь их.** Это упростит чтение и модификацию скриптов в дальнейшем как вам, так и другим участникам проекта

4 Работа с Web-приложениями

TestComplete обладает настолько богатыми возможностями для тестирования веб-приложений, что этому стоит посвятить отдельную главу.

Прежде всего стоит сказать, что для TestComplete неважно, на чем создано приложение (будь то PHP или ASP .NET) – в браузере странички будут отображаться одинаково и работа с ними в TestComplete не будет различаться.

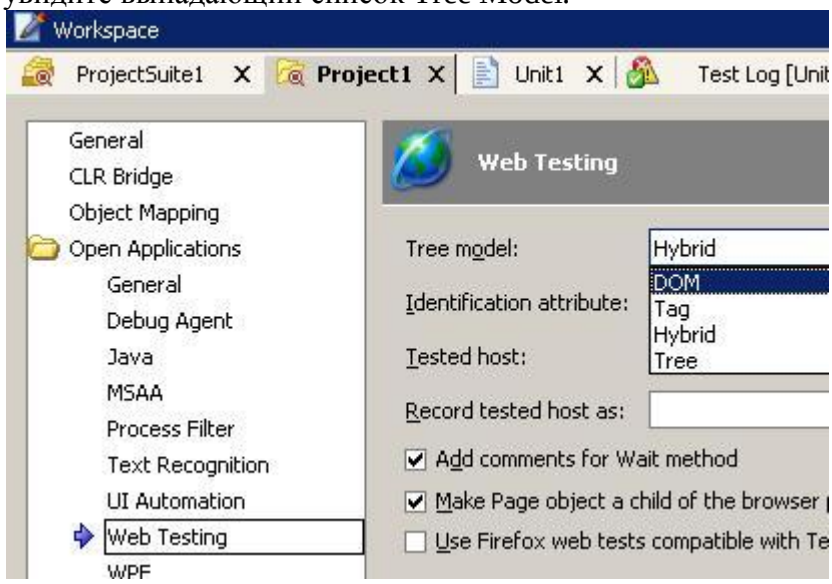
В TestComplete есть средства для функционального тестирования, нагрузочного тестирования веб-приложений, тестирования веб-сервисов, а также для тестирования Flex, Flash и Silverligth приложений. Каждому виду тестирования посвящен отдельный раздел в этой главе.

4.1 Функциональное тестирование Web-приложений

Выбор модели объектов

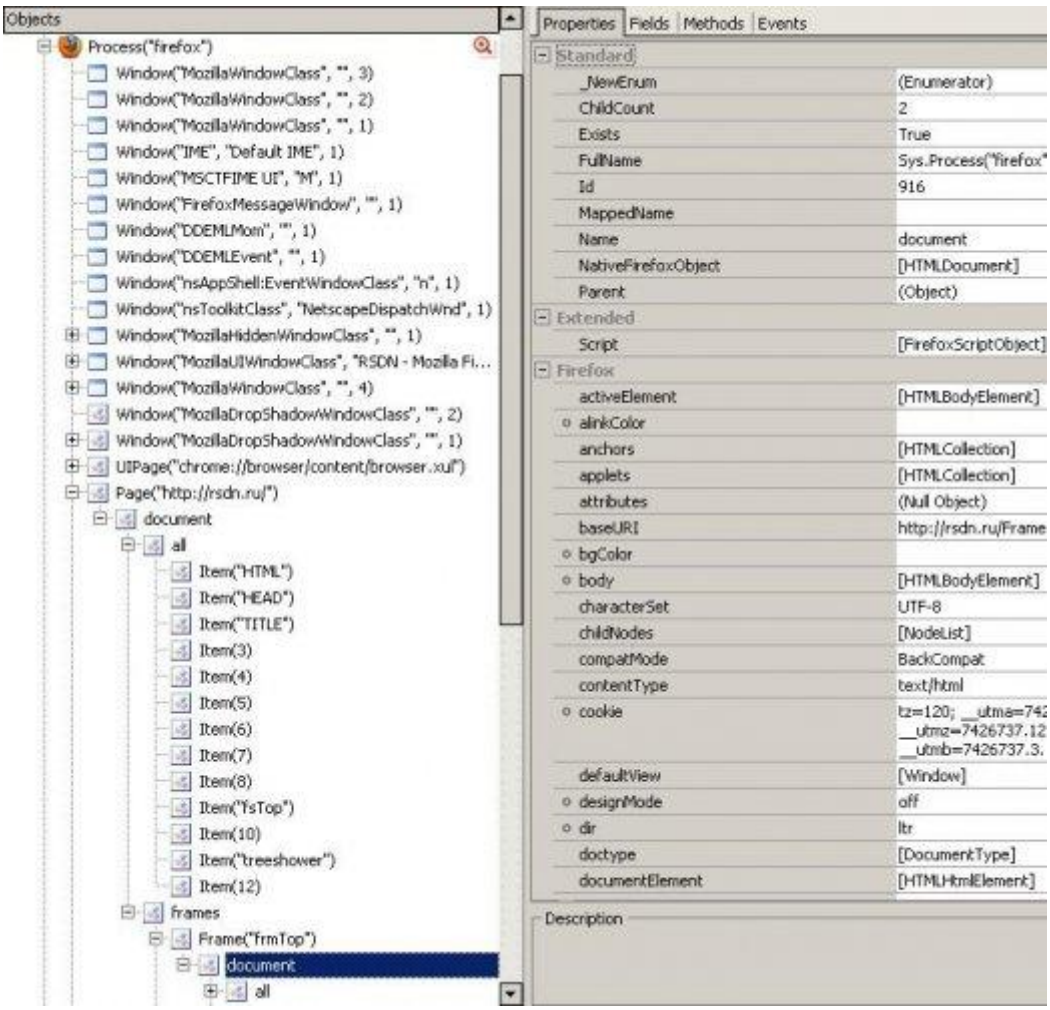
Прежде чем начать разработку скриптов для функционального тестирования веб-приложений, необходимо выбрать модель объектов веб-приложений. В главе 3.1 Выбор модели объектов мы рассмотрели 2 модели приложений: Flat и Tree. В случае тестирования веб-приложений эти опции также важны, однако кроме них необходимо установить и специальную модель для веб-приложений.

Для этого надо щелкнуть правой кнопкой мыши по имени проекта и выбрать пункт меню Edit – Properties, затем open Applications – Web Testing. В правой панели TestComplete вы увидите выпадающий список Tree Model:



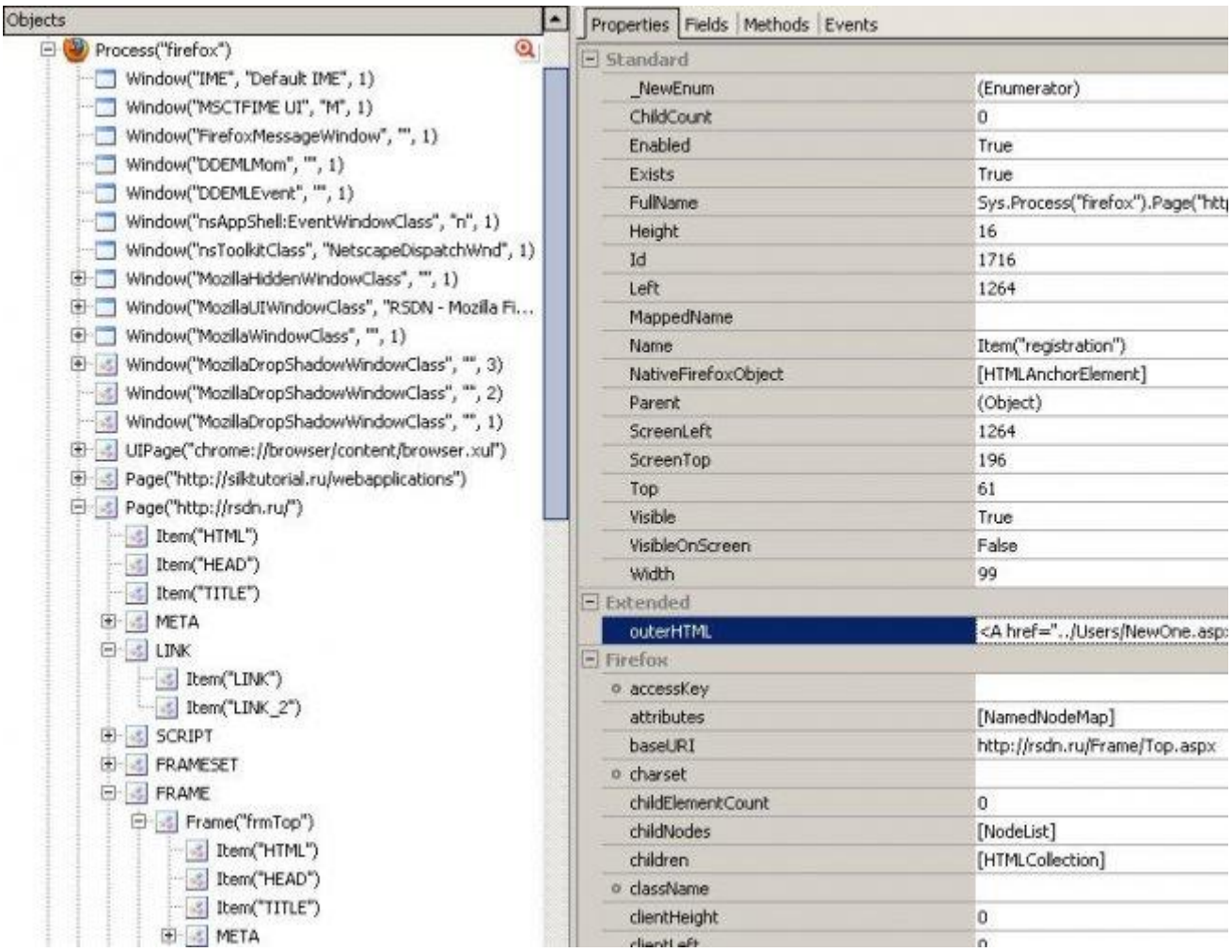
У нас есть выбор из 4х моделей: DOM, Tag, Tree и Hybrid. Мы рассмотрим примеры выбора разных моделей на примере браузера Firefox и сайта http://rsdn.ru/ .

DOM-модель. При выборе DOM-модели объект Page содержит свойства document и frames, а элементы веб-страницы доступны через свойство document.all.



Каждый элемент имеет имя вида Item(Index), где Index – это id, имя или порядковый номер элемента управления.
В некоторых случаях (например, в случае использования элементов управления типа radio buttons) каждый элемент такого элемента управления отображается отдельно, что может существенно замедлить выполнение скриптов. В таких случаях рекомендуется использовать другие модели объектов (Tag или Tree).

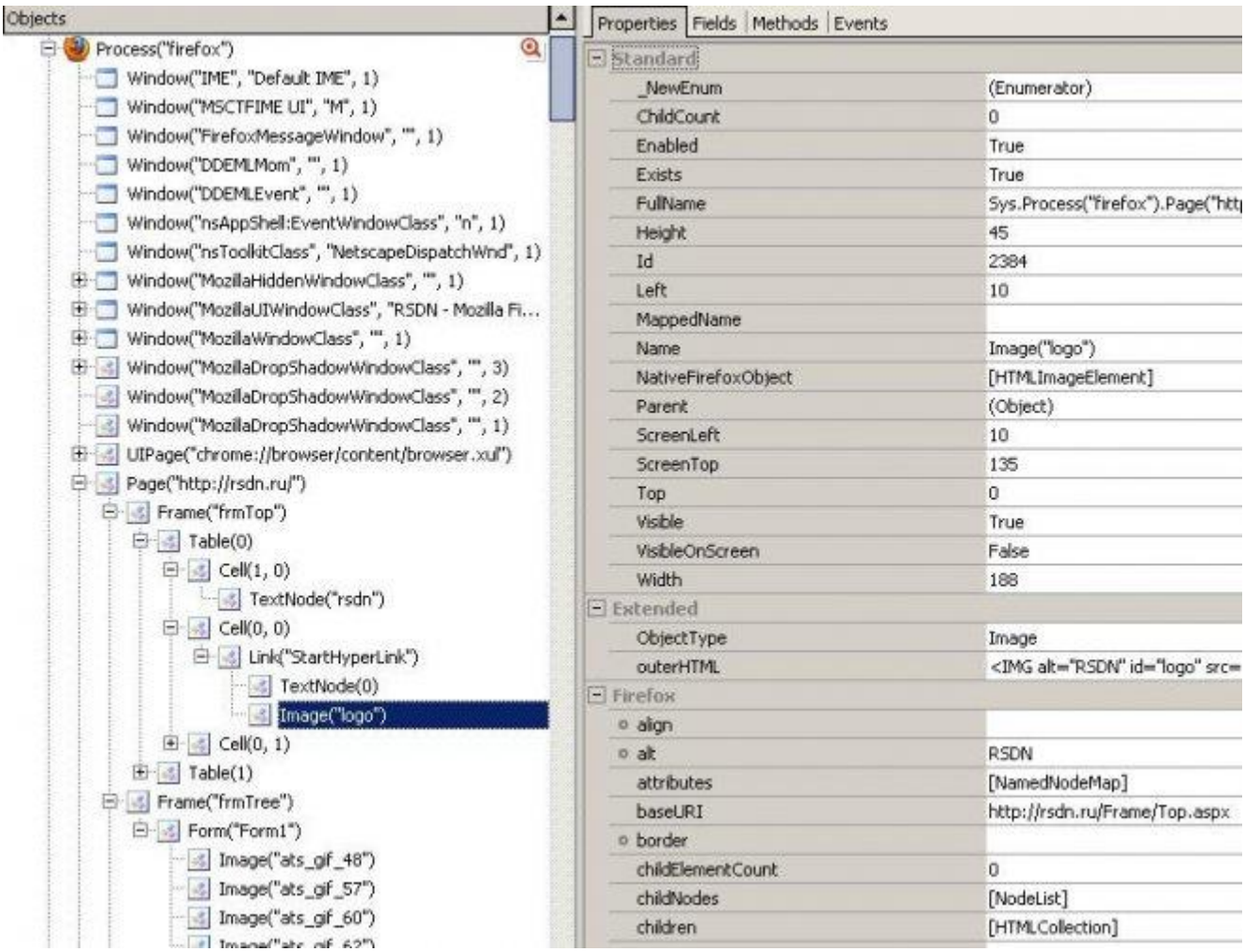
Tag-модель. В случае использования Tag модели все элементы управления тестируемого приложения сгруппированы по тегам (например, все изображения будут находиться в группе IMG, все ссылки – в группе A и т.д.).



Имена элементов также имеют вид Item(Index), где Index может быть Id, имя объекта или, если ни одно из этих свойств не указано, имя группы плюс порядковый номер, разделенные подчеркиванием (например, A_2, IMG_23).

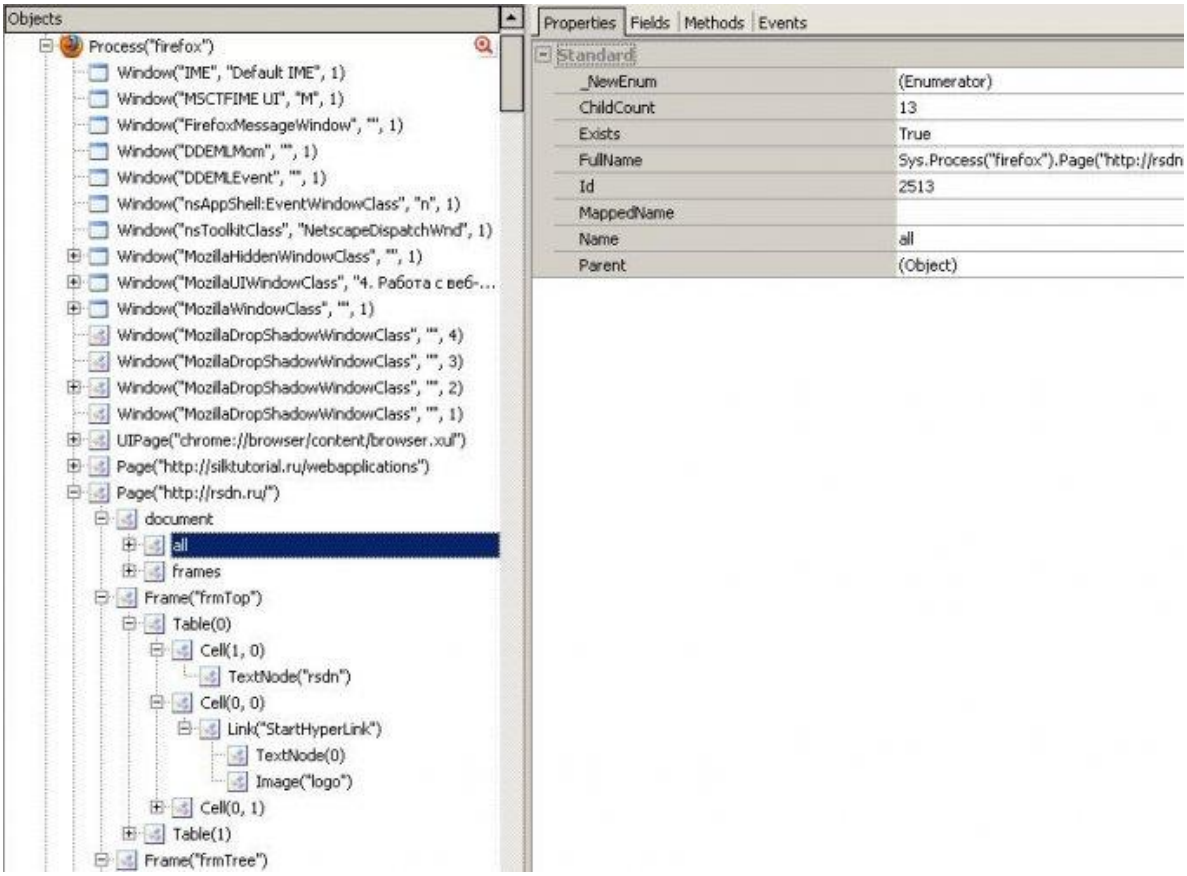
Если у вас возникает необходимость работать с многими элементами одного типа (например, перебирать все ссылки или картинки на странице), модель Tag будет наиболее быстрой и удобной в работе.

Tree-модель. При использовании модели Tree объекты в Object Browser-е отображаются в такой же иерархии, как они находятся на странице приложения. Для доступа к элементам управления используются имена типов объектов, а не их теги.



В случае использования Tree-модели некоторые элементы могут не отображаться в Object Browser-е, если в них отсутствует текст (например, элемент SPAN). Модель Tree рекомендуется использовать в том случае, если на страничках тестируемого приложения находится много элементов управления или если доступ к одним и тем же элементам производится много раз.

Hybrid-модель. Эта модель является комбинацией моделей Tree и DOM. В случае ее использования объекты в Object Browser-е отображаются таким образом:



Эта модель введена для совместимости с более ранними версиями TestComplete, если вы хотите использовать модель Tree. В этом случае вы сможете как записывать новые скрипты (будет использована модель Tree), так и запускать старые (будет использована модель DOM). Естественно, вы можете использовать преимущества DOM-модели и при создании новых скриптов.

Теперь мы напишем один и тот же скрипт с использованием разных моделей. Скрипт будет открывать в браузере Firefox страницу http://ya.ru/ и осуществлять поиск текста "Учебник TestComplete".

```
// Использование DOM модели
function TestDOM()
{
  var  firefox;
  var  page;
  firefox = Sys.Process("firefox");
  page = firefox.Page("*");
  page.ToUrl("http://ya.ru");
  page.Wait();
  page = firefox.Page("http://ya.ru/");
  page.document.all.Item("text_2").Keys("учебник TestComplete");
  page.document.all.Item(45).Click();
  page.Wait();
}

// Использование TAG модели
function TestTAG()
{
  var  firefox;
  var  page;
```

```
  firefox = Sys.Process("firefox");
  page = firefox.Page("*");
  page.ToUrl("http://ya.ru");
  page.Wait();

  page.INPUT.Item("text").Keys("учебник TestComplete");
  page.INPUT.Item("INPUT").Click();
  page.Wait();
}
// Использование TREE модели
function TestTREE()
{
  var  firefox;
  var  page;
  firefox = Sys.Process("firefox");
  page = firefox.Page("*");
  page.ToUrl("http://ya.ru");
  page.Wait();

  page.Table(0).Cell(1, 0).Form(0).Table(0).Cell(0,
1).Panel(0).Textbox("text").Keys("учебник TestComplete");
  page.Table(0).Cell(1, 0).Form(0).Table(0).Cell(0,
2).SubmitButton("Найти").Click();
  page.Wait();
}

// Использование HYBRID модели
function TestHYBRID()
{
  var  firefox;
  var  page;
  firefox = Sys.Process("firefox");
  page = firefox.Page("*");
  page.ToUrl("http://ya.ru");
  page.Wait();

  page.document.all.Item("text_2").Keys("учебник TestComplete");
  page.Table(0).Cell(1, 0).Form(0).Table(0).Cell(0,
2).SubmitButton("Найти").Click();
  page.Wait();
}
```

Как видно из этого примера, в последнем случае использования Hybrid модели для доступа к текстовому полю и кнопке "Найти" мы воспользовались разными способами доступа (DOM и Tree).

Интересной особенностью использования разных моделей является то, что их можно менять в процессе выполнения скрипта. Для этого используется объект Options. Например, следующая функция запустит все 4 написанных ранее функции по очереди, для каждой устанавливая необходимую модель объектов.

```
function TestDifferentModels()
```

```
{
  Options.Web.TreeModel = "DOM";
  TestDOM();
  Options.Web.TreeModel = "Tag";
  TestTAG();
  Options.Web.TreeModel = "Tree";
  TestTREE();
  Options.Web.TreeModel = "Hybrid";
  TestHYBRID();
}
```

В дальнейшем в примерах мы будем использовать Tag модель совместно с Flat моделью проекта.

Если вы попытаетесь записать скрипт с помощью встроенных в TestComplete средств записи, вы увидите, что сгенерированный скрипт выглядит не так красиво и понятно, как приведенные выше примеры. Например:

```
function TestRecording()
{
  var  firefox;
  var  wndMozillaUIWindowClass;
  var  page;
  firefox = Sys.Process("firefox");
  wndMozillaUIWindowClass = firefox.Window("MozillaUIWindowClass", "Mozilla
Firefox");
  wndMozillaUIWindowClass.Window("MozillaWindowClass", "",
1).Keys("~![ReleaseLast][ReleaseLast]ya.[Down][Enter]");
  //Please wait until download completes: "http://ya.ru/"
  page = firefox.Page("http://ya.ru/");
  page.Wait();
  wndMozillaUIWindowClass.Window("MozillaWindowClass", "",
4).Keys("~![ReleaseLast][ReleaseLast]учебник
~![ReleaseLast][ReleaseLast]TestComplete");
  page.Table(0).Cell(1, 0).Form(0).Table(0).Cell(0,
2).SubmitButton("Найти").Click();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
  //Please wait until download completes:
"http://yandex.ua/yandsearch?rdrnd=702097&text=%D1%83%D1%87%D0%B5%D0%B1%D0%BD
%D0%B8%D0%BA%20TestComplete&lr=141"
  page.Wait();
}
```

Это происходит потому, что TestComplete при записи действий записывает ВСЕ действия. В данном случае он записал ввод текста вместе с комбинацией клавиш переключения раскладки клавиатуры (*Alt+Shift*), причем метод Keys записался для всего окна, а не для текстового поля.

Записанный скрипт является нечитабельным даже в таком простом примере, так что можете представить себе, насколько запутанным он будет в случае записи более сложного примера. Именно поэтому мы не рекомендуем пользоваться средствами записи действий особенно при тестировании веб-приложений. Если вы не знаете, как обратиться к тому или иному элементу управления, проще посмотреть путь к нему в окне Object Properties и скопировать полное имя оттуда, чем пользоваться средствами записи.

Тестирование в разных браузерах

TestComplete поддерживает несколько браузеров для тестирования веб-приложений:

- Internet Explorer (версии 5 – 8)
- Mozilla Firefox (версии 1.5.0.1 – 3.x)
- Netscape Navigator (8.1.2, ограниченная поддержка)

Кроме того, поддерживается любой браузер, созданный с использованием элемента управления MS Web Browser (например, Avant Browser, MyIE).

Однако каждый браузер может добавлять свои собственные свойства и методы для работы с элементами веб-страниц, которые не будут работать в других браузерах.

Если открыть в Object Browser-е любую страничку в двух разных браузерах (например, FF и IE), то вы увидите, что свойства находятся в разных секциях. Есть секция Standard, в которой находятся свойства, доступные во всех браузерах без исключения. Некоторые свойства (например, innerText) доступны во всех браузерах, так как они являются стандартными свойствами веб-элементов, однако они будут находиться в секции Firefox (или Internet Explorer). Например, свойство value для текстового поля всегда содержит текст поля.

Поэтому каждый раз, используя какое-то новое свойство, прежде всего проверяйте, доступно ли это свойство в других браузерах и содержит ли оно в разных браузерах одинаковое значение.

Основы разработки скриптов при тестировании веб-приложений

Теперь рассмотрим более подробно процесс создания скриптов при тестировании веб-приложений.

Как вы уже поняли из предыдущих примеров, доступ к элементам веб-страниц производится через объект Page, который является дочерним объектом процесса браузера, например:

```
Sys.Process("firefox").Page("about:blank")
Sys.Process("IEXPLORE").Page("http://ya.ru/")
```

Для того чтобы открыть какой-то адрес внутри страницы, необходимо использовать метод ToUrl. Этот метод открывает указанный адрес, дожидается окончания загрузки страницы и возвращает новую копию объекта Page. Например,

```
var page = Sys.Process("firefox").Page("about:blank");
page =page.ToUrl("http://ya.ru/");
```

Если вам необходимо только открыть страничку и не дожидаться окончания ее завершения, можно воспользоваться методом NavigateTo:

```
page. NavigateTo ("http://ya.ru/");
```

Для того, чтобы узнать текущий адрес какой-то странички, необходимо получить значение его свойства URL:

```
var sCurrentURL = page.URL;
```

Если на страничке были сделаны какие-то изменения, которые требуют обновления данных на странице, можно воспользоваться методом Wait.

```
page.Wait();
```

Кроме перечисленных свойств и методов, можно также использовать методы WaitItem и методы Find (подробнее об этих методах можно прочитать в главе 3.5 Синхронизация выполнения скриптов).

Так как принципы работы с этими методами такие же, как и для любых других приложений, мы здесь не будем подробно останавливаться на них.

Создание чекпоинтов

При записи или написании скриптов, работающих с веб-страницами, можно создавать 2 типа специфических чекпоинтов.

1. **Web Accessibility checkpoint.** Этот чекпоинт позволяет выполнить проверку параметров всех элементов страницы (таких как alt image, tab order, links accessibility, images sizes и т.п.). Т.е. будут выполнены проверки на то, что ссылки доступны, картинки имеют заданные атрибуты height и width и т.п.
2. **Web Comparison Checkpoint**. Этот чекпоинт позволяет сравнивать две HTML-странички либо целиком, либо только структуру тегов (всех или выбранных).

Создание этих чекпоинтов похоже на создание любых других чекпоинтов (например, Image Checkpoint или Property Checkpoint). Естественно, кроме этих специфических чекпоинтов можно создавать и более общие (например, те же Property Checkpoint-ы).

Для запуска верификации Web чекпоинтов необходимо вызвать метод

```
WebTesting.WebAccessibility1.Compare();
```

или

```
WebTesting.WebComparison1.Compare();
```

где WebAccessibility1 и WebComparison1 – соответственно имена созданных чекпоинтов.

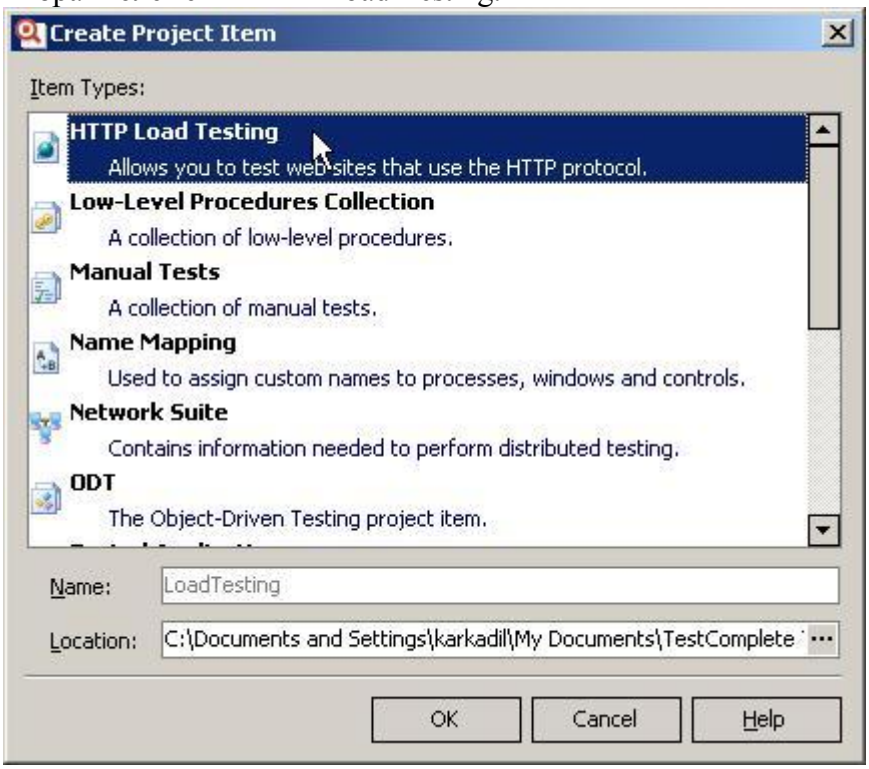
4.2 Нагрузочное тестирование Web-приложений

TestComplete позволяет проводить нагрузочное тестирование приложений, которые для передачи данных используют протоколы HTTP, HTTPS или SOAP (т.е. в основном web-приложений). Для этого сначала записывается трафик, который генерируется при общении локального компьютера с сервером, а затем, используя записанный трафик, эмулирует действия пользователей. При этом неважно, какой браузер используется при записи, а при воспроизведении записанных скриптов можно эмулировать любой браузер.

Общая схема при создании нагрузочных тестов:

1. Записать действия
2. Внести изменения в записанный трафик
3. Создать скрипты, которые будут запускать записанные действия
4. Проанализировать результаты запуска

Прежде чем начать запись нагрузочных тестов, необходимо добавить в проект соответствующий элемент HTTP Load Testing. Для этого щелкнем правой кнопкой мыши на имени проекта, выберем пункт меню *Add – New Item* и в появившемся диалоговом окне выбрать элемент HTTP Load Testing.



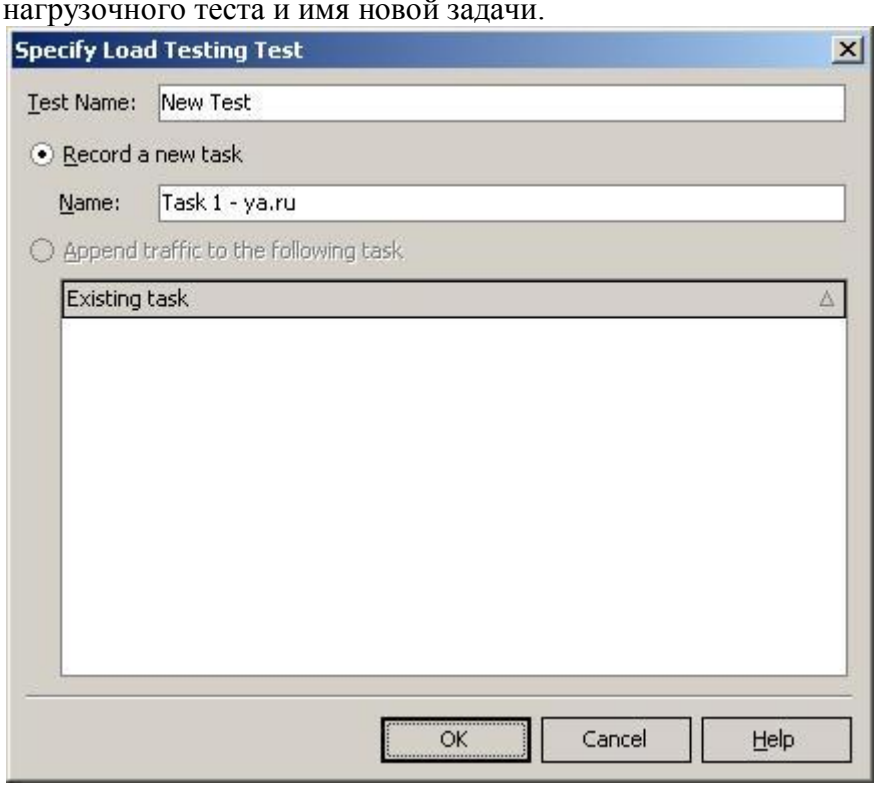
После этого в нашем проекте появится новый элемент LoadTesting с тремя дочерними элементами:

1. **Station** – здесь перечислены все рабочие станции, используемые для нагрузочного тестирования (по умолчанию используется только один локальный компьютер – Master)
2. **Tasks** – задачи, т.е. запросы, которые отправляются серверу, и ответы от него
3. **Tests** – нагрузочные тесты. Их можно создавать как визуально в редакторе, так и в скриптах

Теперь запишем простой тест.

Для этого мы воспользуемся сайтом http://ya.ru/ и браузером Firefox. Перед началом записи мы откроем в Firefox только один этот сайт.
В TestComplete начнем запись (*Test – Record – Record Script*) и на появившейся панели Recording нажмем кнопку Record an HTTP Task.

При этом появится окно Specify Load Testing Test, в котором мы введем имена для нагрузочного теста и имя новой задачи.



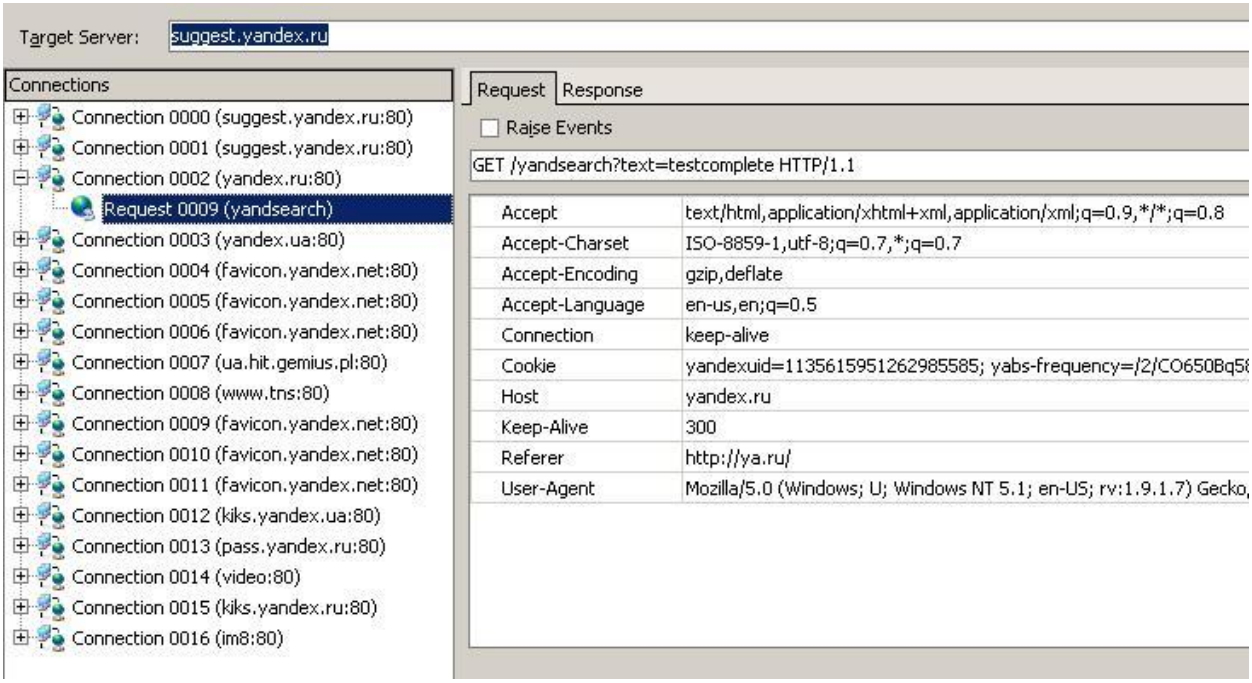
После этого нажмем ОК и начнем выполнение действий, которые хотим записать для нагрузочного теста. При этом записывается только трафик, а работа с элементами управления (например, нажатия на кнопки, перемещение мыши и т.п.) игнорируются.

Для записи простого теста перейдем в браузер (где у нас уже открыт сайт ya.ru) введем в строку поиска какой-нибудь текст (например, " testcomplete") и нажмем кнопку Найти. Как только страница с результатами поиска полностью загрузится, остановим запись и посмотрим, что в итоге записалось.

Собственно, скрипт содержит лишь одну строку, запускающую нагрузочный тест:

```
function Test4()
{
  LoadTesting.Tests.TestByName("New Test").Execute();
}
```

В Tasks добавлена задача "Task 1 – ya.ru". Если дважды щелкнуть по этой задаче, можно посмотреть ее подробности.



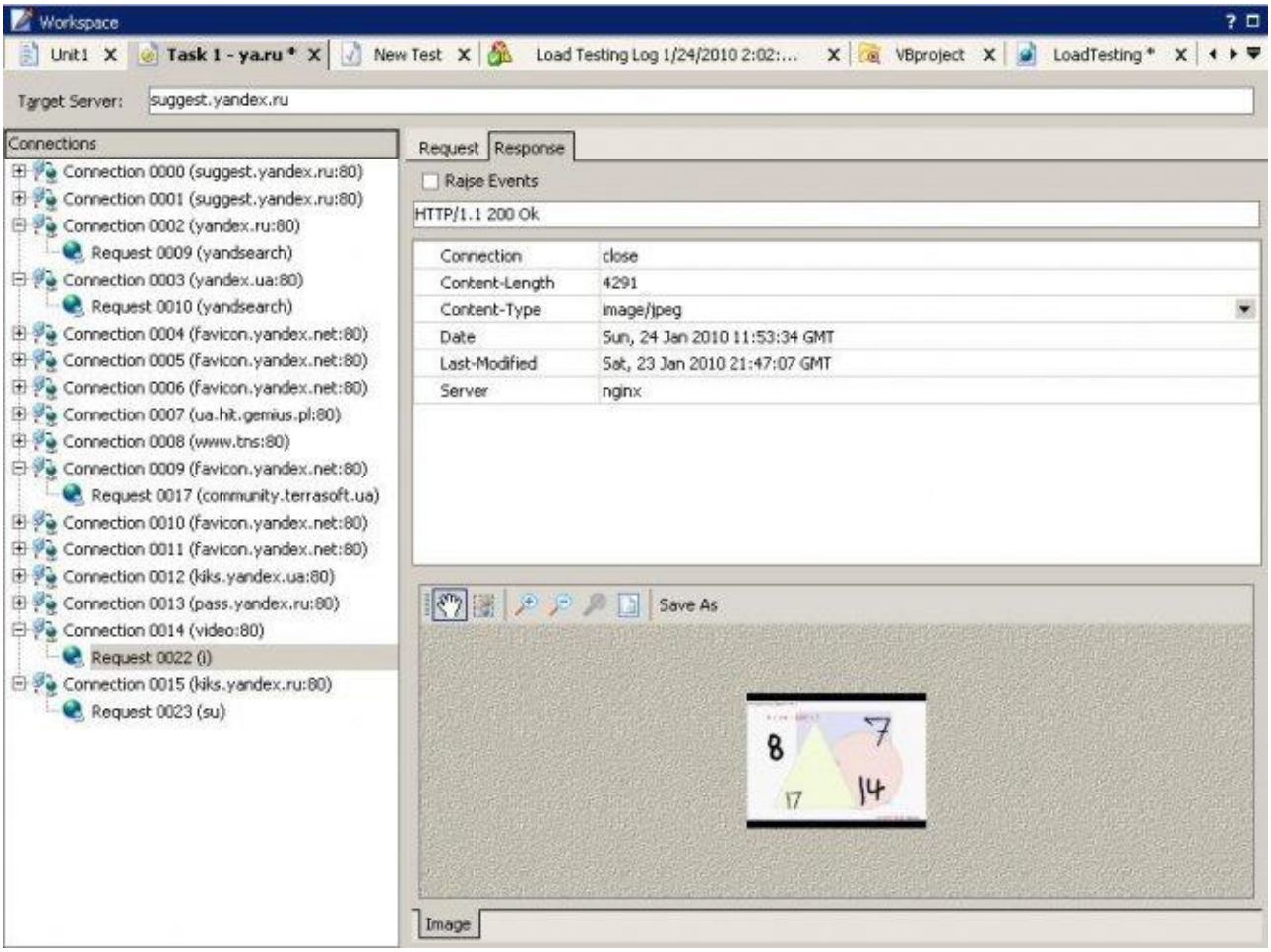
Поле **Target Server** позволяет задать имя сервера, на котором проводится тестирование. Это может быть полезно, например, в том случае, если запись скриптов производится на одном сервере, а запуск будет проводится на другом. В этом случае нам не придется менять имя сервера для каждого подключения, а изменить его один раз в этом поле.

В списке **Connections** слева мы видим список записанных подключений, то есть запросов к серверу и ответов от него. Как видно из нашего примера, запросы могут отправляться не к одному серверу, а к нескольким (например, если на сайте установлена реклама, которая берется из других источников, расположенных на других серверах). Обычно такие подключения не нужно тестировать, а потому их можно смело удалить из задачи. Для этого необходимо щелкнуть правой кнопкой по ненужному подключению и выбрать пункт Delete.

В правой части панели отображаются 2 вкладки – **Request** (Запрос) и **Respond** (Ответ). Это запросы и ответы для выбранного в списке слева подключения. Галочка Raise Events на каждой вкладке позволяет включить генерацию OnLoadTestingRequest и OnLoadTestingResponse события соответственно. По умолчанию генерация этих событий отключена в TestComplete, так как включение этого события для всех подключений может существенно замедлить выполнение скриптов.

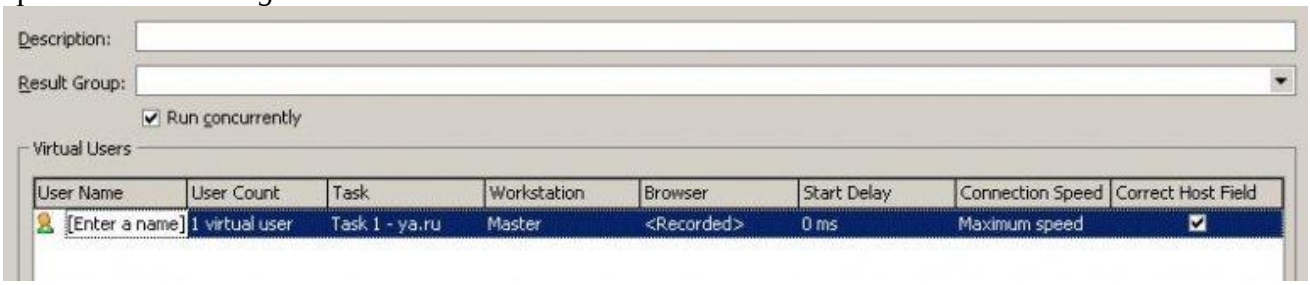
На вкладке Request находится поле **Request Method**,в котором указывается использованный метод (GET или POST), запрошенная страница с параметрами (например, /yandsearch?text=testcomplete) и протокол доступа (например, HTTP/1.1). Все эти параметры можно при необходимости изменить непосредственно в задаче или из скриптов.

Также на вкладке Request находятся все параметры подключения, которые также можно при необходимости менять. Например, параметр User-Agent позволяет задать, какой браузер будет использовать при эмуляции подключений.
На вкладке Response также перечислены параметры ответа от сервера и его содержимое. Например, на скриншоте ниже показан пример ответа от сервера (картинка):



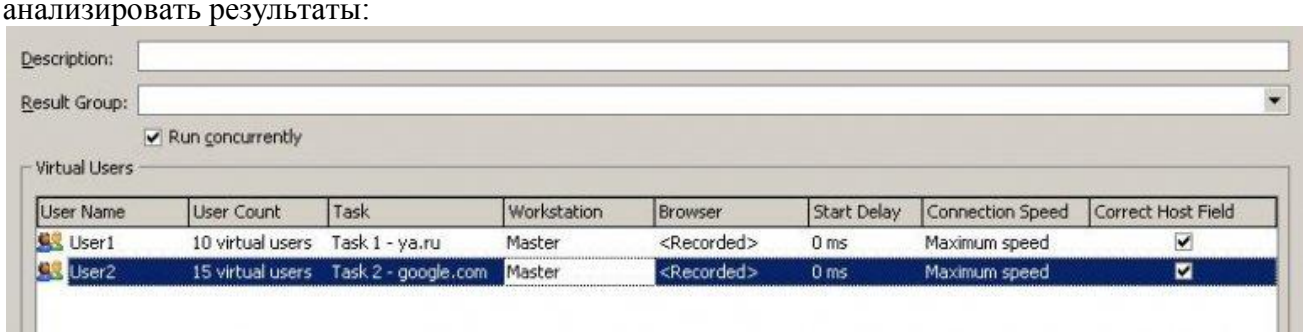
Теперь рассмотрим нагрузочные тесты.

При записи нагрузочного теста TestComplete автоматически создал один тест (мы его назвали New Test),его параметры можно увидеть, если дважды щелкнуть на элементе проекта *LoadTesting – Tests – New Test*.



В каждом тесте может быть несколько пользователей (поле User Name), каждый из которых в свою очередь может запускать сразу несколько экземпляров нагрузочного теста (количество одновременных запусков определяется полем User Count, а задача – полем Task). Также для каждого пользователя можно указать компьютер, который будет использоваться для запуска записанного трафика (поле Workstation), браузер, который будет использоваться при эмуляции (Browser), задержка старта в миллисекундах (Start Delay), скорость подключения (Connection Speed), а также указать, необходимо ли для данного пользователя исправлять Host Field (если галочка включена, то Host Field будет заменяться на значение из поля Target Server).

Так как результаты отображаются отдельно для каждого пользователя, рекомендуется давать виртуальным пользователям легко отличимые имена, чтобы легче было анализировать результаты:

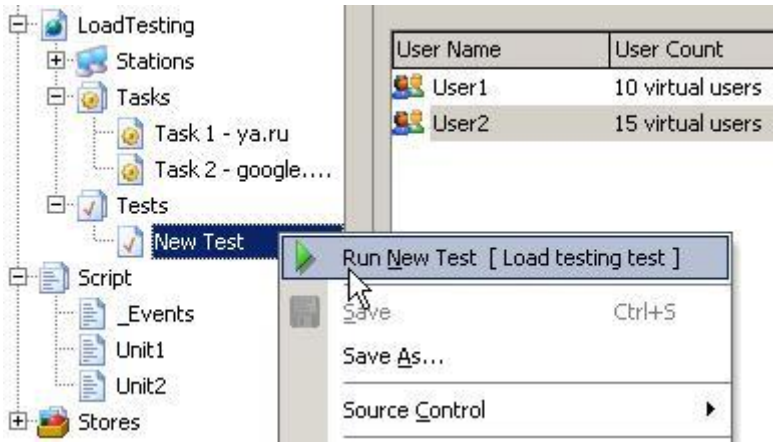


Обратите внимание на опцию **Run Concurrently** в окне нагрузочного теста. Если эта опция включена, то при запуске этого нагрузочного теста все задачи виртуальных пользователей будут запускаться одновременно, а если опция выключена – то задачи будут запущены по очереди.

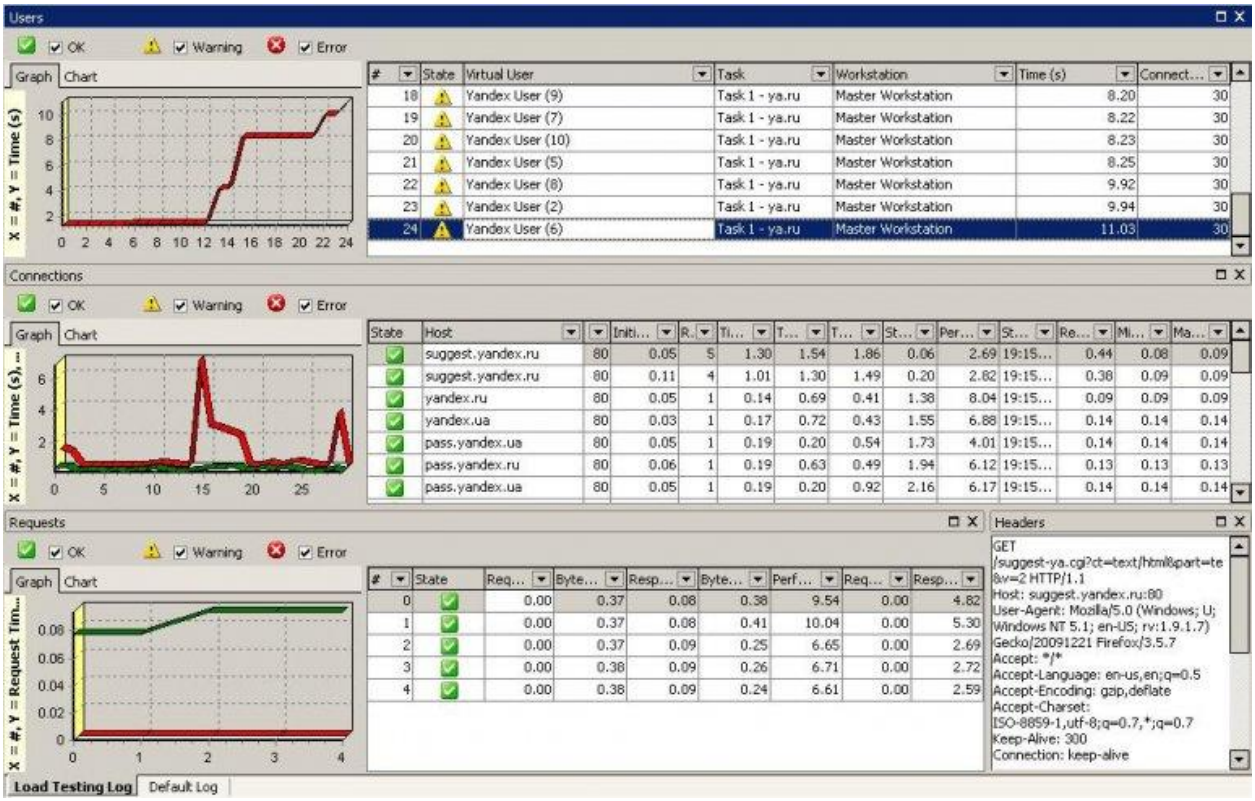
Запуск тестов и анализ результатов

Точно так же, как мы создали нагрузочный тест для сайта ya.ru раньше в этой главе, мы создали аналогичный тест для сайта google.com, добавили в тест New Test еще одного пользователя (см. скриншот выше) и теперь готовы запустить тесты.

Для этого щелкнем правой кнопкой мыши на тесте в дереве проекта и выберем пункт меню Run.



После того, как тест отработает, мы увидим результаты в таком виде:



Лог состоит из двух закладок (в левой нижней части экрана):

- **Load testing Log** – на этой вкладке можно посмотреть различные результаты по каждому пользователю и подключению
- **Default Log** – здесь отображаются все ошибки, предупреждения и сообщения по каждому подключению

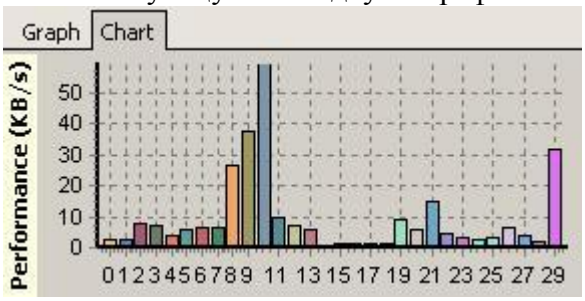
Рассмотрим внимательно вкладку **Load Testing Log.** Здесь находятся три панели:

1. **Панель пользователей.** По оси X на графике отображаются пользователи, а по оси Y – время, которое понадобилось каждому пользователю на выполнение теста. В приведенном выше примере видно, что первому пользователю понадобилось чуть больше одной секунды для выполнения теста, а последнему – 11 секунд.
   В таблице справа от графика можно посмотреть, какую задачу выполнял каждый пользователь, сколько в точности времени понадобилось каждому пользователю времени и сколько подключений они сделали за время выполнения теста. Например, из результатов, показанных выше, видно, что на выполнение запросов к Гуглу тратилось от 1 до 4 секунд, а на запросы к Яндексу – от 8 до 11 секунд.
   При выборе какой-либо строки в таблице пользователей изменяется информация на второй панели – панели подключений.
2. **Панель подключений.** Здесь можно посмотреть подробную информацию по каждому подключению для пользователя, выбранного в первой панели. Информация также отображается в виде таблицы и наглядного графика.
   Например, в приведенном выше примере мы видим, что больше всего времени занимают запросы к серверу kiks.yandex.ua. Это может послужить поводом для выяснения причин такого долгого ответа от сервера, так как полученный результат существенно отличается от средних результатов запросов к другим серверам, а значит именно этот сервер может быть "слабым звеном".
   В таблице Connections отображается очень много информации по каждому подключению (например, время, количество переданных и полученных байт, производительность и т.п.),

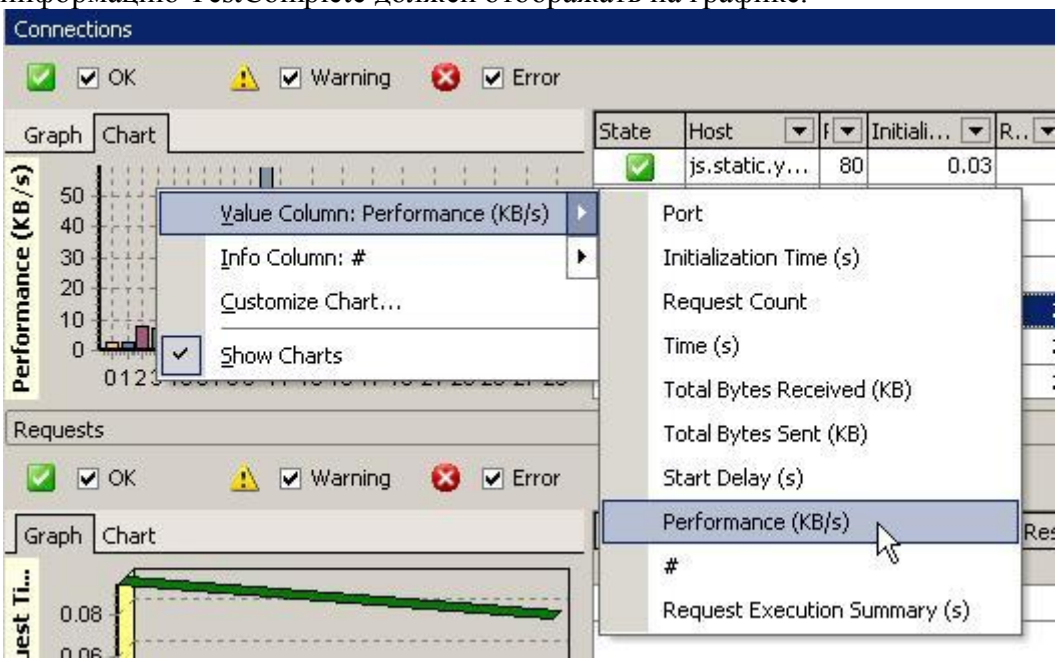
поэтому заголовки столбцов трудно прочитать. Внимательно посмотрите, возможно некоторые из колонок вам не нужны, тогда их можно скрыть, щелкнув правой кнопкой мыши на заголовке таблице и выбрав пункт Field Chooser.

3. **Панель запросов.** В этой панели можно посмотреть подробную информацию по каждому запросу выбранного подключения.

По умолчанию графики представляются в виде обычного графика, однако можно выбрать представление в виде столбчатой диаграммы (chart). Для этого надо просто выбрать соответствующую закладку на графике.

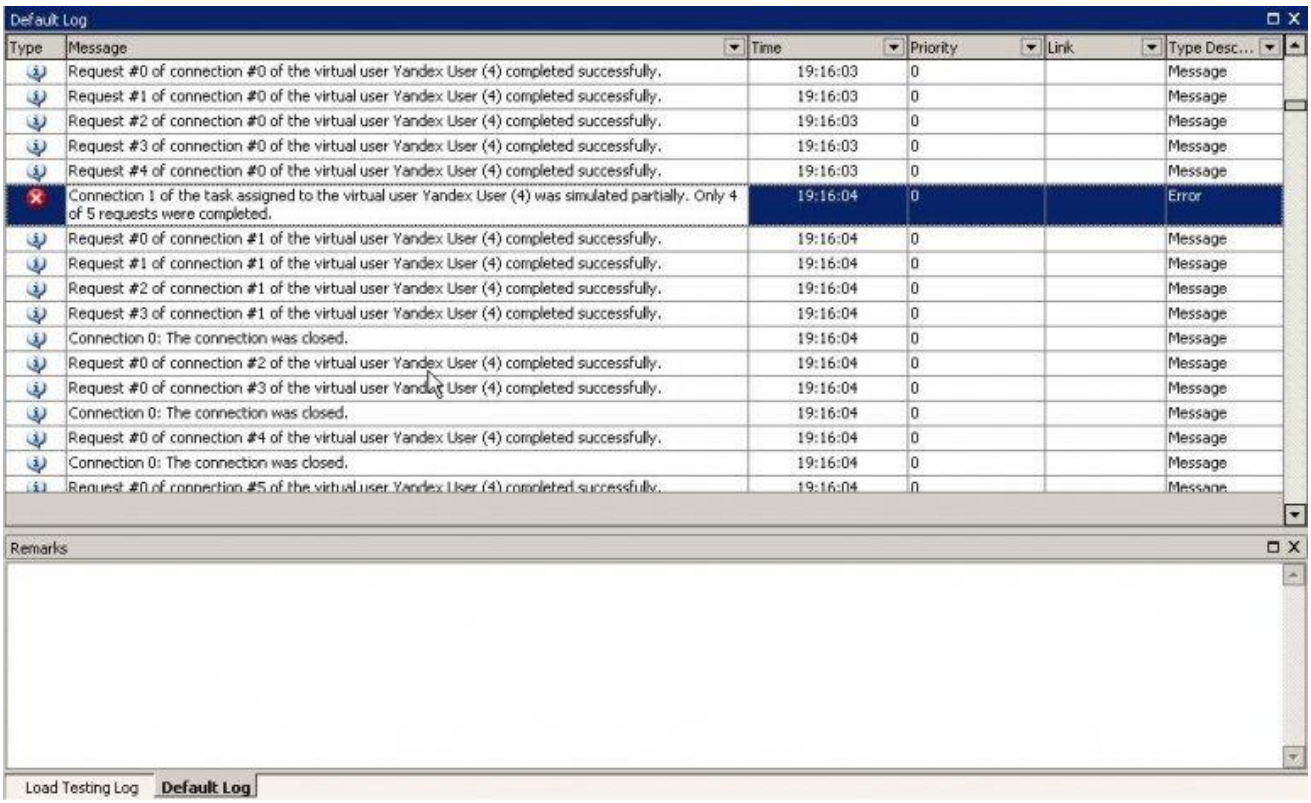


Затем, щелкнув правой кнопкой мыши на диаграмме, можно выбрать, какую именно информацию TestComplete должен отображать на графике.



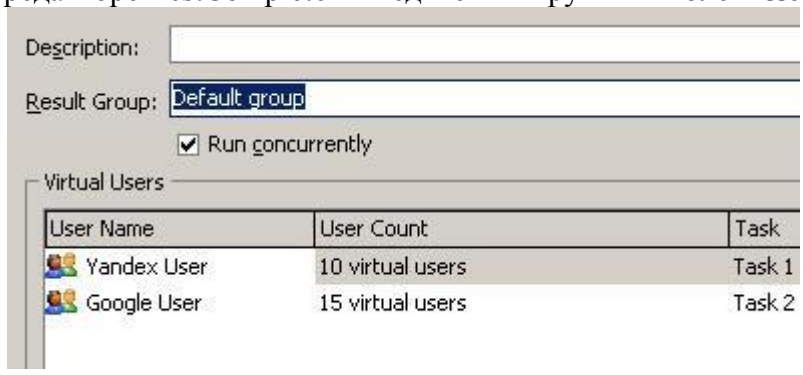
Использование диаграммы вместо графика удобно тем, что при помощи щелчка мышью по какому-либо элементу графика, TestComplete автоматически подсвечивает сообтветствующую строку в таблице справа.

Вкладка Default Log представляет собой обычный лог TestComplete-а, в котором можно посмотреть информацию об ошибках, предупреждениях и сообщения о выполненных подключениях.



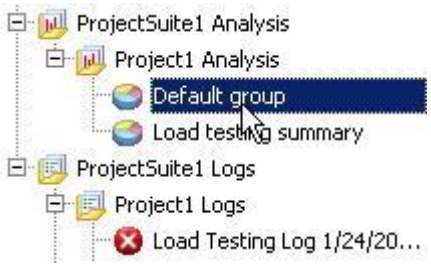
Сравнение результатов нагрузочного тестирования

Если вы хотите сравнивать результаты нагрузочного тестирования какого-то теста, вам необходимо сначала включить этот тест в группу. Для этого откройте нагрузочный тест в редакторе TestComplete и введите имя группы в поле **Result Group:**

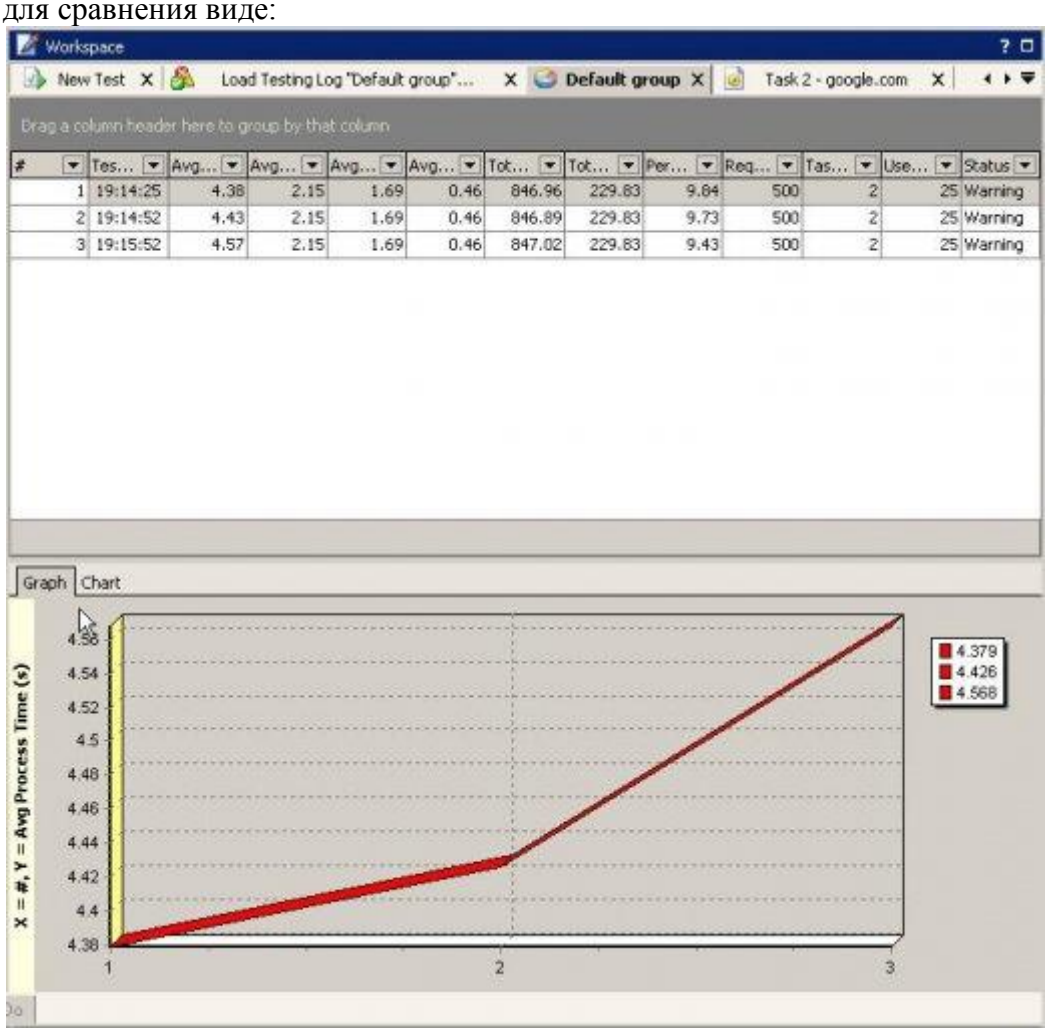


Теперь после нескольких запусков вашего теста, вы можете сравнить полученные результаты друг с другом. Это может быть полезно, если в приложении были сделаны изменения, которые должны были ускорить работу приложения, или наоборот, сделанные изменения могли повлиять на производительность и необходимо проверить насколько сильно.

Для сравнения результатов какой-либо группы в дереве проекта выберите пункт *ProjectSuite Analysis – Project Analysis – Group Name*:



При этом откроется окно, в котором результаты запусков теста представлены в удобном для сравнения виде:



Модификация нагрузочных тестов и задач из тестовых скриптов

До сих пор мы работали с нагрузочными тестами через редактор TestComplete, однако может возникнуть необходимость вносить изменения, так сказать, "на лету", то есть во время работы скрипта.

Например, если мы тестируем поиск на сайте, а сервер кеширует результаты поиска, то нагрузка фактически будет даваться на сервер один раз: при первом выполнении запроса. Все следующие результаты будут выданы из кеша.

В такой ситуации нам надо для каждого запроса к серверу формировать новую строку, поиск которой будет производиться в нагрузочных тестах.

В следующем примере мы создадим 5 виртуальных пользователей, каждому из которых назначим задачу (Task) и нагрузочный тест (Test, который также будет создан в скрипте),

а затем запустим созданный нагрузочный тест, одновременно назначив ему группу (чтобы можно было сравнивать результаты прогона тестов).

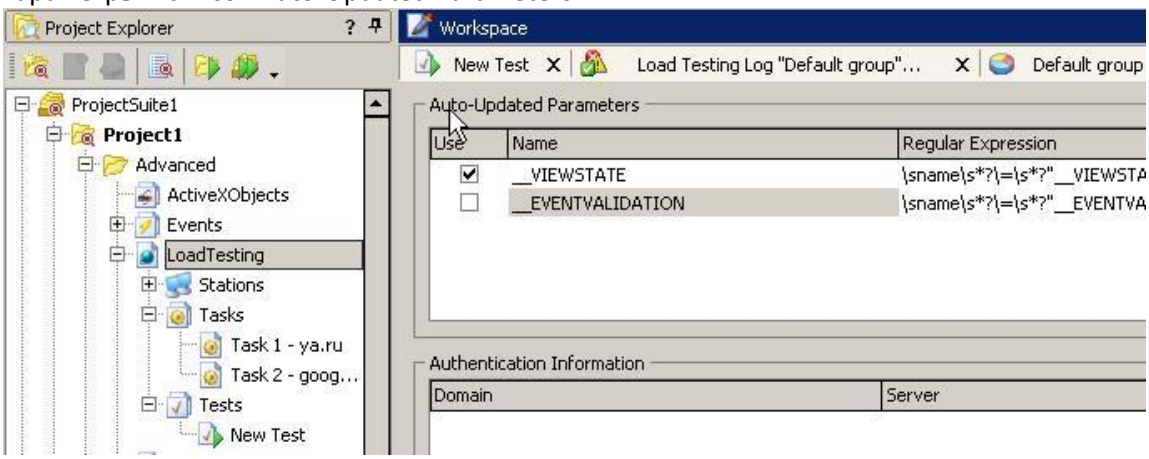
```
function TestLoadTesting()
{
  var i;
  // Создаём новый тест
  var ti = LoadTesting.CreateTestInstance("New Created Test");

  // массив для пользователей
  var aUsers = new Array(5);

  // назначаем каждому пользователю задачу и тест
  for(i = 0; i < aUsers.length; i++)
  {
    aUsers[i] = LoadTesting.CreateVirtualUser("New created user " +
i.toString());
    aUsers[i].Task = LoadTesting.HTTPTaskByName("Task 2 - google.com");
    aUsers[i].TestInstance = ti;
  }

  // запускаем нагрузочный тест
  ti.Run("New result group");
}
```

А в следующем примере мы модифицируем ранее записанную задачу для каждого пользователя таким образом, чтобы каждый из виртуальных пользователей искал не просто строку "testcomplete", а строку "textcompleteN", где N – порядковый номер пользователя (вместо порядкового номера можно, например, написать функцию, которая будет генерировать случайную строку по заданному образцу, мы используем пример с порядковым номером лишь для упрощения примера).

```
function TestLoadTesting()
{
  var i;
  var sHeader;
  // Создаём новый тест
  var ti = LoadTesting.CreateTestInstance("New Created Test");

  // массив для пользователей
  var aUsers = new Array(5);

  // назначаем каждому пользователю задачу и тест
  for(i = 0; i < aUsers.length; i++)
  {
    aUsers[i] = LoadTesting.CreateVirtualUser("New created user " +
i.toString());
    aUsers[i].Task = LoadTesting.HTTPTaskByName("Task 2 - google.com");
    aUsers[i].TestInstance = ti;

    // модифицируем задачу для каждого пользователя
    sHeader = aUsers[i].Task.Connection(0).Request(0).RequestHeader;
    aUsers[i].Task.Connection(0).Request(0).RequestHeader =
aqString.Replace(sHeader, "testcomplete", "testcomplete" + i.toString());
  }

  // запускаем нагрузочный тест
  ti.Run("New result group");
}
```

Несколько важных замечаний по нагрузочному тестированию

1. При записи нагрузочных тестов убедитесь, что работает только одно приложение, работающее по HTTP, HTTPS или SOAP протоколу, - то, в котором записываются тесты. Например, приложение MSN Messenger также работает по HTTP протоколу и при передаче данных может повлиять на записанные тесты. Также в настройках TestComplete можно указать списки приложений или хостов, трафик с которых TestComplete будет игнорировать (*Tools – Options – Engines – HTTP Load Testing*)
2. TestComplete не может эмулировать более 300 виртуальных пользователей на одном компьютере. Если вам необходимо эмулировать большее количество виртуальных пользователей, вам придется распределить нагрузочное тестирование на несколько компьютеров. Как это сделать, рассказано в главе 15 Распределенное тестирование.
3. Если вы тестируете приложение, написанное на ASP.NET, то некоторые элементы управления могут иметь параметры, которые постоянно меняются, а значит ответы от сервера при записи и воспроизведении будут всегда отличаться. Чтобы решить эту проблему, дважды щелкните на элементе LoadTesting в дереве проектов и добавьте эти параметры в список Auto-Updated Parameters



   Здесь же можно задать параметры серверов, которые используют NTLM или Kerberos типы аутентификации.
4. Если вы используете прокси-сервер, вам необходимо указать его параметры в опциях TestComplete (*Tools – Options – Engines – HTTP Load Testing*)
5. Если у вас на компьютере установлен антивирус, он может блокировать запись трафика. Обратитесь к справке TestComplete, раздел "TestComplete Network Activities - Possible Issues With Antiviruses" чтобы узнать, как решить эту проблему

4.3 Тестирование Web-сервисов

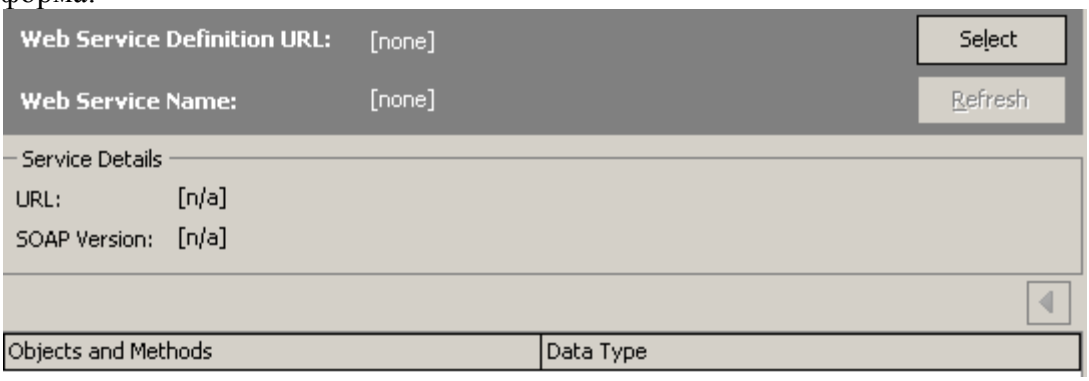
Эта глава изначально была опубликована на сайте Automated Testing Service Group и здесь приводится практически без изменений.

Веб-сервисы – достаточно распространенный вид компонент распределенных систем. Фактически это один из интерфейсов удаленного вызова процедур. То есть, из некоторой программы мы можем отправить запрос на выполнение некоторой операции на стороне сервера и получить результат выполнения операции, при этом графический интерфейс не задействован. Подобные компоненты позволяют реализовать интеграцию между различными системами, которые изначально между собой не связаны. Отчасти это делает веб-сервисы достаточно популярными, как результат – они используются в достаточно большом количестве приложений, а это в свою очередь влечет необходимость их

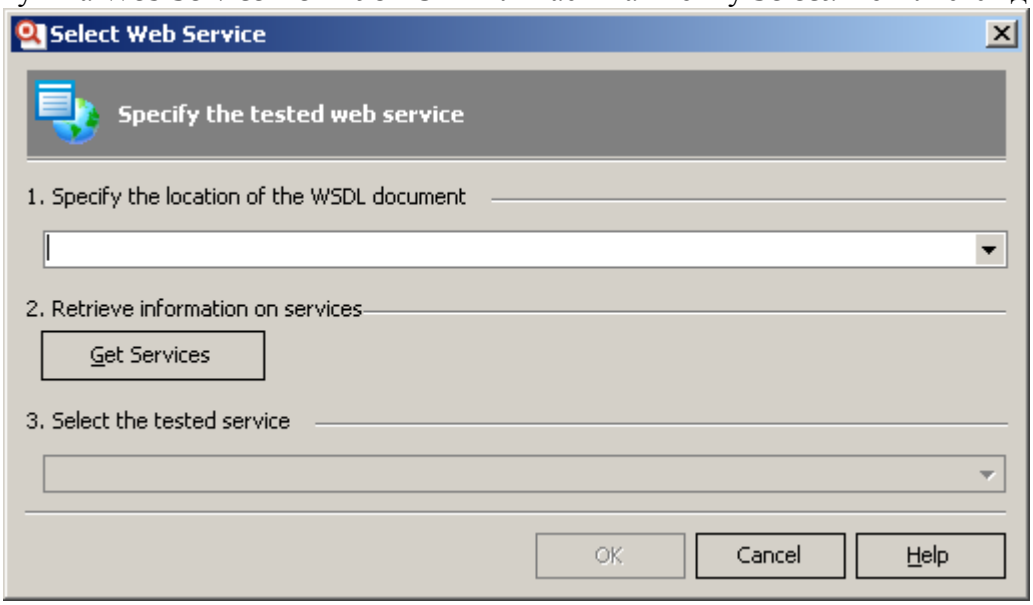
тестировать, в том числе и автоматически. Для этой задачи есть и специальные средства, наподобие soapUI, но помимо этого возможность тестирования веб-сервисов имеется и в TestComplete. В TestComplete, начиная с 6-й версии как раз была добавлена возможность тестирования веб-сервисов. Рассмотрим, как это реализуется и каким образом можно проверить веб-сервисы.

В окне Project Explorer-а кликаем правой кнопкой мыши и выбираем из выпадающего меню *Add* > *New Item*. Появится окно со списком доступных для создания элементов. В этом списке находим Web Services и жмем OK. В результате в нашем проекте появится раздел WebServices, куда в дальнейшем мы будем добавлять наши веб-сервисы.

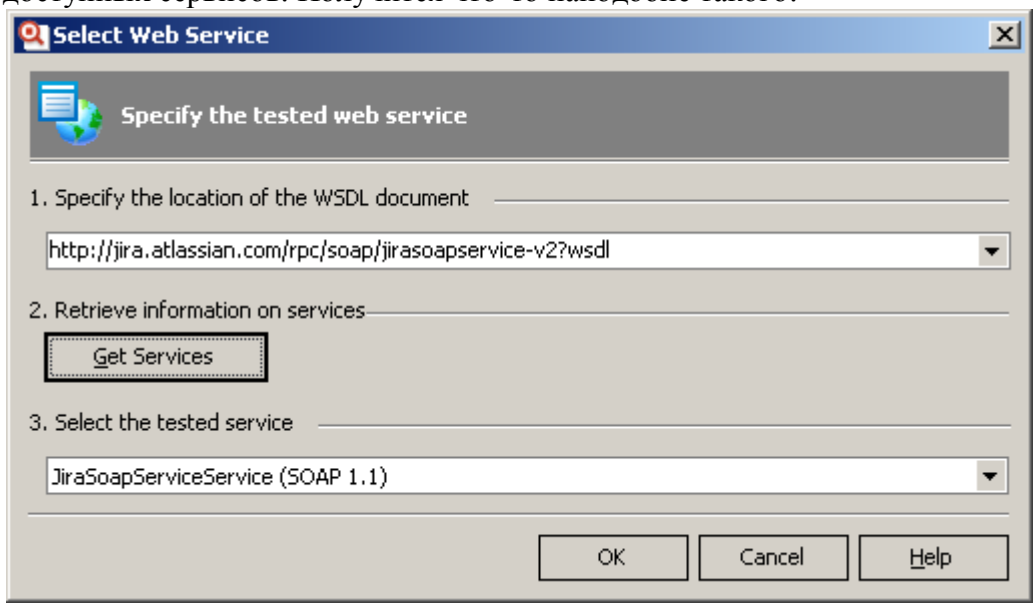
В целях демонстрации работы с веб-сервисами мы воспользуемся веб-сервисами Jira, описания которых доступны по адресу http://jira.atlassian.com/rpc/soap/jirasoapservice-v2 . Это общедоступный ресурс, который используется в целях ознакомления. Итак, добавим данный веб-сервис. Для этого на элементе WebServices сделаем клик правой кнопкой мыши и в выпадающем меню выберем *Add* > *New Item*. В списке возможных типов элементов будет только элемент типа веб-сервис. Кликаем на нем, задаем имя нового веб-сервиса, например, JiraSOAP и нажимаем OK. В результате у нас появляется новый элемент в разделе WebServices. Сделаем на нем двойной клик. В результате отображается форма:



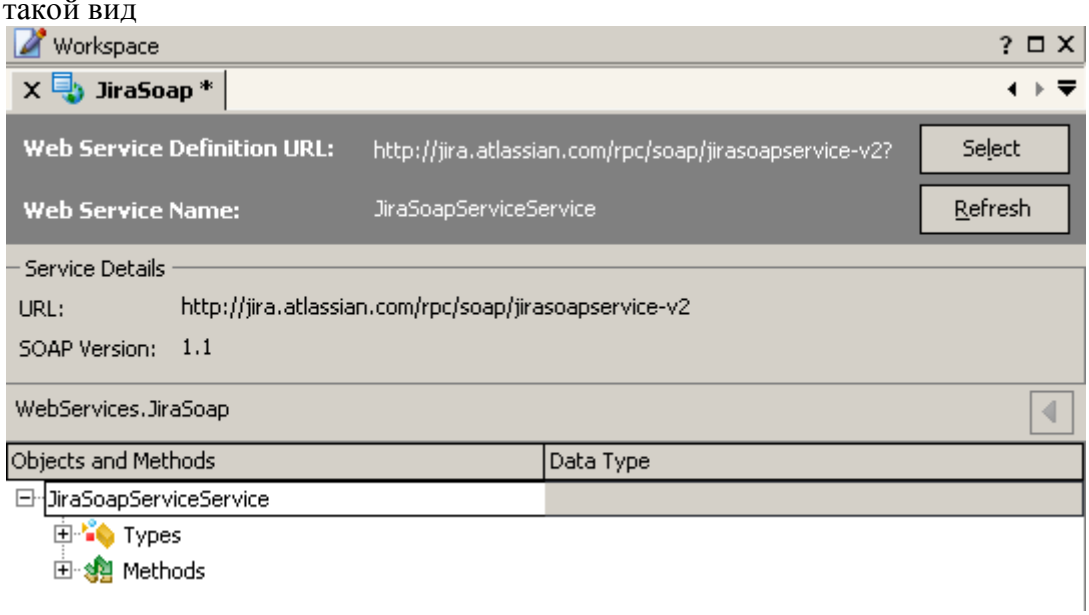
Эта форма пока что пустая, так как мы не задали необходимых реквизитов, как путь к файлу с определениями веб-сервисов, а также имя используемого сервиса. Напротив пункта Web Service Definition URL кликаем на кнопку Select. Появляется диалог вида:



В поле с указанием "Specify the location of the WSDL document" вводим адрес описания нашего веб-сервиса, в нашем случае это http://jira.atlassian.com/rpc/soap/jirasoapservice-v2 . После ввода адреса, нажимаем на кнопку Get Services, чтобы был извлечен список доступных сервисов. Получится что-то наподобие такого:



Для Jira есть всего один веб-сервис JiraSoapServiceService, в списке сервисов выбираем его и кликаем на кнопку OK. Всё, наш сервис добавлен, теперь форма веб-сервиса имеет такой вид



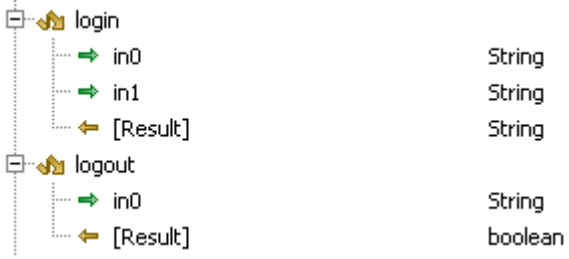
Как видно, все поля заполнены значениями, указывающими адрес сервиса, имя сервиса, а также внизу появились описания типов и методов выбранного сервиса.

Всё, теперь мы можем вызывать имеющиеся веб-сервисы. Сохраним все файлы и создадим новый скриптовый юнит. Назовем его JiraTest. В этом юните создадим функцию вида:

```
function WebServicesSample()
```

```
{
        ;
}
```

Теперь попробуем воспользоваться подключенными веб-сервисами. Для примера залогинимся в Jira и выйдем оттуда. Посмотрим на эти функции. Кликнем еще раз на недавно созданный веб-сервис и посмотрим на список методов:



Нужные нам методы находятся рядом и мы можем их просмотреть вместе. Итак, метод login принимает 2 параметра-строки (непосредственно логин и пароль) и возвращает строку – token, который представляет собой своего рода идентификатор сессии. Практически все остальные методы данного веб-сервиса используют его в качестве первого параметра. Соответственно, метод logout принимает этот самый token, чтобы завершить сессию пользователя. Ссылка на веб-сервис, с которой мы работаем в данном примере - это ссылка на тестовый сайт Jira, в котором помимо всего прочего есть зарезервированный пользователь soaptester с паролем soaptester специально для тестирования доступа к Jira через SOAP. Соответственно, будем заходить под этим пользователем. Если мы вернемся к скриптовому юниту JiraTest, перейдем к созданной выше функции WebServiceSample, наберем WebServices. и нажмем Ctrl + Space, то в выпадающем меню мы получим следующее:

Фактически во встроенный объект был добавлен объект JiraSOAP, содержащий все функции добавленного нами веб-сервиса. Используя его, мы уже вызываем нужные нам методы, в нашем примере это login и logout. Тогда тело функции WebServiceSample имеет вид:

```
function WebServicesSample()
{
      var login = "soaptester";
      var password = "soaptester";

      var token = WebServices.JiraSoap.login( login , password );
      WebServices.JiraSoap.logout( token );
}
```

Как видно из примера, ничего сложного работа с веб-сервисами в TestComplete не представляет. Просто создается интерфейс доступа к нужным методам, всё остальное сводится к знанию спецификаций используемых веб-сервисов.

4.4 Тестирование Flash, Flex и Silverlight приложений

Тестирование Flash, Flex и Silverlight приложений мало чем отличается от тестирования любых других приложений. Вам точно так же необходимо работать с элементами управления (наживать на кнопки, выбирать значения, вводить текст), а потому если вы

читали первые три главы этого учебника, то вам не составит труда создавать скрипты для этих типов приложений. Также желательно прочитать главу 4.1 Функциональное тестирование Web-приложений.

Есть, однако, и особенности при тестировании этих приложений.

1. Web model должна быть установлена в Hybrid или Tree (об этом можно прочитать в главе 4.1 Функциональное тестирование Web-приложений). Если вы используете другую модель в проекте и менять её означает терять многие часы на изменение готовых скриптов, то для тестирования этих приложений можно устанавливать web tree model прямо во время исполнения скриптов с помощью опции Options.Web.TreeModel
2. Полное и корректное распознавание элементов управления в таких приложениях возможно не всегда. Например, в случае, когда элементы управления добавляются динамически, вполне вероятно, что положение элементов на странице будет определяться неправильно, или же считать некоторые данные из них будет невозможно и т.п. Короче говоря, готовьтесь к «сюрпризам»
3. Прежде, чем начинать тестирование этих приложений, необходимо подготовить как само приложение (перевести его в «оконный»), так и TestComplete. Зачастую перевод приложения в «оконный» режим также способствует появлению «сюрпризов», о которых мы писали выше (например, некоторые элементы управления будут перекрывать друг друга, чего не случается в «обычном» режиме работы приложения).Ниже мы вкратце рассмотрим, как необходимо это сделать, а полное описание всех возможных установок можно найти в справке TestComplete-а

Подготовка Flash-приложений

Чтобы элементы управления во Flash-приложении распознавались, необходимо установить опцию Window Mode в режим «Window». Сделать это можно либо в настройках среды разработки, либо вручную прямо в HTML-файле. В разделе **«Preparing Flash Applications for Testing»** справки TestComplete вы найдете все подробности, как это сделать.

Подготовка Flex-приложений

Для тестирования Flex-приложений можно воспользоваться либо технологией MSAA, либо включив Flex Automation API. Первый способ даёт весьма ограниченный доступ к элементам управления внутри приложения по сравнению со вторым, однако в случае подключения Flex Automation API существенно возрастает размер тестируемого приложения (что также сказывается на скорости обновления дерева объектов в TestComplete).

Кроме того, тестируемое приложение необходимо внедрить в веб-страницу и изменить настройки безопасности.

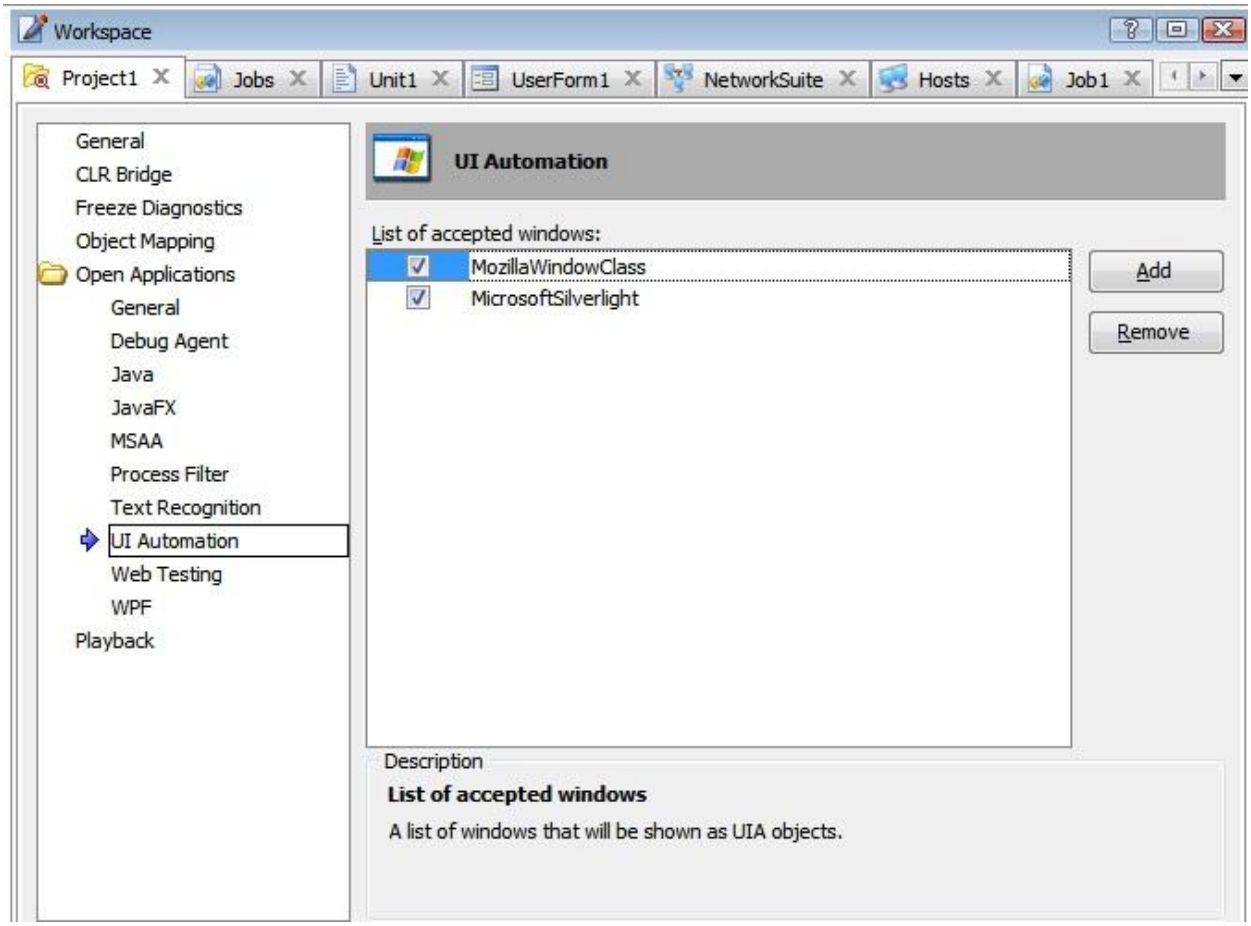
Обо всех подробностях того, как это сделать, можно прочитать в справке TestComplete-а, раздел **«Preparing Flex Applications for Testing».**

Подготовка Silverlight-приложений

Чтобы иметь доступ к внутренним элементам Silverlight-приложений, необходимо установить параметр windowless в значение false (если приложение встроено в страницу с помощью элемента object), либо параметр isWindowless в значение false (если приложение

встроено с помощью JavaScript'a). Подробнее об этих настройках можно прочитать в главе «Preparing Silverlight Applications for Testing» справочной системы TestComplete.

Кроме того, необходимо в самом TestComplete добавить класс элемента управления, в котором находится приложение, в раздел UI Accessibility (настройки проекта TestComplete). Чтобы сделать это, в Object Browser-е выберите элемент, в котором находится ваше Silverlight-приложение, скопируйте значение его свойства ClassName, затем перейдите в настройки проекта (правый щелчок мышью на имени проекта – Edit – Properties – UI Accessibility) и добавьте скопированное имя класса в список поддерживаемых элементов управления, как показано на рисунке ниже. Обычно для браузера Internet Explorer это класс *MicrosoftSilverlight*, а для браузера Mozilla Firefox – *MozillaWindowClass*.



Подробнее об этом можно прочитать в разделе справки **«Testing Silverlight Applications with TestComplete».**

5 Присоединяемые и Самотестируемые приложения

TestComplete, по сути, представляет собой OLE-сервер, с которым можно работать как из самого TestComplete, так и из других приложений. Такая возможность бывает необходима в том случае, если возможностей встроенных в TestComplete языков вам недостаточно и вы хотите использовать более сложную логику организации своих тестовых скриптов.

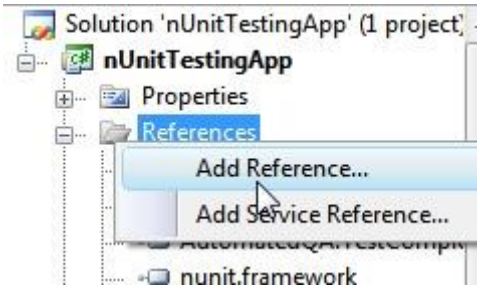
Подключив к своему приложению определенные библиотеки, входящие в состав TestComplete, вы создадите Присоединяемое приложение (Connected Application), из которого можете обращаться к любым объектам TestComplete (Sys, Log, TestedApps, Runner и т.д.).

Если при этом приложение содержит код для тестирования себя самого, такое приложение называется Самотестируемым (Self-Testing Application).

Мы рассмотрим эти обе возможности в одной главе на примере простого C# приложения nUnitTestingApp. Для более детального ознакомления с Самотестируемыми и Присоединяемыми приложениями (например, Delphi, MS Visual C++ и т.д.) вам придется обратиться к справочной системе TestComplete.

Присоединяемое приложение

Итак, прежде всего нам надо подключить необходимые библиотеки. В случае .NET-приложения это файлы *AutomatedQA.script.dll u AutomatedQA.TestComplete.CSConnectedApp.dll,* которые находятся в папке *<TestComplete>\Connected Apps\.NET*. Чтобы подключить их, необходимо выполнить для C#-проекта команду Add Reference.



Теперь в модуле программы необходимо подключить пространства имен

using AutomatedQA.script;
using AutomatedQA.TestComplete;

Обращение к объектам TestComplete из кода программы производится с помощью объекта Connect. Например, Connect.Sys.

Теперь мы можем написать простой метод, который будет отображать имя комьютера (используя объект Sys) и запускать тестовое приложение из проекта TestComplete.

private void btnConnected_Click(object sender, EventArgs e)

```
    {

        MessageBox.Show(Connect.Sys["HostName"]);

        Connect.RunTest("Test Connected App", "Project1",
@"C:\ProjectSuite1\ProjectSuite1.pjs");

        var p = Connect.TestedApps["Items"](0)["Run"]();

        Connect.StopTest();

    }
```

Посмотреть на этот пример вы можете, взяв его в архиве с примерами.

Здесь следует обратить внимание на то, что объект Sys – это единственный объект TestComplete, с которым можно работать из присоединяемых приложений, не запустив сначала тест (с помощью метода RunTest). Если попытаться это сделать, возникнет исключение NullReferenceException.

Также учтите, что объект Connect предоставляет доступ только к стандартным объектам TestComplete, но не к объектам сторонних плагинов. Чтобы обращаться к таким объектам, необходимо использовать объект Integration:

```
var MyObject = Connect.Integration["GetObjectByName"]("MyObjectName");
```

Самотестируемое приложение

Теперь сделаем из нашего Присоединенного приложения Самотестируемое. Основное отличие Самотестируемого приложения от Присоединенного в том, что тесты должны запускаться в другом потоке (Thread), иначе TestComplete не сможет работать с экранными объектами (окна, кнопки и т.п.). Поэтому сам тест (который просто нажимает на кнопку запущенного приложения) мы вынесем в отдельный класс:

```
public class SelfTests:Connect

  {

    public void PlusTest()

    {

        Connect.RunTest("Test Self-Testing App", "Project1",
@"C:\ProjectSuite1\ProjectSuite1.pjs");


Connect.Sys["Process"]("nUnitTestingApp")["WinFormsObject"]("Form1")["WinFormsObject"
]("button1")["Click"]();

        Connect.StopTest();

    }
```

```
}
```

А вот код запуска теста, который мы привязали к нажатию на кнопку:

```
private void btnSelf_Click(object sender, EventArgs e)
    {
        SelfTests slf = new SelfTests();
        ThreadStart ths = new ThreadStart(slf.PlusTest);
        Thread thrd = new Thread(ths);
        thrd.Start();
    }
```

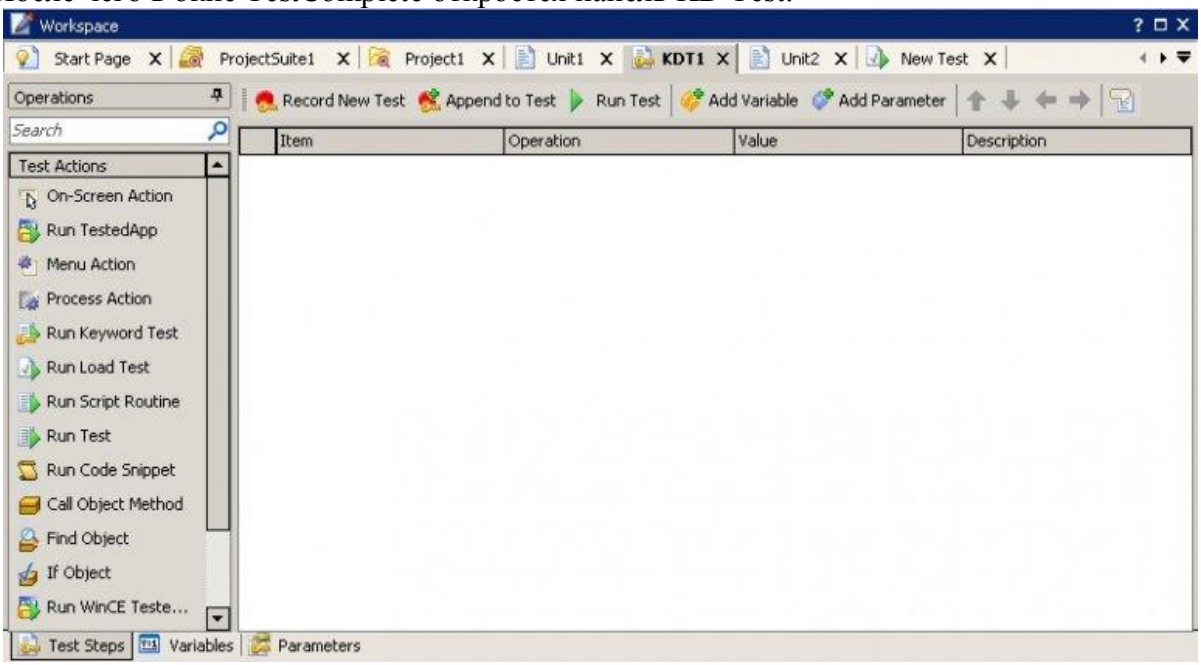
Также следует сказать, что есть два способа создания тестов для самотестирующихся приложений: написание их вручную и конвертация записанных C#Script тестов. Во втором случае придется еще немного изменить код (например, ключевое слово function заменить возвращаемым значением и т.п.).

6 Keyword Driven Testing (Тесты, управляемые ключевыми словами)

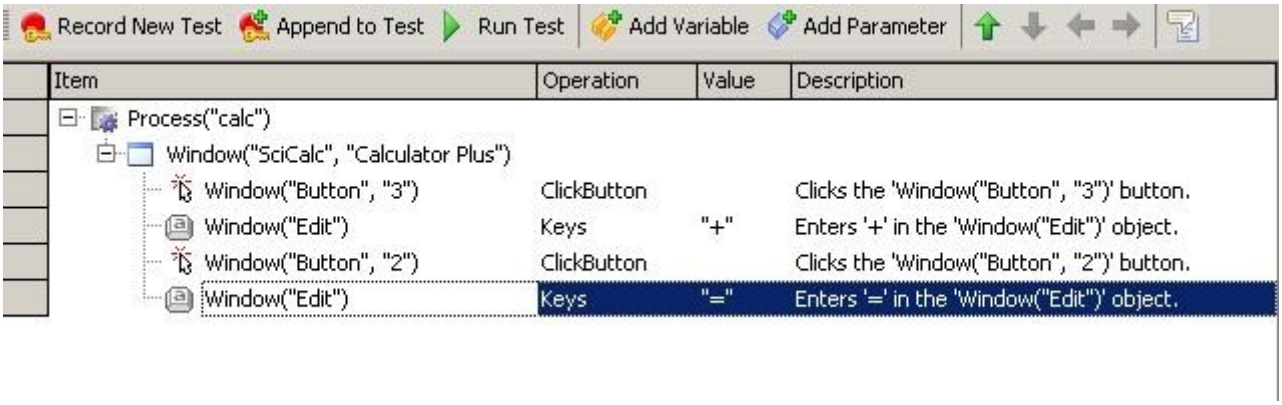
Keyword Driven Testing – это сравнительно новая возможность, появившаяся впервые в TestComplete 7. Keyword Driven Testing – это визуальное представление тестовых скриптов, когда каждому действию (щелчок мышью, нажатие клавиш, выбор элементов списка и т.п.) сопоставляются ключевые слова (keywords). Несколько ключевых слов объединяются в действия (actions). Подобная организация тестовых скриптов очень хорошо знакома пользователям QuickTest Pro, для тех же, кто ранее работал с другими инструментами, эта технология будет новой.
Хотя начиная с 7 версии TestComplete по умолчанию записывает именно Keyword Driven тесты, мы намеренно рассматривали сначала работу с обычными тестами, чтобы учебник был одинаково полезен в том числе для пользователей более ранних версий.
В документации к TestComplete вместо Keyword Driven Test используется сокращенное название Keyword Test, в нашем учебнике мы для краткости будем использовать название KD Test.

Запись KD Test-ов

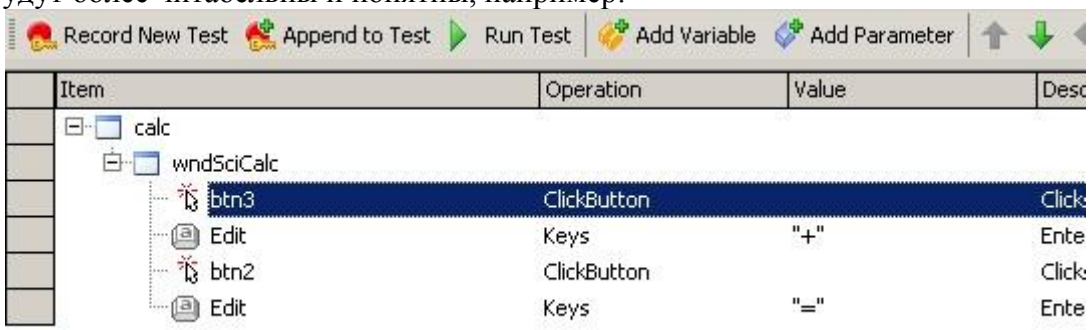
Давайте для начала создадим простой KD Test. Для этого необходимо щелкнуть правой кнопкой мыши в проекте на элементе **KeywordTests**, выбрать пункт меню *Add – New Item* и в появившемся окне Create Project Item ввести имя нового теста (например, KDT1). После чего в окне TestComplete откроется панель KD Test.



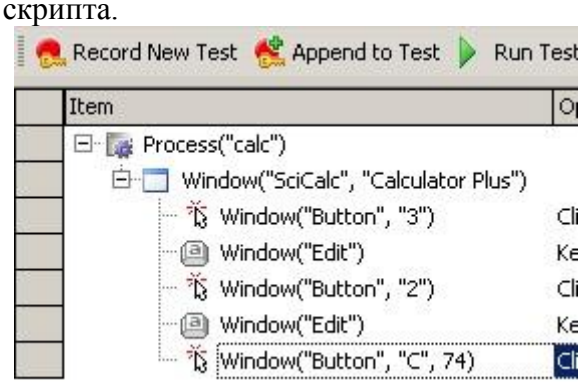
Мы создали пустой тест, который можно изменять (к этому мы вернемся немного позже). Теперь давайте запишем новый тест, используя встроенные возможности TestComplete. На панели инструментов в верхней части нажмем кнопку Record New Test и запишем какие-нибудь действия в приложении Калькулятор (например, вычислим значение выражения «3+2»), причем кнопки «3» и «2» будем нажимать мышью, а «+» и «=» - при помощи клавиатуры. В результате получим такой скрипт:



В колонке **Item** отображается имя объекта, с которым производится действие, в колонке **Operation** – производимая операция (например, ClickButton – нажатие на кнопку), в колонке **Value** – параметр операции (например, имя кнопки в нашем случае) и в колонке **Description** – описание операции.
При этом, если вы используете NameMapping в вашем проекте, значения в колонке **Item** будут более читабельны и понятны, например:



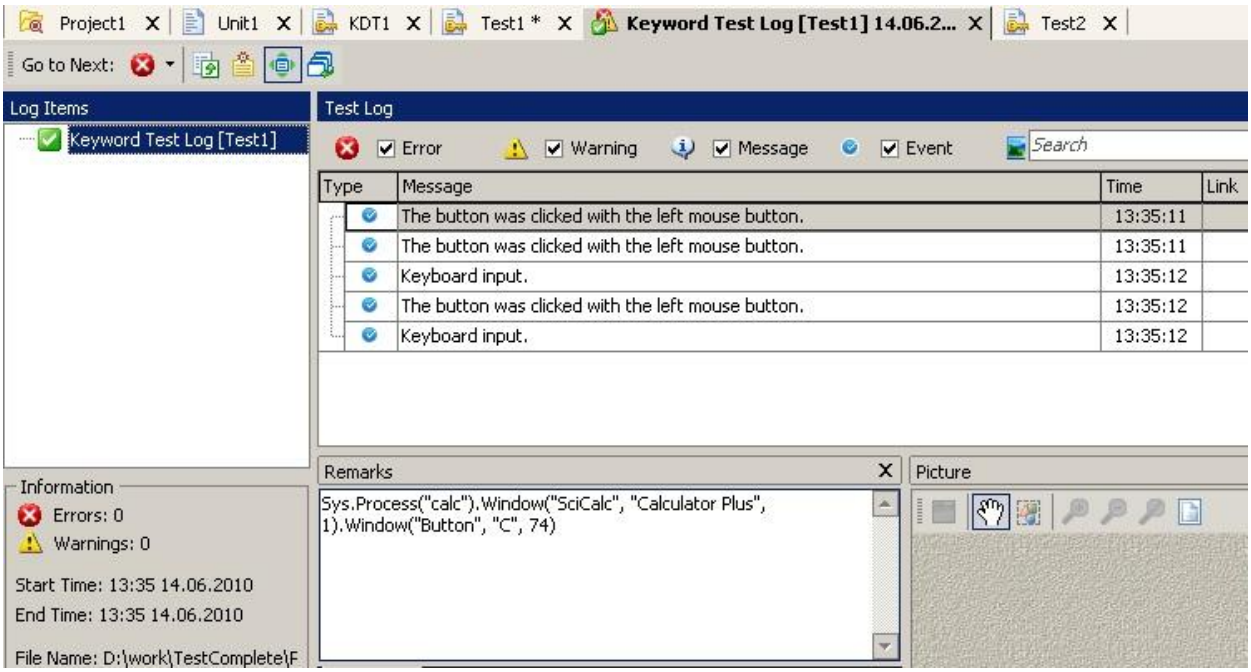
Предположим теперь, что вы хотите очищать поле результатов Калькулятора перед тем, как начать выполнять операции, для чего первым действием хотите нажимать кнопку C (Cancel). Для добавления действий в существующий тест предназначена кнопка Append to Test панели инструментов. После нажатия на эту кнопку вы опять переходите в режим записи, записываете необходимые действия и они добавляются в конец существующего скрипта.



С помощью кнопок со стрелками в правой части панели инструментов можно переместить действие нажатия на кнопку C вверх. Теперь перед началом выполнения вычислений поле результатов будет очищаться.

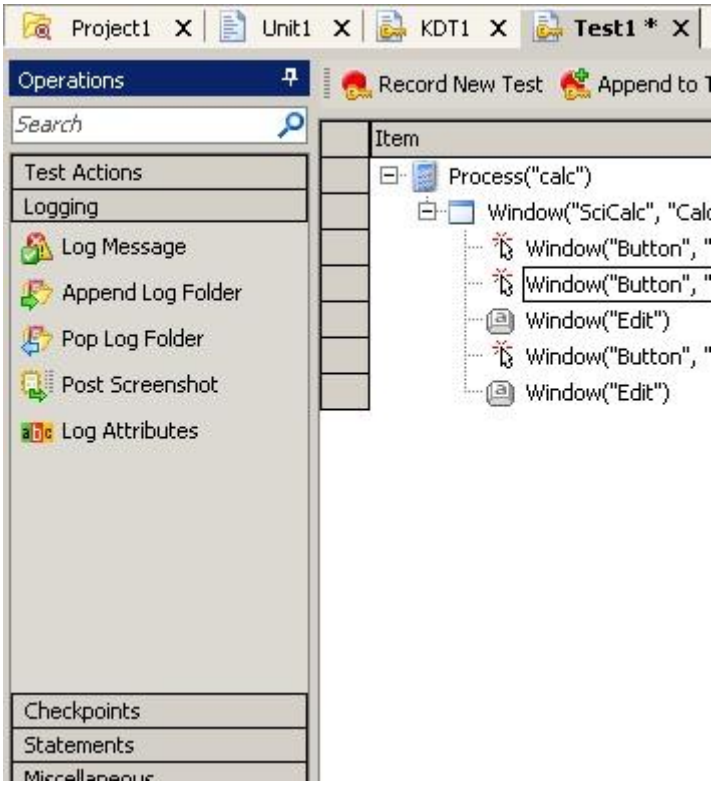
Item	Operation	Value	Description
Process("calc")			
Window("SciCalc", "Calculator Plus")			
Window("Button", "C", 74)	ClickButton		Clicks the 'Window("Button", "C"
Window("Button", "3")	ClickButton		Clicks the 'Window("Button", "3"
Window("Edit")	Keys	"+"	Enters '+' in the 'Window("Edit")
Window("Button", "2")	ClickButton		Clicks the 'Window("Button", "2"
Window("Edit")	Keys	"="	Enters '=' in the 'Window("Edit")

Теперь с помощью кнопки **Run Test** на панели инструментов запустим наш тест и убедимся, что все работает так, как задумывалось.



Модификация KD Test-ов

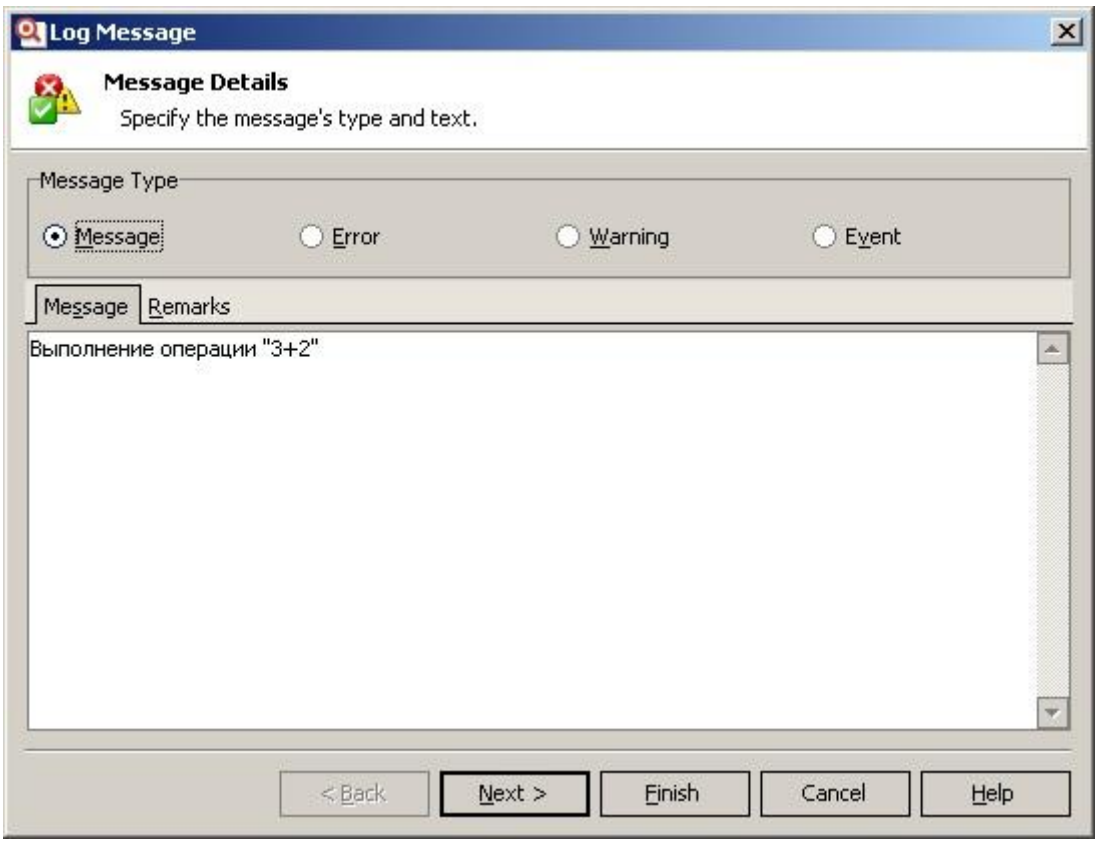
Основное отличие KD Test-ов от обычных в том, что они позволяют осуществлять те же самые операции, что и обычные скрипты, однако при этом используя визуальный редактор вместо написания кода. В KD Test-ах можно использовать циклы, поиск и ожидание объектов по различным свойствам, использовать условные выражения, блоки перехвата исключений и т.п.
Начнем с простых действий. Например, добавления в лог сообщения (Message).
В левой части панели KDT находится панель Operations, которая позволяет добавлять новые действия в KD Test-ы.



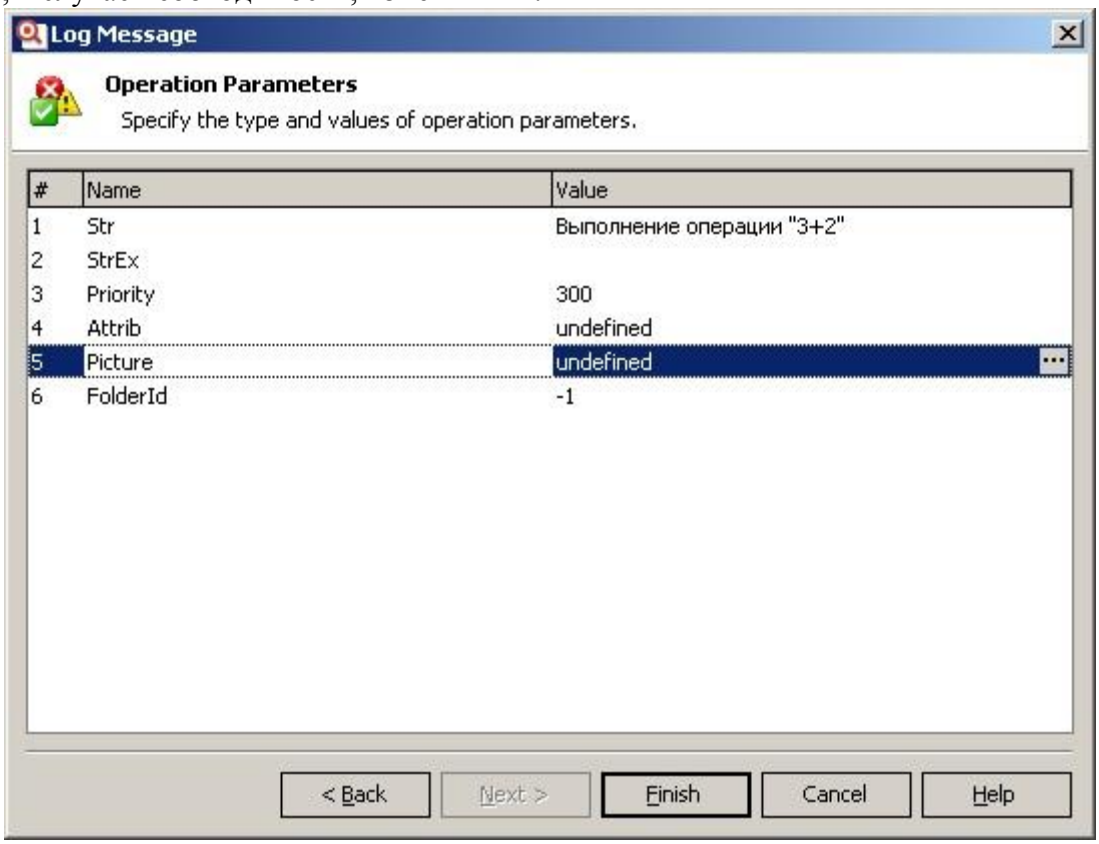
Операции разделены на логические группы:

- **Test Actions** – сюда входят операции с различными элементами управления, работа с меню, функции для поиска объектов и пр.
- **Logging** – работа с логом (добавление сообщений, ошибок, предупреждений, папок и т.п.)
- **Checkpoint** – добавление чекпоинтов (контрольных точек)
- **Statements** – добавление различных условий, циклов и т.п.
- **Miscellaneous** – прочие возможности (вставка задержки, работа с индикатором)

Для начала попробуем добавить простое действие: вывод в лог сообщения «выполнение действие 3+2». Для этого перетащим элемент Log Message с панели Operations в то место скрипта, где вы хотите выводить это сообщение (например, сразу за нажатием на кнопку C).
После этого на экране появится диалоговое окно Log Message, в котором задаются параметры команды.

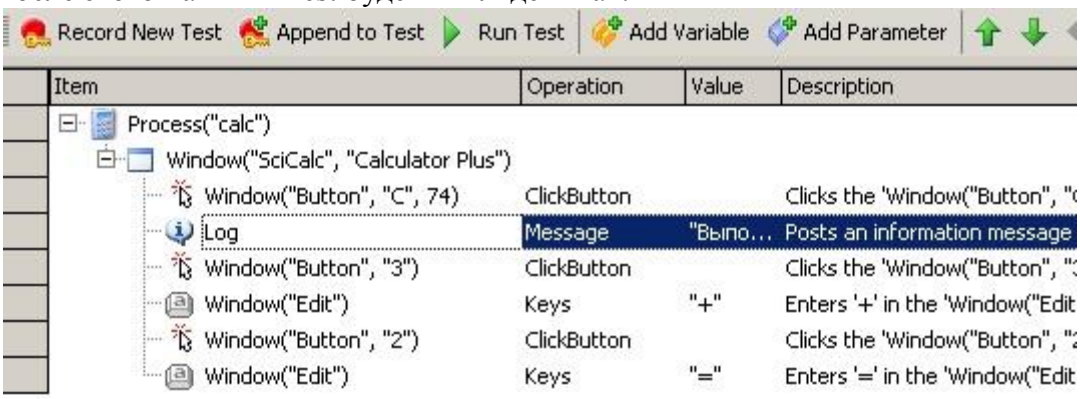


Здесь мы указываем, какой тип сообщения мы хотим использовать (Message, Error, Event, Warning), а также основной и дополнительный текст (Message и Remarks).
После этого можно нажать **Finish** и закончить операцию добавления действия, а можно нажать **Next** и перейти на вторую страницу окна, где посмотреть все параметры действия и, в случае необходимости, изменить их.



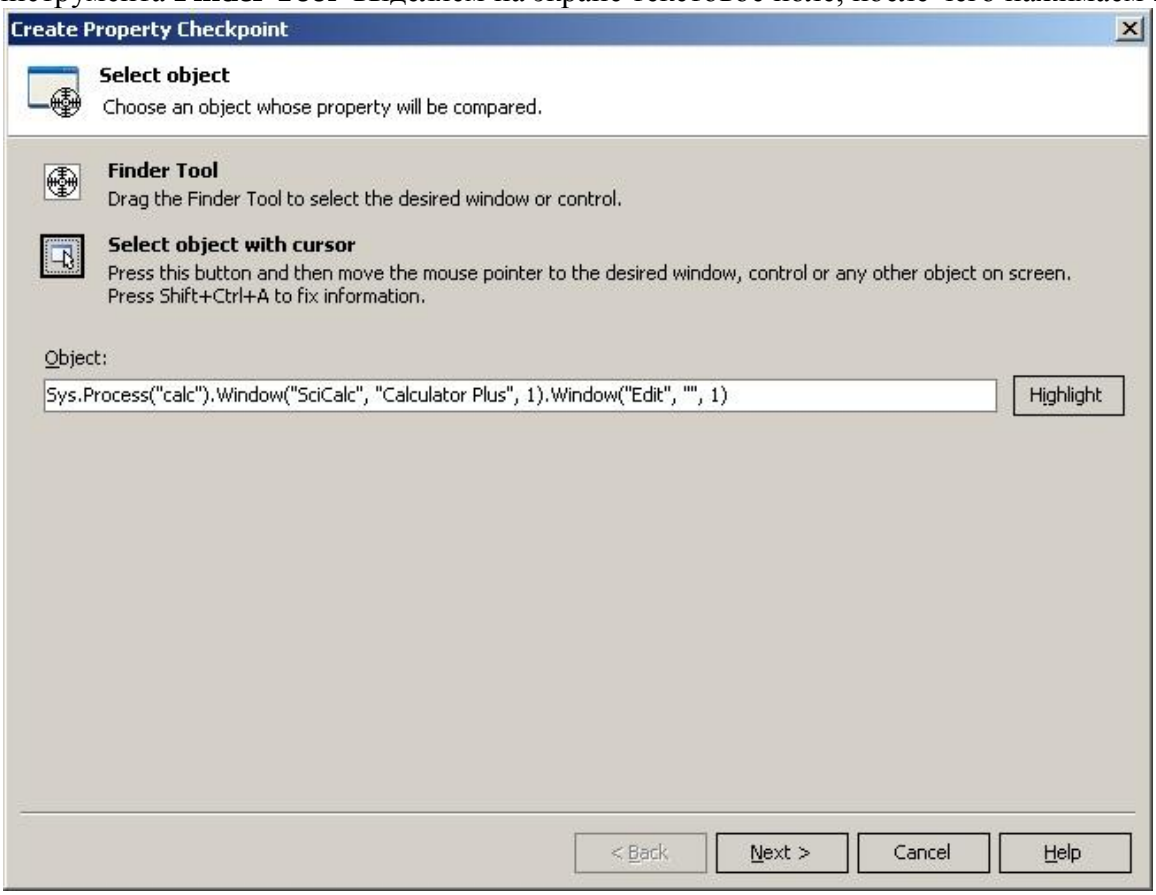
Эти параметры обсуждались в главе 3.8 Использование логов и анализ результатов, поэтому мы не будем здесь подробно на них останавливаться.

После этого наш KD Test будет выглядеть так:



Аналогичным образом с использованием мастера добавляются и другие элементы скрипта. Например, в следующем примере мы вставим проверку значения в текстовом поле Калькулятора после того, как выполним действия. Для этого мы используем чекпоинт **Property Checkpoint.**

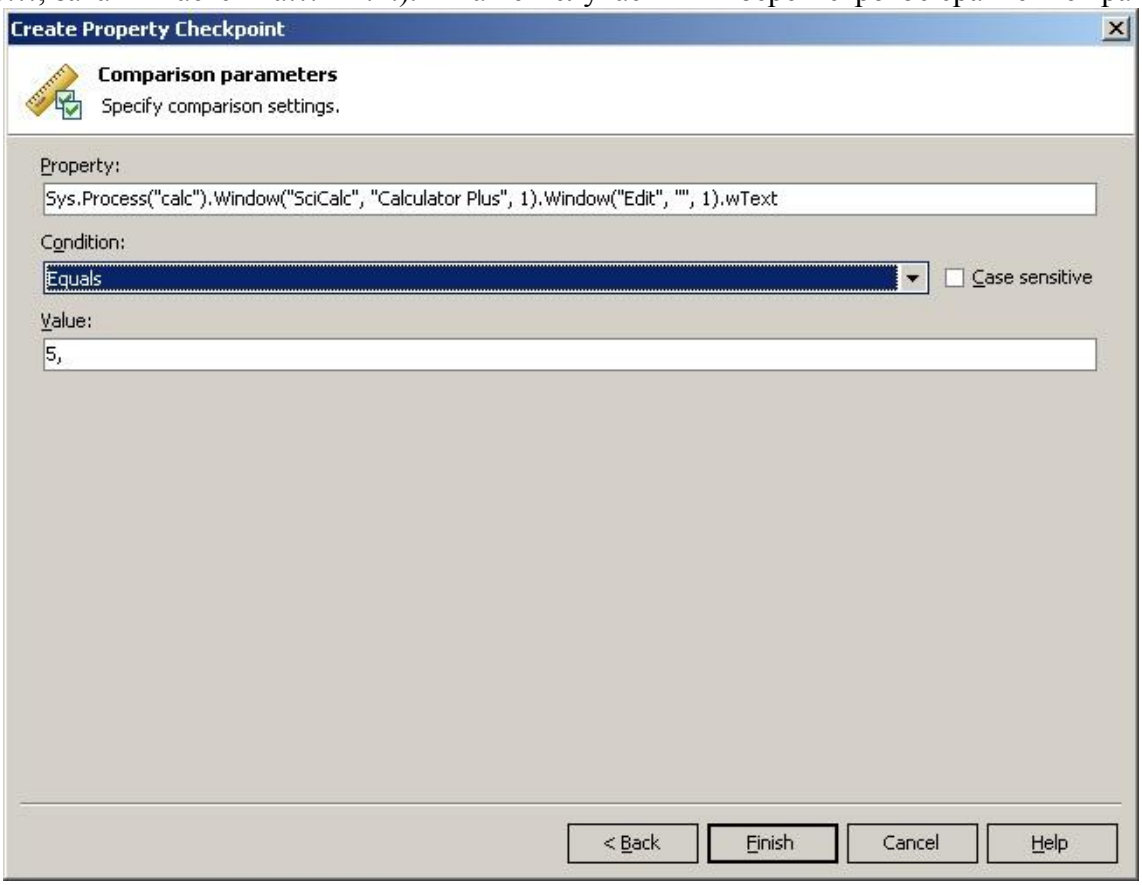
Точно так же перетаскиваем элемент из панели **Operations** в то место, где хотим выполнить проверку (в нашем случае сразу после нажатия на кнопку «=») и с помощью инструмента **Finder Tool** выделяем на экране текстовое поле, после чего нажимаем **Next**.

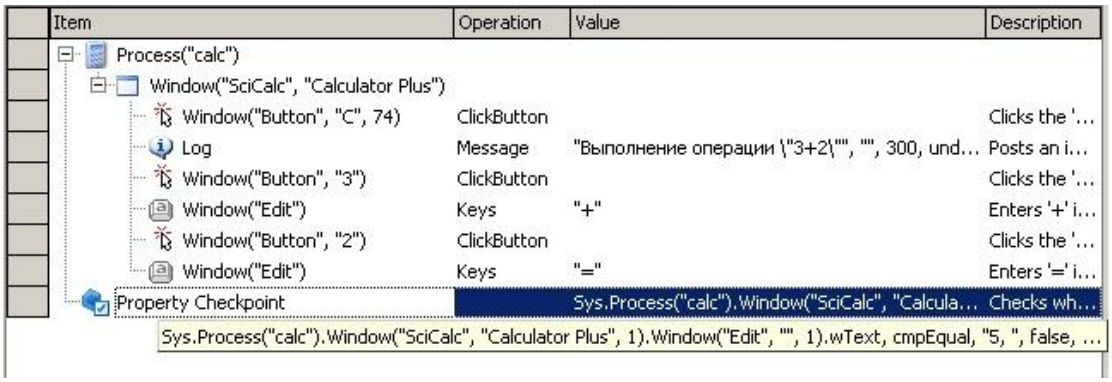


На второй страничке диалогового окна **Create Property Checkpoint** выбираем нужное свойство (в нашем случае wText) и нажимаем Next.

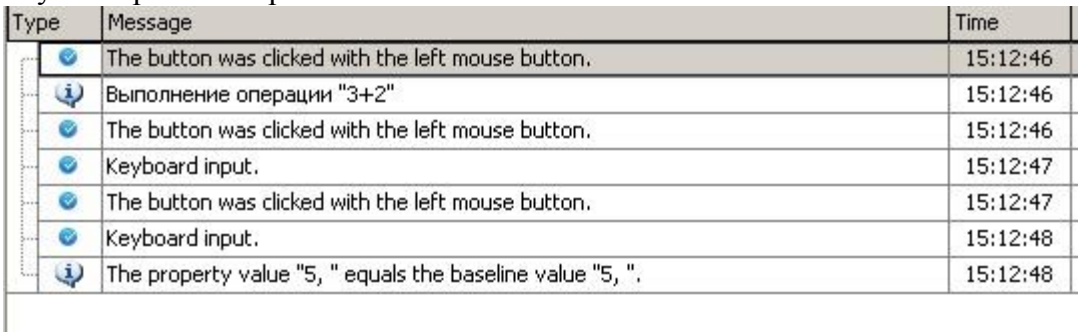
На третьей страничке выбираем одно из условий проверки (равно, неравно, начинается

с…, заканчивается на… и т.п.). В нашем случае мы выберем строгое сравнение «равно»:



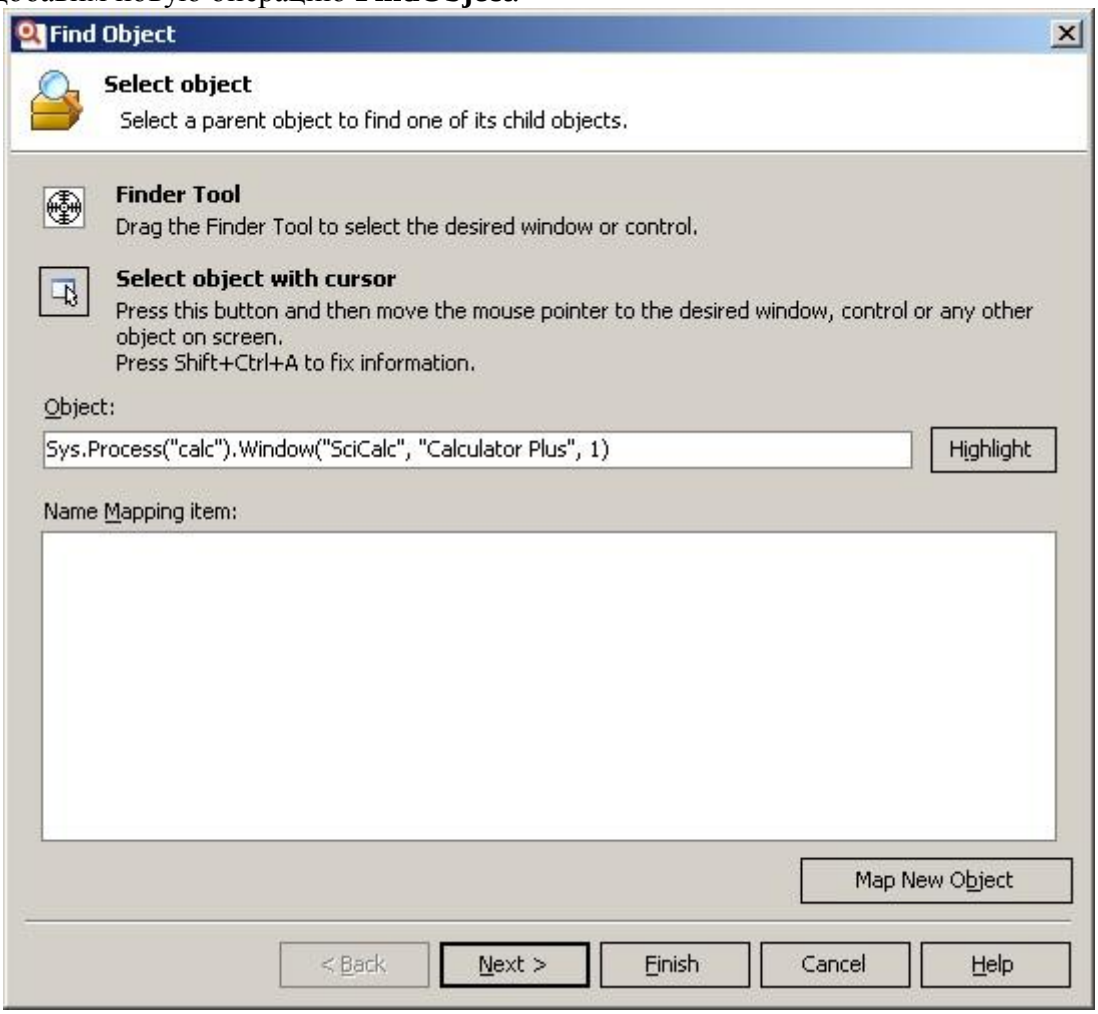


И еще раз запустим наш скрипт, чтобы убедиться, что все работает как предполагалось. Результат работы скрипта:

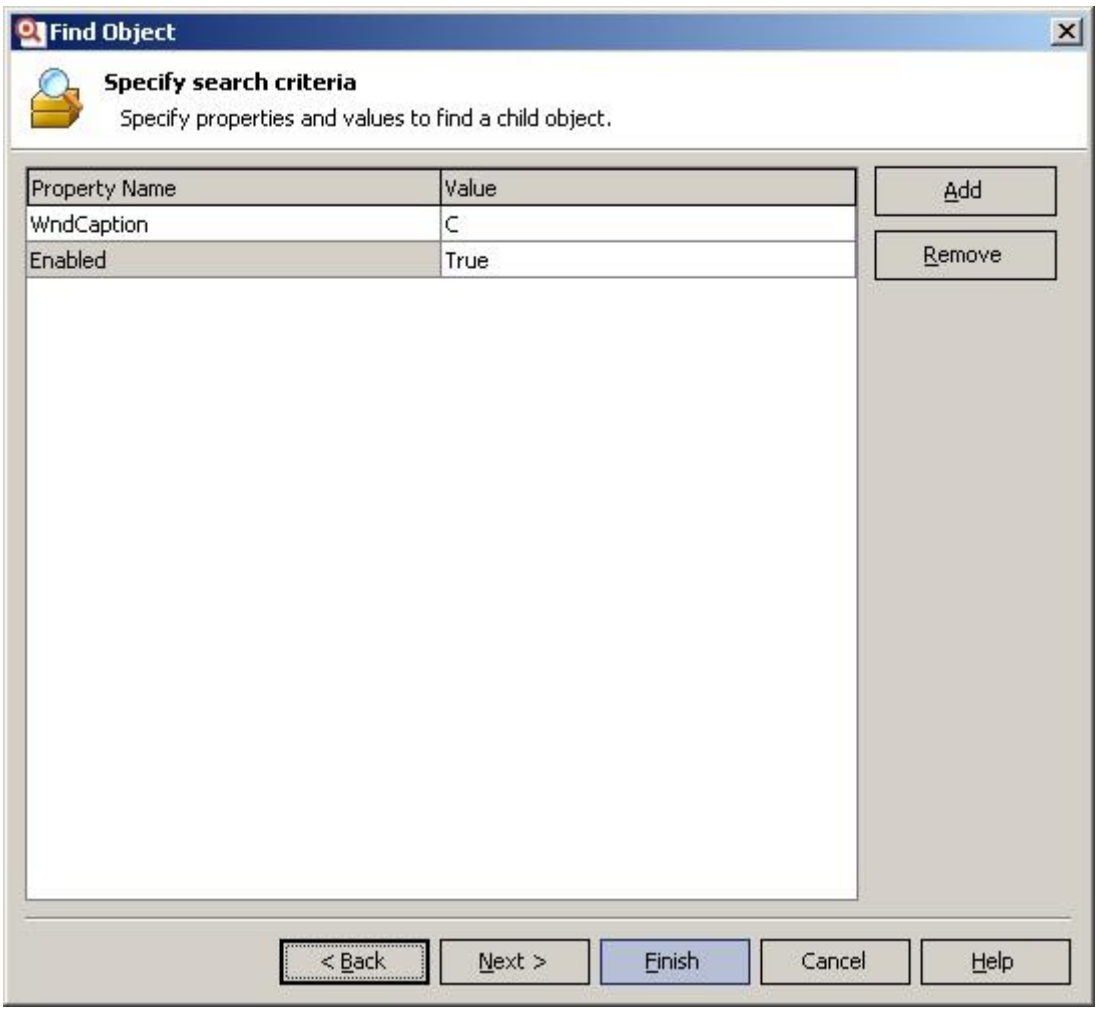


В качестве еще одного примера рассмотрим операцию **FindObject**, аналогично такой, которую мы использовали в главе 3.5 Синхронизация выполнения скриптов. Мы нажмем на кнопку C, предварительно найдя ее по двум свойствам: *WndCaption=C* и *Enabled=True*. Первое свойство характерно для двух кнопок (кнопка сброса C и кнопка C, используемая

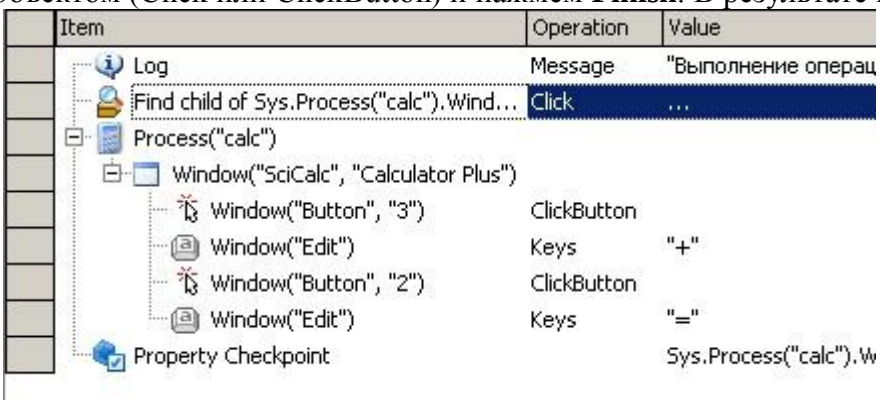
в шестнадцатеричных вычислениях), а второе свойство в нашем случае характерно только для кнопки сброса.
Для этого сначала удалим действие нажатия на кнопку C в начале скрипта, а затем добавим новую операцию **FindObject**.



С помощью инструмента **Finder Tool** выделим родительский элемент, чьим потомком является кнопка (в нашем случае это главное окно Калькулятора) и нажмем Next.
В следующем окне введем свойства и значения, по которым будем искать элемент



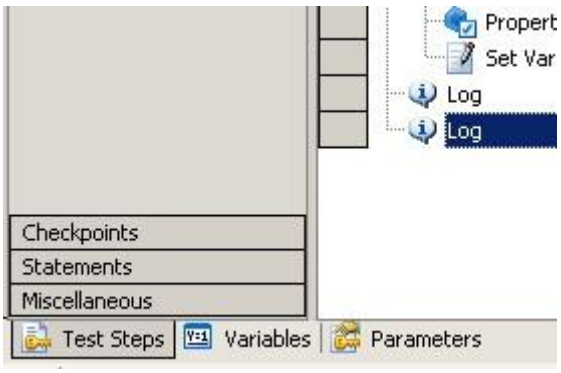
И снова нажмем **Next**. Обратите внимание, что если TestComplete не сможет на этом этапе найти подходящий элемент управления, он выдаст сообщение об ошибке.
На следующей страничке выберем действие, которое необходимо проделать с найденным объектом (Click или ClickButton) и нажмем **Finish**. В результате получим такой тест:



Теперь нам не нужно передавать индекс кнопки в качестве параметра, что улучшает структуру и читабельность тестов.

Переменные и параметры

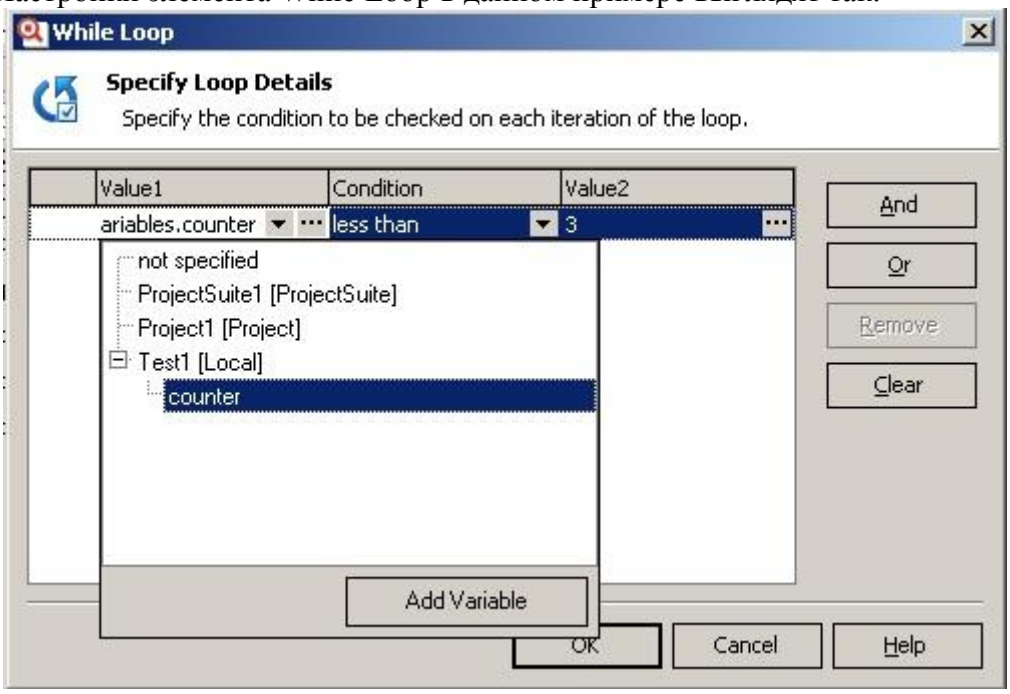
Как и в обычных тестах, в KD Test-ах могут использоваться переменные и параметры. Для них предназначены соответственно вкладки **Variables** и **Parameters** на панели Keyword Driven Test-ов.



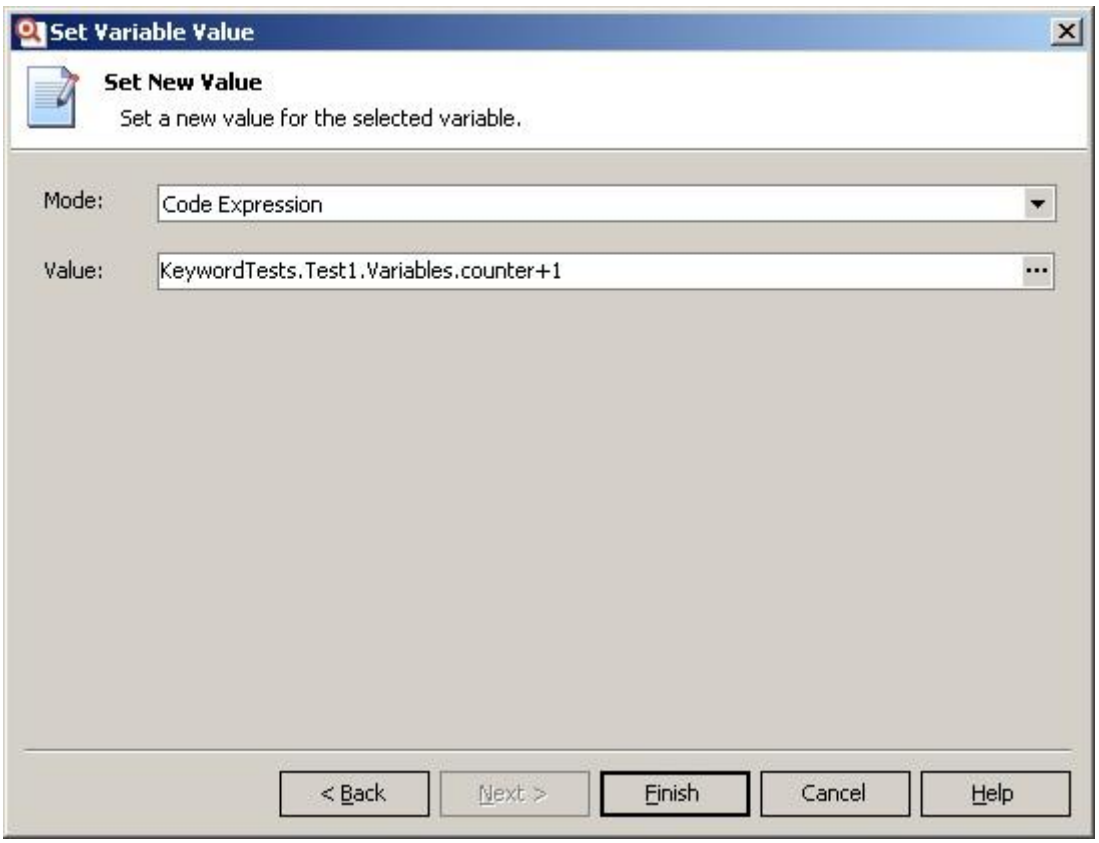
Для добавления переменной необходимо перейти на вкладку **Variables**, щелкнуть правой кнопкой мыши по списку и выбрать пункт меню *New Item*. Затем выбрать для появившейся переменной имя, тип и значение по умолчанию. Затем эту переменную можно использовать в KD Test-ах. Например, ниже показан пример цикла while, который исполняется до тех пор, пока переменная counter не достигнет значения 3. Чтобы этот цикл не был бесконечным, мы поместили в него инструкцию, увеличивающую значение переменной на единицу во время каждого прохода цикла.

Item	Operation	Value
Log	Message	"Выполнение операции \"3+2\"", "", 300...
While Loop		Variables.counter less than 3
Find child of Sys.Process("calc").....	Click	...
Process("calc")		
Window("SciCalc", "Calculator Plus")		
Window("Button", "3")	ClickButton	
Window("Edit")	Keys	"+"
Window("Button", "2")	ClickButton	
Window("Edit")	Keys	"="
Property Checkpoint		Sys.Process("calc").Window("SciCalc", "C...
Set Variable Value	counter [Local]	KeywordTests.Test1.Variables.counter+1

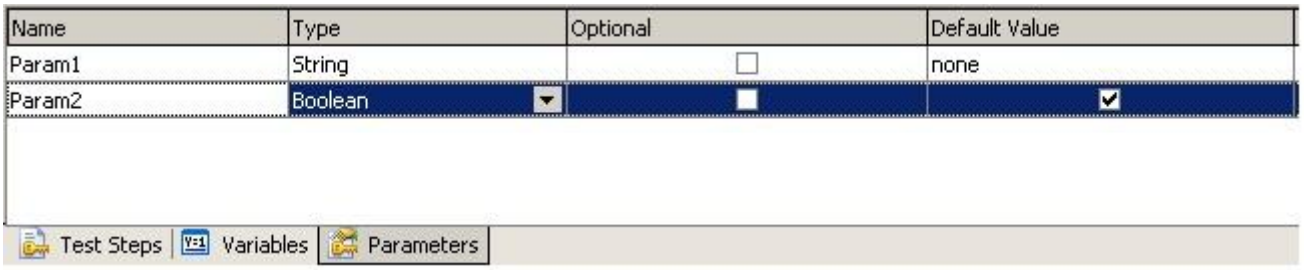
Настройки элемента While Loop в данном примере выглядят так:



Для увеличения значения переменной мы использовали операцию **Set Variable Value**, выбрав в настройках переменную counter, а затем выбрав в качестве значения элемент **Code Expression** и введя код для увеличения счетчика.



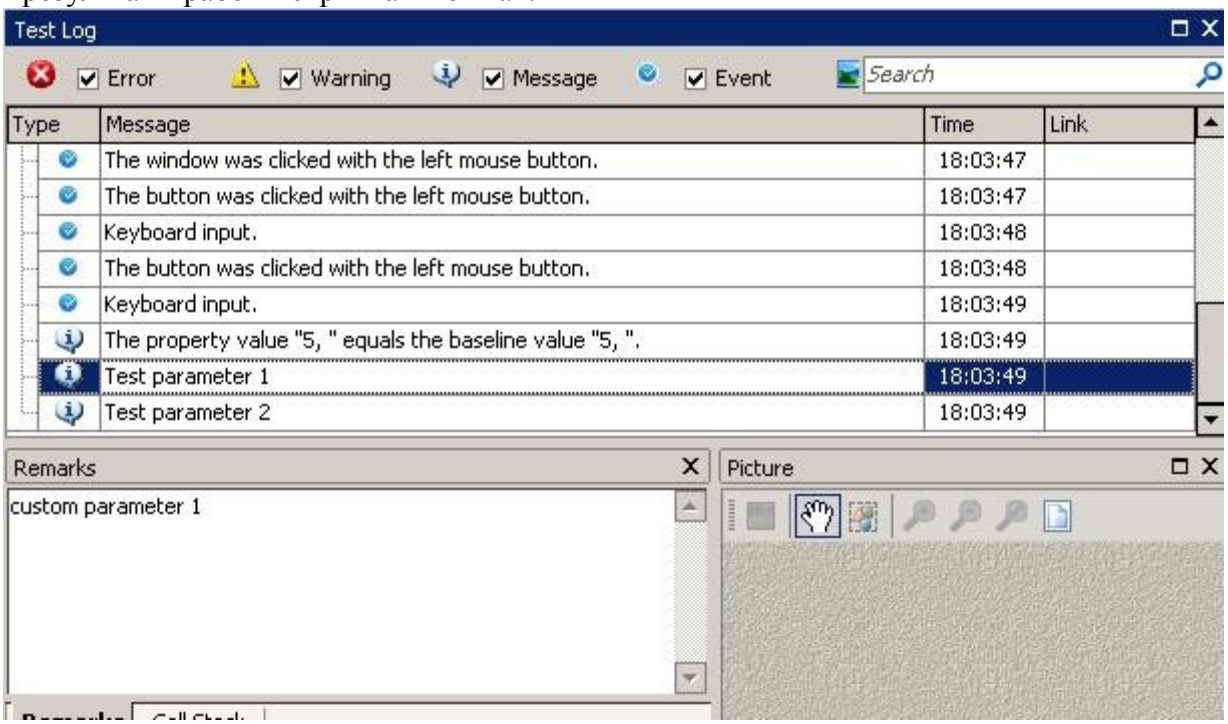
Добавление параметров делается точно так же на вкладке **Parameters**. Единственная разница – это колонка **Optional**, позволяющая задавать необязательные параметры. Например, если в тесте заданы следующие параметры:



То вызов KD Test-а из скриптов будет выглядеть так:

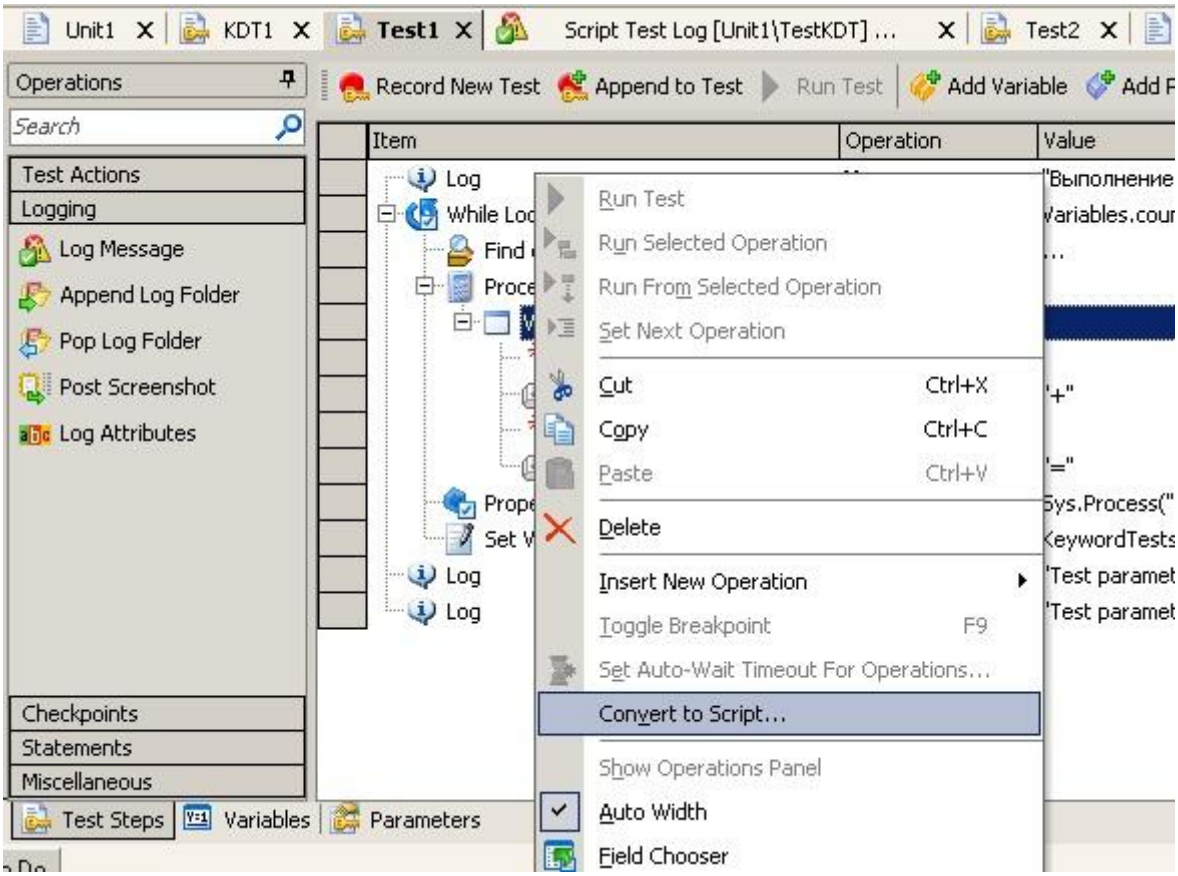
```
function TestKDT()
{
  KeywordTests.Test1.Run("custom parameter 1", false);
}
```

А результаты работы скрипта – вот так:

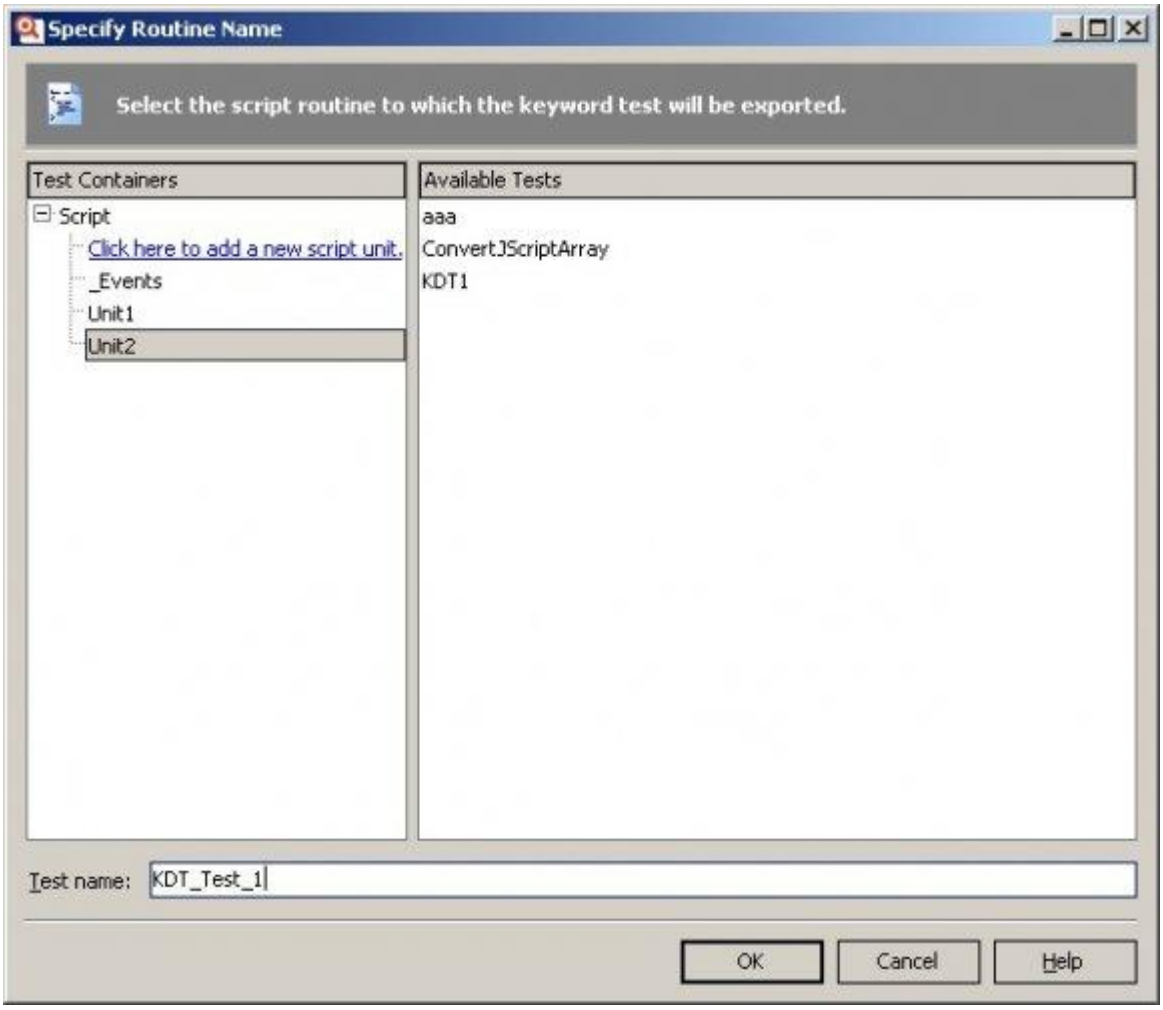


Конвертация Keyword-Driven тестов

При всем удобстве и простоте KD Test-ов, не все действия можно реализовать с их помощью. Частично это обусловлено различиями в языках программирования, а частично тем, что при помощи языков программирования можно организовать гораздо более сложные конструкции, чем при использовании Keyword-Driven тестов. Также, возможно, вы захотите изучать тот или иной поддерживаемый TestComplete-ом язык при помощи конвертации ранее записанных скриптов, тогда вам тоже понадобится возможность конвертирования KD Test-ов в обычные скрипты.
Для того, чтобы сконвертировать KD Test в обычный тест, достаточно открыть его, щелкнуть в любом месте теста правой кнопкой мыши и выбрать пункт меню *Convert to Script*



После чего в появившемся окне **Specify Routine Name** необходимо выбрать существующий (или добавить новый) модуль и ввести имя функции, в которую будет сконвертирован KD Test.



В результате получим вот такой скрипт:

```
function KDT1( Param1,  Param2)
{
  var  counter, PropNames, PropValues, ConvertedPropArray,
ConvertedValuesArray;
  counter = 0;
  Log.Message("Выполнение операции \"3+2\"", "");
  for(; counter < 3;)
  {
    PropNames = new Array("WndCaption", "Enabled");
    PropValues = new Array("C", "True");
    ConvertedPropArray = ConvertJScriptArray(PropNames);
    ConvertedValuesArray = ConvertJScriptArray(PropValues);
    Sys.Process("calc").Window("SciCalc", "Calculator Plus",
1).Find(ConvertedPropArray, ConvertedValuesArray, 1000,
true).Click();
    Sys.Process("calc").Window("SciCalc", "Calculator
Plus").Window("Button", "3").ClickButton();
    Sys.Process("calc").Window("SciCalc", "Calculator
Plus").Window("Edit").Keys("+");
    Sys.Process("calc").Window("SciCalc", "Calculator
Plus").Window("Button", "2").ClickButton();
    Sys.Process("calc").Window("SciCalc", "Calculator
Plus").Window("Edit").Keys("=");
```

```
aqObject.CompareProperty(Sys.Process("calc").Window("SciCalc",
"Calculator Plus", 1).Window("Edit", "", 1).wText, 0, "5, ",
false);
    counter = KeywordTests.Test1.Variables.counter + 1;
  }
  Log.Message("Test parameter 1", Param1);
  Log.Message("Test parameter 2", Param2);
}
```

Пожалуй, единственным недостатком такой конвертации является тот факт, что во всех случаях при доступах к элементам управления используются полные пути, без использования дополнительных переменных.

7 Data Driven Testing (Тесты, управляемые данными)

Data Driven Testing (Тесты, управляемые данными) – это такой подход к тестированию, при котором тестовые данные хранятся отдельно от скриптов, обычно в документе Excel, файле CSV или в базе данных.

Допустим, нам необходимо создать 10 разных записей с помощью тестируемого приложения, т.е. 10 раз выполнить одни и те же действия с разными данными. Мы можем просто объявить массив, в который поместить все данные, а можем поместить данные в отдельный файл. Преимущество этого подхода в том, что все данные будут храниться в одном месте и когда нам надо будет их просмотреть или отредактировать, мы сможем быстро переключаться между разными данными, а не искать их по всему проекту.

Для работы с подобными данными в TestComplete существует объект DDT, с помощью которого можно создать подключение к файлу Excel, CSV или таблице из базы данных.

Мы подробно рассмотрим лишь один из них (Excel), так как работа с остальными двумя отличается несущественно. В конце этой главы мы вкратце рассмотрим особенности работы с CSV и базами данных при помощи DDT.

Для демонстрации работы с ДДТ мы протестируем несколько базовых операций Калькулятора. Для этого мы создали файл calc.xls и поместили его в папку проекта TestComplete (Stores\Files). В этом файле один лист с такими данными:

	A	B	C	D	E
1	id	digit1	operation	digit2	result
2	1	12	+	213	225
3	2	44	-	23	21
4	3	6565	**	2	13130
5	4	0	/	234	0
6	5	5	x^y	3	1255
7	6	321	-	144	177
8	7	77	+	386	463
9	8	20	**	12	240
10	9	4	n!		24
11	10	32	/	4	8
12					

Первая колонка (id) – это уникальный идентификатор строки (в принципе она необязательна). В колонках digit1 и digit2 находятся числа, с которыми мы будем производить операции. Операции указаны в столбце operation. В колонке result находится заранее вычисленный результат, с которым мы будем сравнивать результат в Калькуляторе.

Теперь рассмотрим процесс вычитки данных из файла с помощью ДДТ. Сначала необходимо создать подключение к файлу:

```
var ddtExcel = DDT.ExcelDriver(Project.Path +
"\\Stores\\Files\\calc.xls", "Table1", true);
```

В параметрах мы передаем полное имя файла, имя листа в файле и третий параметр UseACEDriver. Если параметр UseACEDriver=true, это дает возможность работать с файлами, созданными как в MS Office 2007, так и в более ранних версиях Office. Если же этот параметр равен false, то для доступа к данным будет использован драйвер ODBC, который не поддерживает работу с файлами Office 2007.

Теперь переменная ddtExcel содержит всю информацию из листа Table1, а указатель стоит на первой строке. Нам остается лишь считать данные из этого объекта.

Работа с данными происходит так: указатель находится на первой строке и мы имеем возможность с помощью свойств и методов объекта DDTDriver считать данные из любой колонки первой строки.

Затем мы перемещаем указатель на следующую строку и так же считываем данные из нее. Таким образом мы двигаемся по строкам, пока не достигнем конца файла. Первая строка файла содержит имена колонок, поэтому первая строка DDTDriver-а соответствует второй строке в файле.

Для доступа к данным в строке используется свойство Value. Это свойство принимает один параметр – имя колонки, из которой необходимо считать значение. Следующий пример считывает значение из первой строки данных, колонка digit1.

```
function TestDDT()
{
  var ddtExcel = DDT.ExcelDriver(Project.Path + "\\Stores\\Files\\calc.xls",
"Table1", true);
  Log.Message(ddtExcel.Value("digit1"));
  DDT.CloseDriver(ddtExcel.Name);
}
```

Обратите внимание на последнюю строку кода – она закрывает ранее открытый драйвер. Это необходимо делать всегда, так как количество открытых драйверов ДДТ ограничено.

Этот пример выведет нам в лог число 12 – именно такое число у нас находится в ячейке B2. Таким же образом можно узнать количество колонок и имя любой колонки по ее номеру. Например:

```
function TestDDT()
{
  var ddtExcel = DDT.ExcelDriver(Project.Path + "\\Stores\\Files\\calc.xls",
"Table1", true);
  Log.Message(ddtExcel.ColumnCount);
  Log.Message(ddtExcel.ColumnName(0));
  DDT.CloseDriver(ddtExcel.Name);
}
```

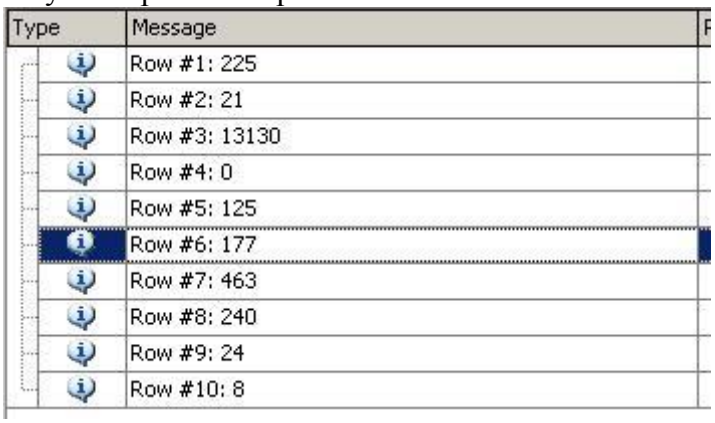
Этот скрипт выведет в лог цифру 5 (количество колонок) и слово id – имя первой колонки. Обратите внимание, что нумерация колонок начинается с нуля.

Метод Next позволяет переместиться на следующую строку, а метод EOF позволяет определить, достигнут ли конец файла. В качестве примера считаем все значения из столбца result.

```
function TestDDT()
{
```

```
  var ddtExcel = DDT.ExcelDriver(Project.Path + "\\Stores\\Files\\calc.xls",
"Table1", true);
  while(!ddtExcel.EOF())
  {
    Log.Message("Row #" + ddtExcel.Value("id") + ": " +
ddtExcel.Value("result"));
    ddtExcel.Next();
  }
}
```

Результат работы скрипта:



Type	Message
	Row #1: 225
	Row #2: 21
	Row #3: 13130
	Row #4: 0
	Row #5: 125
	Row #6: 177
	Row #7: 463
	Row #8: 240
	Row #9: 24
	Row #10: 8

Теперь напишем скрипт, который будет считывать данные, выполнять необходимые действия и затем выполнять проверку полученного значения. Если результат в Калькуляторе отличается от ожидаемого результата, будет выведено сообщение об ошибке. Специально для того, чтобы продемонстрировать выведение ошибки, мы ввели один неправильный ответ (в бой строке результат должен быть 125, а не 1255).

```
function TestDDT()
{
  var ddtExcel = DDT.ExcelDriver(Project.Path + "\\Stores\\Files\\calc.xls",
"Table1", true);
  var sKey, sRes, sExpRes;
  var wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");

  // очищаем предыдущие вычисления
  wnd.Activate();
  wnd.Keys("[Esc]");

  while(!ddtExcel.EOF())
  {
    sKey = ddtExcel.Value("digit1");
    wnd.Keys(sKey);

    sKey = ddtExcel.Value("operation");
    wnd.Window("Button", sKey).Click();

    sKey = ddtExcel.Value("digit2");
    // если второе число не указано, то вводить его не надо
    // и кнопку = наживать тоже не надо, иначе действие выполнится дважды
    if(sKey != null)
    {
      wnd.Keys(sKey);
      wnd.Window("Button", "=").Click();
    }

    // сравнение ожидаемого и полученного результатов
    sExpRes = ddtExcel.Value("result");
```

```
    sRes = wnd.Window("Edit", "", 1).wText.split(".")[0];

    if(sExpRes != sRes)
    {
      Log.Error("Wrong result, see remarks", "Expected: " + sExpRes +
"\nActual: " + sRes);
    }
    ddtExcel.Next();
  }
  DDT.CloseDriver(ddtExcel.Name);
}
```

Здесь мы показали, как можно пройтись по всем записям в DDT таблице с помощью цикла while и метода EOF. Однако в TestComplete есть более удобное средство для этого: метод DriveMethod. С его помощью можно избавиться от цикла и проверки конца файла. Для этого необходимо в метод DriveMethod в качестве параметра передать имя функции, которую необходимо выполнить для каждой строки. Тогда функция считывания данных и выполнения операций в Калькуляторе будет выглядеть так:

```
function TestDDT2()
{
  var sKey, sRes, sExpRes;
  var wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");

  // очищаем предыдущие вычисления
  wnd.Activate();
  wnd.Keys("[Esc]");

  sKey = DDT.CurrentDriver.Value("digit1");
  wnd.Keys(sKey);

  sKey = DDT.CurrentDriver.Value("operation");
  wnd.Window("Button", sKey).Click();

  sKey = DDT.CurrentDriver.Value("digit2");
  // если второе число не указано, то вводить его не надо
  // и кнопку = наживать тоже не надо, иначе действие выполнится дважды
  if(sKey != null)
  {
    wnd.Keys(sKey);
    wnd.Window("Button", "=").Click();
  }

  // сравнение ожидаемого и полученного результатов
  sExpRes = DDT.CurrentDriver.Value("result");
  sRes = wnd.Window("Edit", "", 1).wText.split(".")[0];

  if(sExpRes != sRes)
  {
    Log.Error("Wrong result, see remarks", "Expected: " + sExpRes +
"\nActual: " + sRes);
  }

}
```

А сам вызов этой функции будет выглядеть таким образом:

```
function TestDriveMethod()
{
  var ddtExcel = DDT.ExcelDriver(Project.Path + "\\Stores\\Files\\calc.xls",
"Table1", true);
```

```
  ddtExcel.DriveMethod("Unit1.TestDDT2");
  DDT.CloseDriver(ddtExcel.Name);
}
```

Обратите внимание, что имя функции, которое передается в метод DriveMethod должно иметь полный вид (имя_модуля.имя_функции), а для доступа к текущему драйверу DDT в функции TestDDT2 используется свойство CurrentDriver.

Работа с CSV файлами ничем не отличается от работы с файлами Excel, с той лишь разницей, что при создании подключения не нужно передавать имя таблицы (так как файл один и таблиц там нет в принципе: CSV имеет обычный текстовый формат, данные в нем отделены запятыми).

При работе с ADO драйвером вместо имени файла передается строка подключения (connection string) и имя таблицы. Если вы хотите подключиться через ADO драйвер к локальному файлу, то имя файла будет задано в строке подключения вместе с другими параметрами.

Например, в следующем примере мы подключимся к файлу Excel с помощью ADO. Для этого мы внесем изменения в использованную выше функцию TestDriveMethod, при этом никаких изменений в функции TestDDT2 делать не нужно!

```
function TestDriveMethod()
{
  var sConnStr = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\\calc.xls;Extended Properties='Excel 8.0;HDR=Yes;IMEX=1';";
  var ddtExcel = DDT.ADODriver(sConnStr, "[Table1$]");
  ddtExcel.DriveMethod("Unit1.TestDDT2");
  DDT.CloseDriver(ddtExcel.Name);
}
```

Примеры строк подключения к разным базам данных можно найти на сайте http://connectionstrings.com/ , откуда мы и взяли пример строки подключения к Excel файлу. Обратите внимание на символ $ в конце имени таблицы: без него при попытке подключения произойдет ошибка!

Еще раз напоминаем, что каждый открытый драйвер необходимо закрывать с помощью метода CloseDriver, так как одновременно не может быть открыто более 64 драйверов. При попытке открыть 65й драйвер TestComplete выдаст ошибку.

И еще один момент, связанный с файлами Excel. Для того, чтобы работать с файлами Excel через ДДТ драйвер, необходимо чтобы структура листа Excel была такой же, как у базы данных (т.е. данные должны располагаться строго по строкам и столбцам). Если в файле из нашего примера ввести какой-то текст в ячейку G20, то считать значение из нее с помощью ДДТ будет невозможно!

Для того чтобы узнать, как работать с excel-файлами с произвольно расположенными данными, обратитесь к главе 14.3 Работа с MS Excel через COM.

При использовании драйвера ДДТ вы можете только считывать данные из файлов, но не записывать их туда.

8 Работа с базами данных

В TestComplete существует 4 способа работы с базами данных:

- С помощью функциональности ADO DB, включенной в Windows
- С помощью объекта ADO
- С помощью объекта BDE
- При подключении ActiveX-элементов

Обычно сложности возникают при подключении к базе данных, так как в любом случае необходимо правильно сформировать строку подключения (Connection String). Если вы работаете с базой данных тестируемого приложения, то правильную строку подключения лучше узнать у разработчиков. В противном случае можно воспользоваться ресурсом http://connectionstrings.com/ , где можно найти много различных примеров строк подключения.

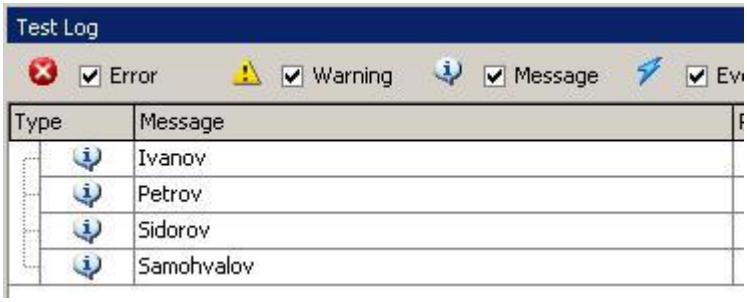
ADO DB

Подключение через ADO DB – это самый простой способ работы с базами данных, который работает всегда, так как эта возможность встроена в Windows.

В качестве примера приведем функцию, которая подключается к файлу базы данных MS Access, выбирает данные из таблицы и выводит результат в лог TestComplete-а.

```
function TestADODB()
{
  // подключаемся к базе
  var mydb = Sys.OleObject("ADODB.Connection");
  //var mydb = new ActiveXObject("ADODB.Connection");

  mydb.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" +
Project.Path + "\\Stores\\Files\\tctutorial.mdb";
  mydb.Open();

  // выбираем данные
  var rs = mydb.Execute("SELECT * FROM tblData");
  rs.MoveFirst();

  while(!rs.EOF)
  {
    Log.Message(rs.Fields("fname").Value);
    rs.MoveNext();
  }

  // закрываем подключения
  rs.Close();
  mydb.Close();
}
```

Результат работы функции:



Подробно свойства и методы объектов OLE DB распианы на сайте MSDN.

Объект ADO

ADO – это программный объект, дающий широкие возможности по работе с базами данных.

Для того чтобы иметь возможность работы с ним, необходимо установить Microsoft Data Access Components версии 2.1 или выше.

У объекта ADO есть несколько методов, позволяющих по-разному работать с базой данных:

- CreateADOCommand, CreateADOConnection – создание подключений к базе данных, позволяющих выполнять запросы и обрабатывать результаты, полученные в результате отработки запросов
- CreateADODataset, CreateADOTable, CreateADOQuery – позволяют работать с таблицами, наборами данных и запросами
- CreateADOStoredProc –позволяет работать с хранимыми процедурами

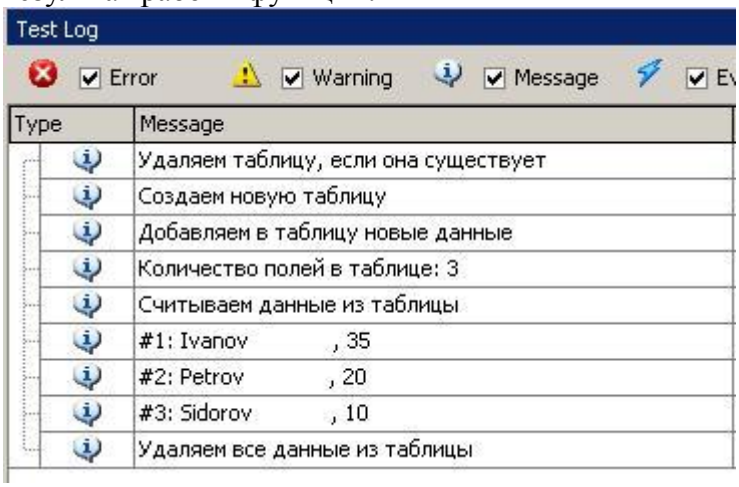
Кроме этих методов (специфичных для Borland) у метода ADO есть еще 3 метода (CreateCommand, CreateConnection и CreateRecordset), которые позволяют работать с базой аналогично рассмотренному выше ADO DB.

Рассмотрим пример использования ADO. В этом примере мы сначала попытаемся удалить таблицу (используя блок try...catch на тот случай, если таблицы в базе нет и возникнет исключительная ситуация, затем создадим новую таблицу, внесем в нее данные и затем вычитаем только что записанные данные).

```
function TestADO()
{
  var cmd = ADO.CreateADOCommand();
  var e, rs, prm;
  Log.Message("Удаляем таблицу, если она существует");
  cmd.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" +
Project.Path + "\\Stores\\Files\\tctutorial.mdb";
  cmd.CommandType = cmdText;

  cmd.CommandText = "DROP TABLE [tblTest]";
  try
  {
    cmd.Execute();
  }
  catch(e)
```

```
  {
    Log.Message("Таблица не существует");
  };

  Log.Message("Создаем новую таблицу");
  cmd.CommandText = "CREATE TABLE [tblTest](id integer PRIMARY KEY, fname
char(20), age integer)";
  cmd.Execute();

  Log.Message("Добавляем в таблицу новые данные");
  cmd.CommandText = "INSERT INTO [tblTest](id, fname, age) VALUES(1,
'Ivanov', 35)";
  cmd.Execute();

  cmd.CommandText = "INSERT INTO [tblTest](id, fname, age) VALUES(2,
'Petrov', 20)";
  cmd.Execute();

  cmd.CommandText = "INSERT INTO [tblTest](id, fname, age) VALUES(3,
'Sidorov', 10)";
  cmd.Execute();

  cmd.CommandText = "SELECT * FROM tblTest";
  rs = cmd.Execute();

  Log.Message("Количество полей в таблице: " + rs.Fields.Count);
  Log.Message("Считываем данные из таблицы");
  rs.MoveFirst();
  while (!rs.EOF)
  {
    Log.Message("#" + rs.Fields("id").Value + ": " + rs.Fields("fname").Value
+ ", " + rs.Fields("age").Value);
    rs.MoveNext();
  };

  Log.Message("Удаляем все данные из таблицы");
  cmd.CommandText = "DELETE FROM tblTest";
}
```

Результат работы функции:

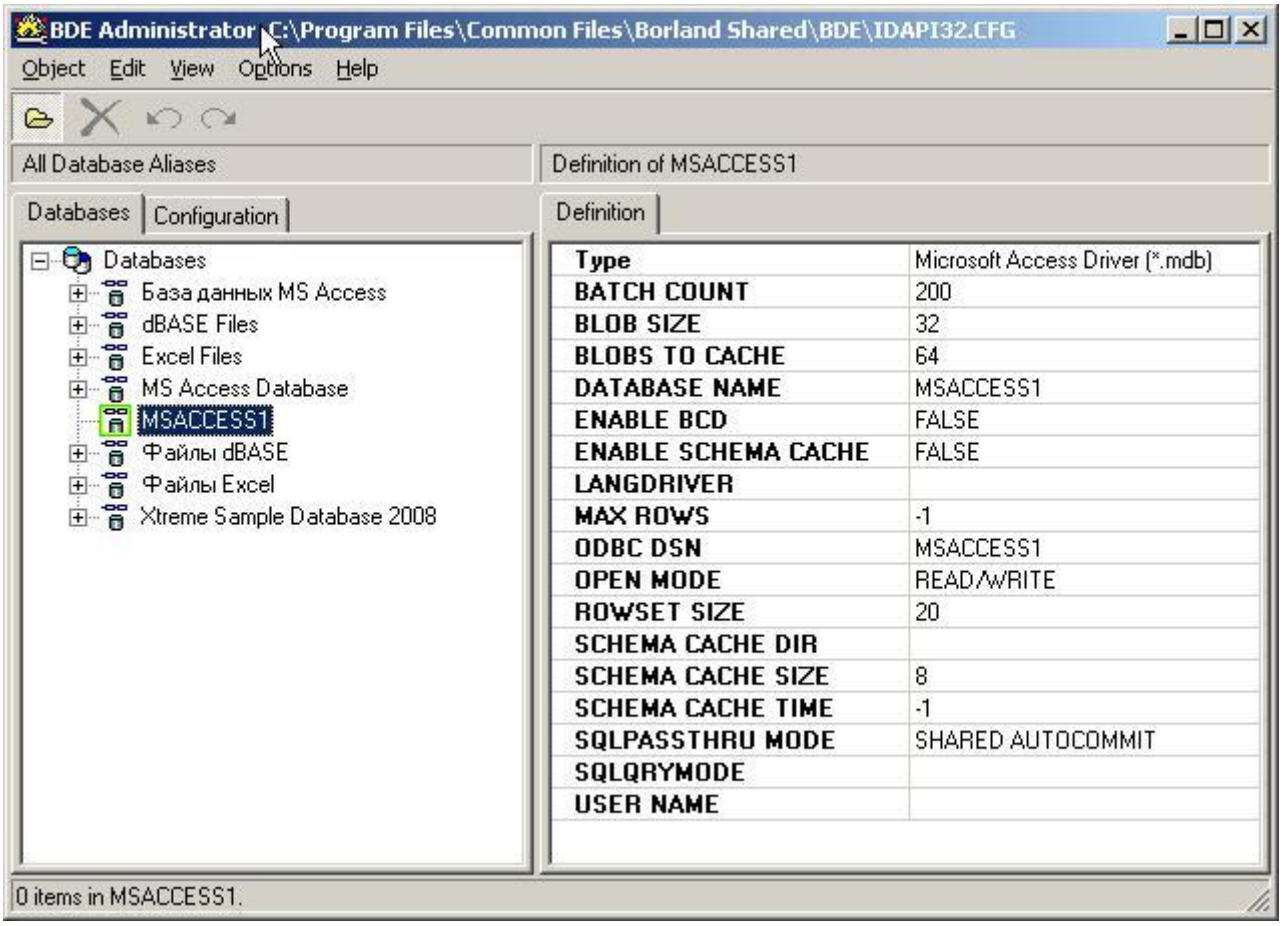


BDE

BDE (Borland Database Engine) – это устаревшая технология работы с базами данных, которая уже не поддерживается. Если же вам по какой-то причине необходимо работать с BDE, то вам придется:

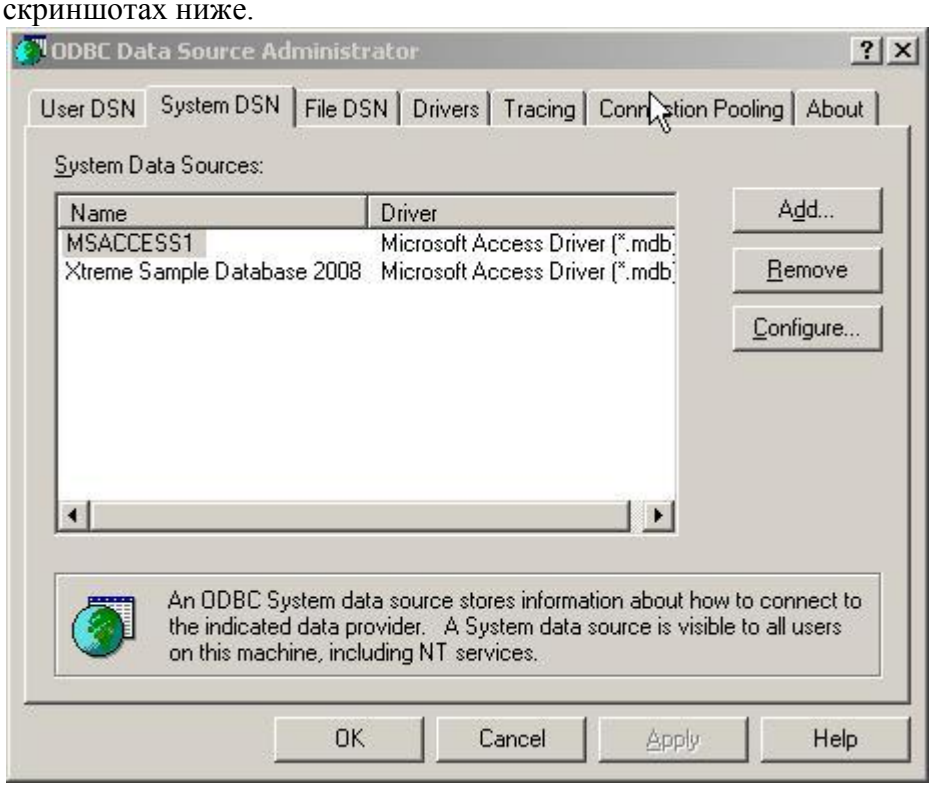
1. Скачать и установить BDE Administrator
2. Настроить BDE Administrator для работы с вашей базой данных (ниже приведены скриншоты настроек для базы данных tctutorial.mdb)

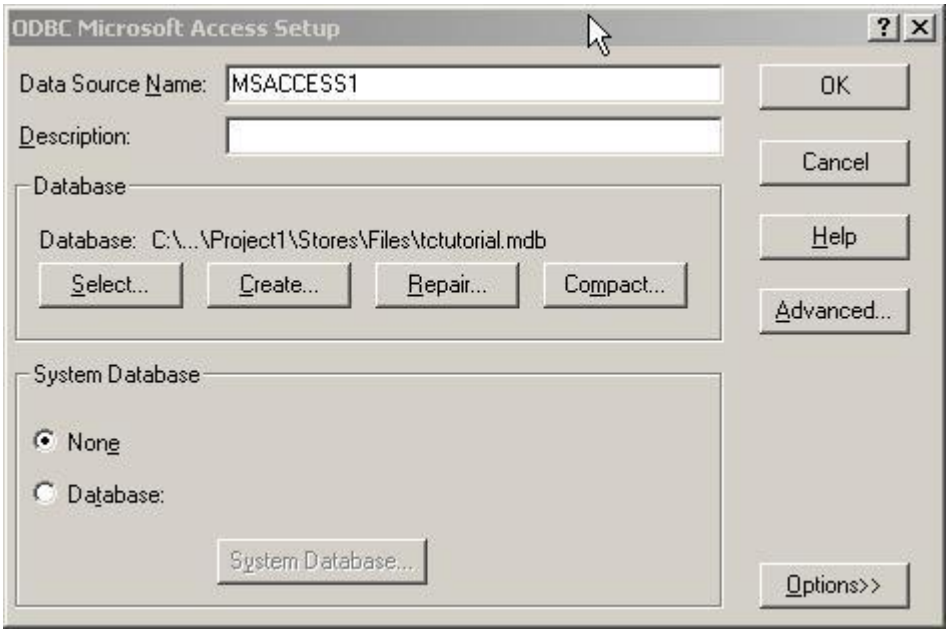
В окне BDE Administrator выберите пункт меню *Object – New* и создайте новый псевдоним (Alias), указав драйвер Microsoft Access Driver, затем укажите его имя (мы назвали его MSACCESS1).



Щелкните правой кнопкой мыши на созданном псевдониме, выберите пункт меню *ODBC Administrator*, затем *System DSN* и добавьте файл tctutorial.mdb как показано на

скриншотах ниже.



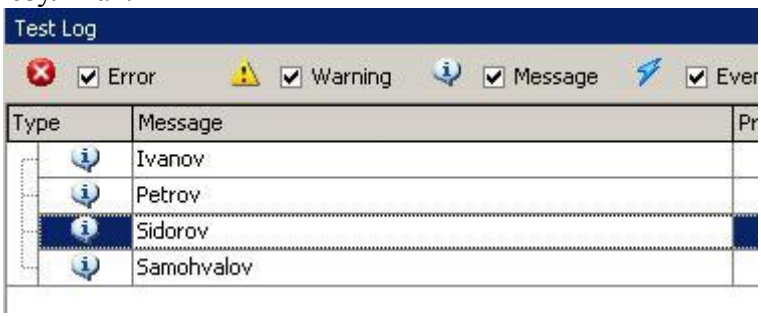


Теперь можно создать пример работы с базой через BDE:

```
function TestBDE()
{
  var d, w;
  d = BDE.CreateDatabase();
  d.DatabaseName = "MSACCESS1";
  d.AliasName = "MSACCESS1";
  d.LoginPrompt = 0;
  d.Connected = 1;
  w = BDE.CreateTable();
  w.DatabaseName = "MSACCESS1";
  w.TableType = 0;
  w.TableName = "tblData";
```

```
  w.Open();
  w.First();

  while (!w.EOF)
  {
    Log.Message(w.FieldByName("fname").AsString);
    w.Next();
  }
}
```

Результат:



ActiveX

Если компонент для работы с базами данных является ActiveX-компонентом, то его можно добавить в TestComplete как ActiveX элемент и работать с ним через ActiveX редактор и в скриптах. В этом случае вы получаете доступ ко всем свойствам и методам ActiveX элемента.

9 Object Driven Testing (Тесты, управляемые объектами)

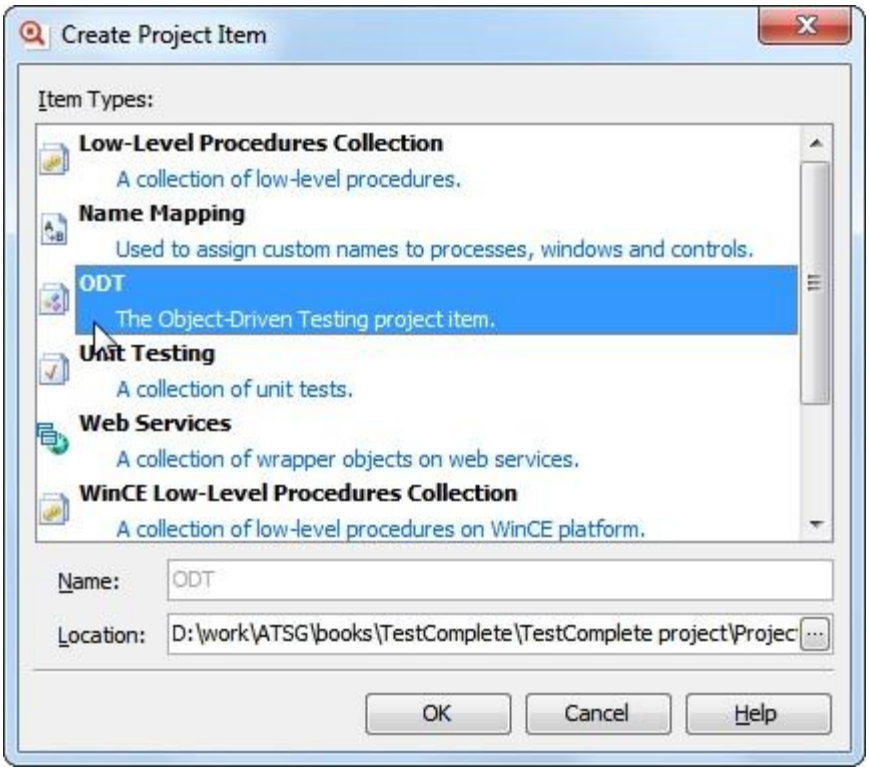
Object Driven Testing (тесты, управляемые объектами) – это такой подход к автоматизации тестирования, при котором тестовые скрипты проектируются в виде классов, в которых реализуется логика работы с приложением. Такие скрипты легче создавать и поддерживать, так как в тесткейсах используются лишь методы «высокого уровня», позволяя скрыть подробности реализации тех или иных действий.

В TestComplete есть специальный элемент проекта ODT, который и позволяет представить скрипты в виде классов со свойствами и методами.

Сразу заметим, что в языках JScript/C++Script/C#Script и VBScript вы можете создавать классы, пользуясь только встроенными средствами этих языков. Эти возможности кратко рассмотрены в конце этой главы. А здесь мы рассмотрим возможности объекта ODT.

Использование ODT

Прежде, чем начать использовать ODT, его необходимо добавить в проект. Правый щелчок на имени проекта, Add – New Item, ODT.



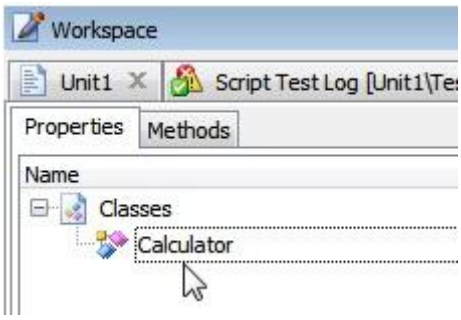
После чего в проекте появится новый элемент ODT с двумя дочерними элементами: Classes и Data.

Структура данных в ODT разделена на классы и данные. Классы содержат свойства (которые могут быть как обычными переменными, так и массивами) и методы (имена методов привязываются к существующим функциям). Данные могу содержать обычные переменные, массивы и объекты классов. В отличие от большинства объектно-

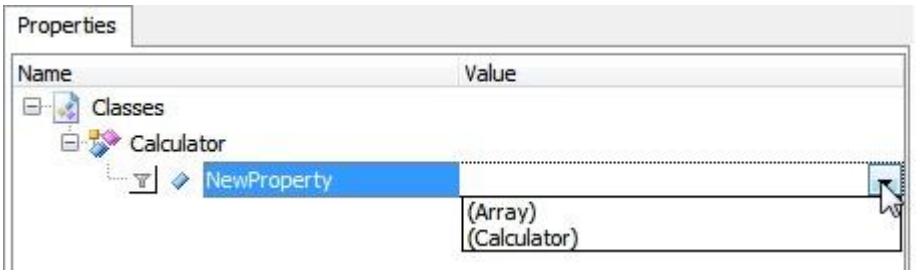
ориентированных языков, в ODT есть интересная особенность: объекты класса могут содержать дополнительные (собственные) методы, которые будут доступны только в этом объекте, но не в других объектах этого же класса.

В качестве примера создадим простой класс и объект для Калькулятора.

Дважды щелкнем на элементе Classes в Project Explorer, затем в редакторе справа щелкнем правой кнопкой мыши по элементу Classes и выберем пункт меню New Item. Добавится новый класс NewClass. Нажмем клавишу F2 и переименуем класс в Calculator. Обратите внимание, что теперь у нас стала доступной вкладка Methods. Эта вкладка доступна только тогда, когда в редакторе выделен класс или объект класса.

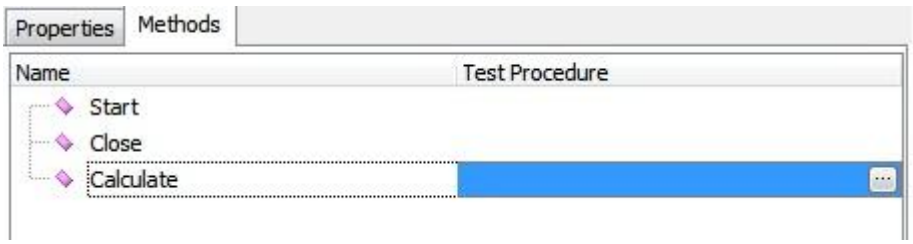


Если теперь щелкнуть правой кнопкой мыши на имени класса и выбрать пункт меню New Item, то добавится новое свойство:



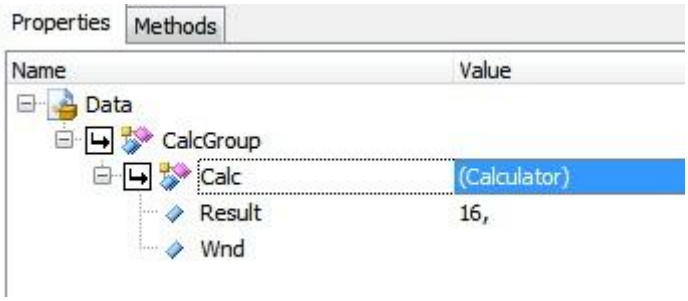
В колонку Value можно ввести какое-то значение напрямую (строку, число, true/false) или выбрать значение (Array), чтобы создать переменную типа массив. Мы назовем это свойство Result и не будем сейчас присваивать ему никакого значения. Это свойство будет в дальнейшем использовано для хранения результата вычислений.

Теперь выделим в редакторе класс Calculator и перейдем на вкладку Methods. Здесь мы определим 3 метода (правый щелчок мышью, New Item), как показано на скриншоте ниже.



Как уже было сказано, имена методов связываются с существующими функциями в проекте, для чего на вкладке Methods есть колонка Test Procedure. Мы создадим необходимые процедуры позже, а сейчас перейдем к созданию данных (объектов).

Для этого в Project Explorer выберем элемент Data, затем в редакторе данных щелкнем правой кнопкой мыши и выберем пункт меню New item. При этом создастся новая группа NewGroup. Данные (или объекты) в ODT сгруппированы по группам, в каждой группе может хранится сколько угодно объектов. Мы назовем нашу группу CalcGroup, после чего добавим в нее новый элемент (Calc) и свяжем его с классом Calculator, как показано на скриншоте ниже.



Обратите внимание, что как только мы связали объект Calc с классом Calculator, мы сразу видим его свойство Result, которое мы определили для класса. Дополнительно определим еще одно свойство Wnd. С помощью этого свойства мы будем обращаться к главному окну Калькулятора.

Также на вкладке Methods мы можем определить дополнительные методы, которые будут доступны для этого объекта, но не для класса и не для других объектов этого класса.

Выше мы определили 3 метода для класса Calculator и сейчас самое время создать для них тестовые процедуры. Для этого в любом модуле проекта вставим следующий код:

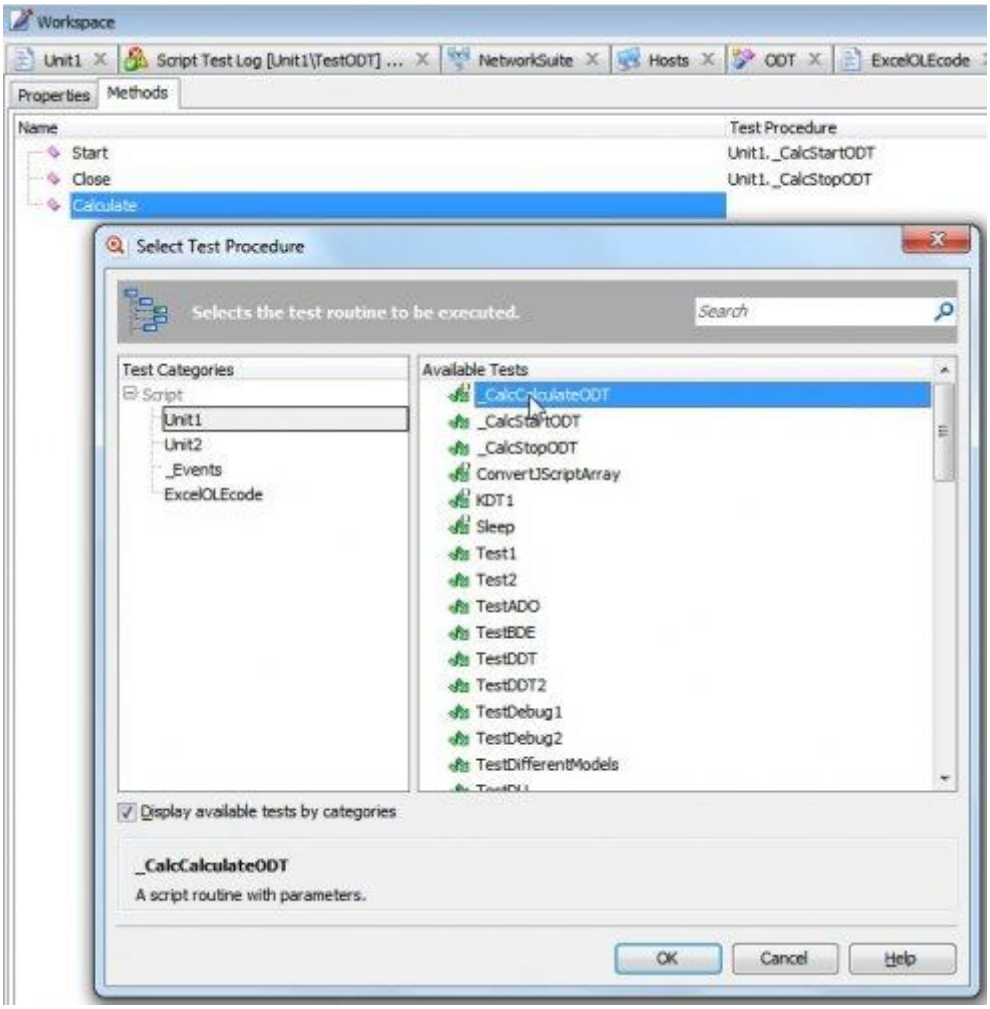
```
function _CalcStartODT()
{
  TestedApps.calc.Run();
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus",
1);
  ODT.Data.CalcGroup.Calc.Wnd = wCalc;
  ODT.Data.CalcGroup.Calc.Result = wCalc.Window("Edit", "", 1).wText;
}

function _CalcStopODT()
{
  Sys.Process("CalcPlus").Terminate();
}

function _CalcCalculateODT(expression)
{
  var i;
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus",
1);
  for(i = 0; i < expression.length; i++)
  {
    wCalc.Keys(expression.substr(i, 1));
  }
  wCalc.Keys("=");
  This.Result = wCalc.Window("Edit", "", 1).wText;
  return This.Result;
}
```

Функция _CalcStartODT запускает Калькулятор и присваивает переменной Wnd объекта Calc объект Window(“SciCalc”), который является главным окном Калькулятора, а также

присваивает свойству Result значение текстового поля Калькулятора; функция _CalcStopODT закрывает Калькулятор, попросту убивая его процесс; функция _CalcCalculateODT позволяет рассчитать значение переданного выражения с помощью калькулятора. Кроме того, функция _CalcCalculateODT демонстрирует, как можно с помощью ключевого слова This (или Self в случае DelphiScript) из метода обратиться к собственному объекту. Обратите внимание, что в данном случае ключевые слова This и Self необходимо писать с большой буквы, так как аналогичные слова в нижнем регистре (this и self) являются зарезервированными словами в языках программирования.

Функции могут называться как угодно, мы же использовали суффик ODT в их именах просто для наглядности.

Теперь самое время связать методы класса с созданными функциями. Для этого вернемся в список методов созданного ранее класса Calculator и выберем для каждого метода соответствующую функцию, как показано ниже.

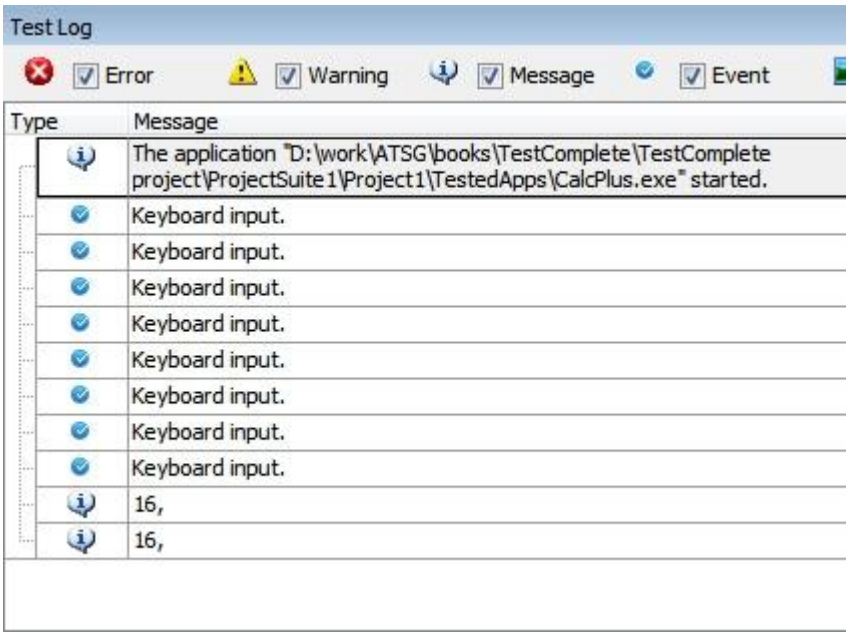


Теперь мы готовы написать простой тест, который в Калькуляторе вычисляет значение переданного выражения и результат выводит в лог.

```
function TestODT()
{
  var calc = ODT.Data.CalcGroup.Calc;
  calc.Start();
  Log.Message(calc.Calculate("(5+3)*2"));
  Log.Message(ODT.Classes.Calculator.Result);
```

```
  calc.Close();
}
```

Результат работы программы:



Как видите, хотя нам и пришлось потрудится, создавая структуру данных в ODT, соответствующие функции и связывая все эти данные вместе, в результате мы получили очень простой и удобочитаемый скрипт, который легко понять и отредактировать в случае надобности.

Object Driven Testing – это очень мощный инструмент, который позволяет создавать тестовые скрипты высокого класса.

Кроме рассмотренных выше возможностей, TestComplete позволяет создавать классы и объекты ODT динамически (т.е. во время работы скрипта). Если вас интересует эта возможность, обратитесь к справочной системе TestComplete, раздел «Creating Custom Objects Programmatically».

В случае использования JScript/C++Script/C#Script и VBScript подобного эффекта можно добиться и с помощью встроенных возможностей языков. Ниже кратко рассматриваются эти возможности.

ODT и JScript

Язык JScript изначально является объектно-ориентированным, что означает, что в нем также можно создавать объекты и классы. Пример объявления класса со свойствами и объектами в JScript:

```
function MyClass(arg1)
{
  // создаем свойство
  this.myProperty = arg1;

  // объявляем метод
  this.method = function(arg2)
```

```
  {
    Log.Message("Method #1. Text: " + arg2);
  }
}
```

Сама функция MyClass, как видно из примера, выступает в роли конструктора класса, в котором мы присваиваем свойству myProperty переданное значение arg1. И пример использования созданного класса:

```
function TestJscriptClass()
{
  // создаем объект класса MyClass
  var myVar = new MyClass("some value");
  myVar.method("val1"); // вызов метода
  Log.Message(myVar.myProperty);// обращение к свойству
}
```

Таким образом можно написать свои классы со свойствами и методами для работы с тестируемым приложением. Подробнее об ООП в JavaScript-е можно узнать здесь.

У TestComplete есть один недостаток, связанный с подобным объявлением классов и методов. Если мы объявляем обычную функцию, а затем используем ее где-то, то мы можем затем легко перейти к её исходному коду из любого места, где используется эта функция, зажав клавишу Ctrl и щелкнув мышью по имени функции. В случае с классами и методами, однако, такой переход невозможен. Кроме того, с непривычки может оказаться тяжело находить в коде место, где произошла ошибка, когда мы дважды щелкаем по ошибке в логе.

ODT и VBScript

VBScript изначально был процедурно-ориентированным языком, но начиная с версии 5.0 в него была добавлена возможность создания классов. Например:

```
Class MyClass

  Private m_prop


  Private Sub Class_Initialize   ' конструктор

    Log.Message "Конструктор класса MyClass"

    m_prop = "init value"

  End Sub


  Private Sub Class_Terminate   ' деструктор

    Log.Message "Деструктор класса MyClass"
```

```
  End Sub


  Public Function Method1
    Method1 = "return value"
  End Function


  Public Property Get MyVar
    MyVar = m_prop
  End Property


  Public Property Let MyVar(value)
    m_prop = value
  End Property
End Class
```

И пример использования класса:

```
Sub TestClass
  Set myvar = New MyClass
  Log.Message myvar.Method1
  Log.Message myvar.MyVar
  myvar.MyVar = 123
  Log.Message myvar.MyVar
  Set myvar = Nothing
End Sub
```

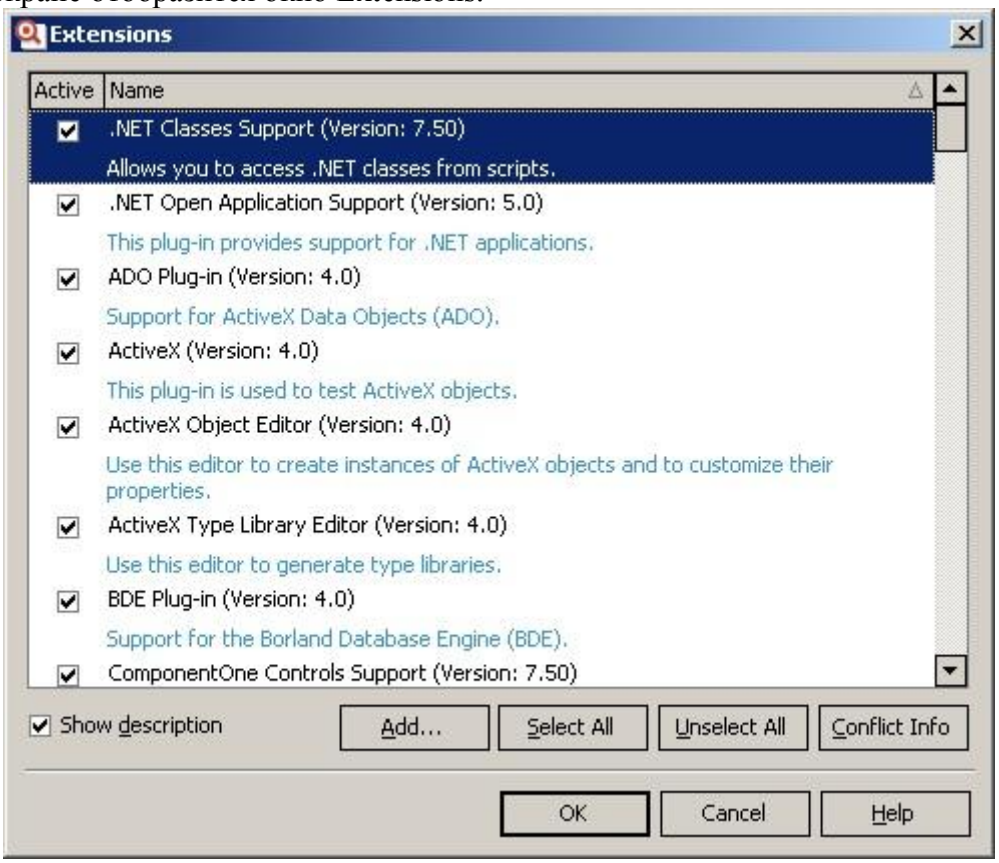
Как видите, возможности VBScript в ООП тоже весьма неплохие.

Итог

Из приведенных выше примеров видно, что ни возможности TestComplete, ни встроенные возможности языков, не дают достаточно широких возможностей ООП. Если вы хотите создавать в TestComplete более сложные скрипты, используя более современные языки и среды разработки, обратитесь к главе 5 Присоединяемые и Самотестируемые приложения.

10 Создание собственных надстроек (Extensions)

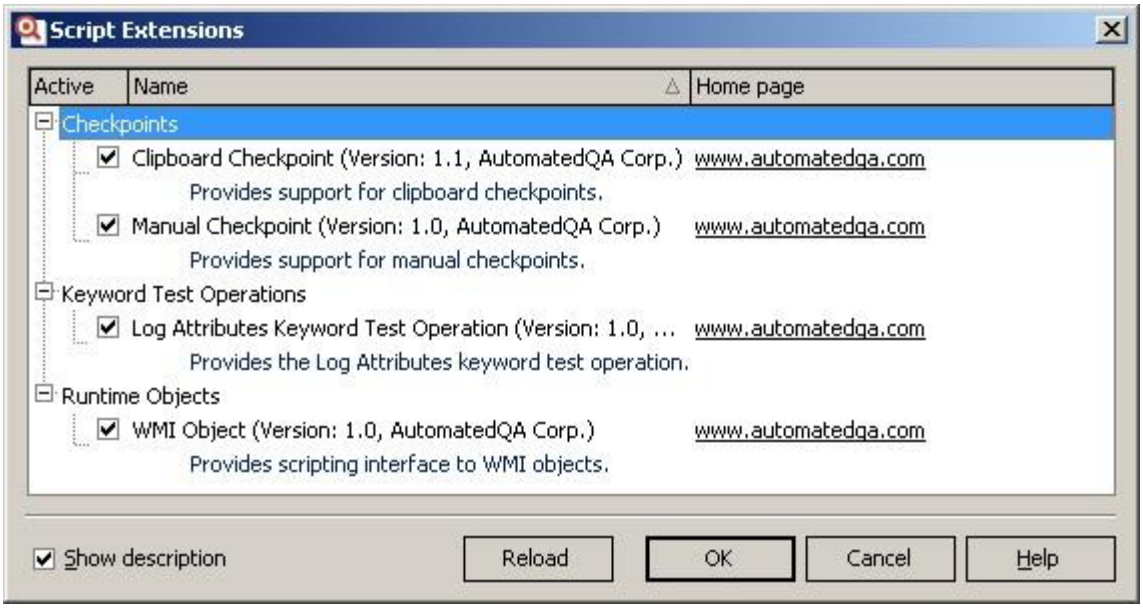
Большинство возможностей в TestComplete реализовано с помощью надстроек (Extensions). Например, работа с ADO, поддержка .NET приложений, пользовательские формы и многое, многое другое – всё это реализовано с помощью надстроек (или плагинов – plug-ins). Чтобы посмотреть полный список установленных надстроек, необходимо в TestComplete выбрать пункт меню *File – Install Extension*… При этом на экране отобразится окно Extensions.



Здесь можно посмотреть список всех доступных надстроек и отключить те из них, которые не используются.

Для создания собственных плагинов необходимо скачать TestComplete SDK на сайте AutomatedQA.

Кроме обычных надстроек есть так называемые скриптовые надстройки. Их отличие от обычных надстроек в том, что они создаются с помощью самого TestComplete без использования сторонних средств разработки. Посмотреть список скриптовых надстроек можно, выбрав пункт меню *File – Install Script Extensions.*



С помощью скриптовых надстроек можно создавать новые программные объекты (например, объекты, аналогичные объектам ADO, DDT, BuiltIn), новые действия, которые доступны во время записи скриптов (аналогично checkpoint-ам) и новые действия для Keyword-Driven тестов.
Конечно, с помощью языков JScript и VBScript можно создать подобные объекты и не прибегая к использованию надстроек, однако такие объекты менее удобны: при использовании методов этих объектов вы не увидите всплывающей подсказки с описанием параметров методов, а в случае, если вы захотите передать свой объект кому-то, вам придется его передавать в виде отдельного модуля или просто текстового файла с дополнительным описанием всех свойств и методов. В случае использования надстроек эти проблемы решаются.
В этой главе в качестве примера мы рассмотрим создание Runtime Object-ов. Создание Checkpoint-ов и Keyword-driven Extension-ов очень похоже на создание Runtime объектов, а использование SDK выходит за рамки этого пособия.

Создание новых объектов (Runtime Objects)

Создание своих объектов позволяет улучшить читабельность кода и упростить доступ к различным функциям, предназначенным для работы с одной сущностью. Например, TestComplete предоставляет возможность работать с MS Excel через OLE. Это единственный способ считать данные из произвольной ячейки в Excel-файле (так как объект DDT предоставляет последовательный доступ к строкам, что не всегда удобно).
Предположим, мы хотим создать новый объект с названием ExcelOLE, с помощью которого и будем считывать/записывать данные. У этого объекта должны быть следующие методы:

- **Create** – создать новый файл
- **Open** – открыть существующий файл
- **Close** – закрыть подключение к файлу

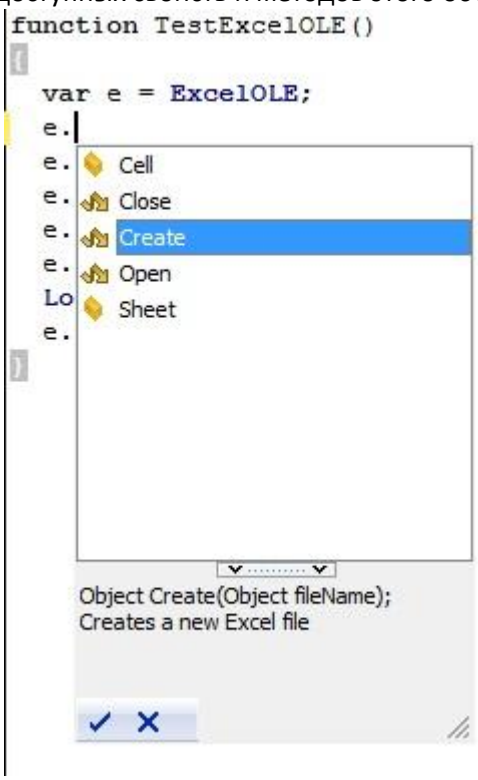
И свойства:

- **Sheet** – текущая рабочая страница
- **Cell** – ячейка

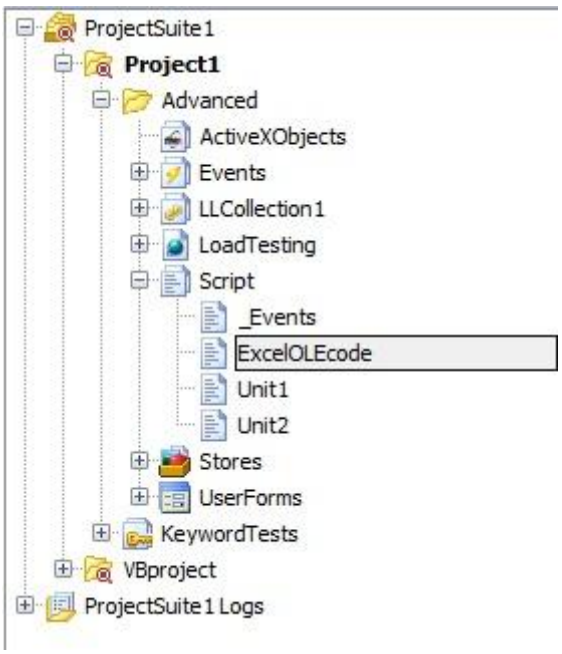
Методы Create и Open должны принимать в качестве параметров имя файла, свойство Cell должно принимать 2 параметра: строку и колонку. Для простоты мы будем задавать колонку ее номером, а не именем (т.е. колонка A=1, B=2 и т.д.).
То есть в идеале мы должны будем написать следующую функцию

```
function TestExcelOLE()
{
  var ee = ExcelOLE;
  ee.Create("c:\\file1.xls");
  ee.Open("c:\\file1.xls");
  ee.Sheet = "Лист1";
  ee.Cell(3, 3) = "New text";
  Log.Message(ee.Cell(3, 3));
  ee.Close();
}
```

и она должна создать новый файл, записать текст «New text» в ячейку C3, а затем считать записанное значение и вывести в лог. Причем в редакторе TestComplete мы будем видеть список доступных свойств и методов этого объекта.



Приступим.
Прежде всего создадим новый модуль, причем его имя должно отличаться от имени объекта. Мы назовем объект ExcelOLE, а модуль – ExcelOLEcode.



В модуль необходимо поместить все объекты, с которыми будут работать методы нашего объекта. В нашем случае это будут:
`var Sheet` – текущий рабочий лист
`var oWorkBook` – текущий файл
Далее необходимо определить функции-методы. Имена функций могут отличаться от имен методов, которые вы хотите видеть в своем объекте, их соответствия все равно придется устанавливать вручную. Мы же будем называть функции точно так же, как будут называться методы объекта (create, Open и Close).

```
function Create(fileName)
{
  oWorkBook = Sys.OleObject("Excel.Application").Workbooks.Add();
  oWorkBook.SaveAs(fileName);
  oWorkBook.Close();
  Sys.OleObject("Excel.Application").Quit();
}

function Open(fileName)
{
  oWorkBook = Sys.OleObject("Excel.Application").Workbooks.Open(fileName);
}

function Close()
{
  oWorkBook.Save();
  oWorkBook.Close();
  Sys.OleObject("Excel.Application").Quit();
}
```

Как видите, пока ничего сложного. Мы просто создаем обычные функции, которые работают с обычными переменными. Теперь перейдем к свойствам.
Свойства бывают двух типов: обычные (в нашем примере это Sheet) и индексируемые (в нашем примере это свойство Cell, которое принимает 2 параметра: строку и колонку). Как и во многих языках программирования, для того, чтобы работать со свойствами, нам необходимо для каждого из них указать пару методов-аксессоров get/set. Причем можно как указать оба аксессора, так и один из них, тогда свойство будет доступно только для чтения или только для записи. И опять же, имена методов-аксессоров могут быть любыми, а соответствие между методами и свойствами будут установлены позже. Вот как выглядят

аксессоры для свойства Sheet:

```
function GetSheet()
{
  return Sheet;
}

function SetSheet(sheetName)
{
  Sheet = sheetName;
}
```

Здесь мы связываем методы-аксессоры с объявленной в начале переменной Sheet.
А для свойства Cell нам переменная не нужна. Мы объявим методы-аксессоры, которые будут напрямую работать с ячейкой Excel'я. Вот как они будут выглядеть:

```
function CellRead(row, col)
{
  return oWorkBook.Sheets(Sheet).Cells(row, col).value;
}

function CellWrite(row, col, text)
{
  oWorkBook.Sheets(Sheet).Cells(row, col).value = text;
}
```

Как видите, в этих методах мы обращаемся к переменной oWorkBook, которая предварительно должна быть инициализирована с помощью метода Open, описанного выше.
Теперь, когда наш код написан и отлажен, мы можем создать собственно описание объекта. Для этого нам понадобится создать xml-файл в любом редакторе с названием description.xml. Вот содержимое, которое необходимо поместить в этот файл:

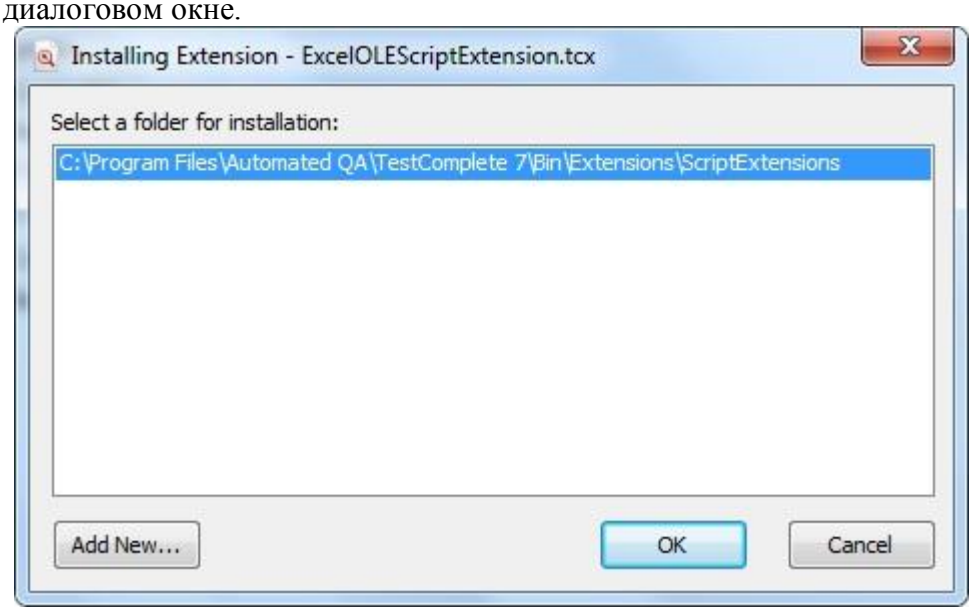
```
<?xml version="1.0" encoding="UTF-8"?>
<ScriptExtensionGroup>
  <Category Name="Runtime Objects">
    <ScriptExtension Name="Excel OLE" Author="Gennadiy Alpaev"
Version="1.0" HomePage="www.tctutorial.ru">
      <Script Name="ExcelOLEcode.js">
        <RuntimeObject Name="ExcelOLE">
          <Method Name="Open" Routine="Open">
            Opens an Excel file as OLE object
          </Method>
          <Method Name="Create" Routine="Create">
            Creates a new Excel file
          </Method>
          <Method Name="Close" Routine="Close">
            Closes an Excel file
          </Method>
          <Property Name="Cell" GetRoutine="CellRead"
SetRoutine="CellWrite">Excel cell</Property>
          <Property Name="Sheet" GetRoutine="GetSheet"
SetRoutine="SetSheet">Excel sheet</Property>
          <Description>Provides access to Excel files via
OLE</Description>
        </RuntimeObject>
      </Script>
```

```
    </ScriptExtension>

  </Category>
</ScriptExtensionGroup>
```

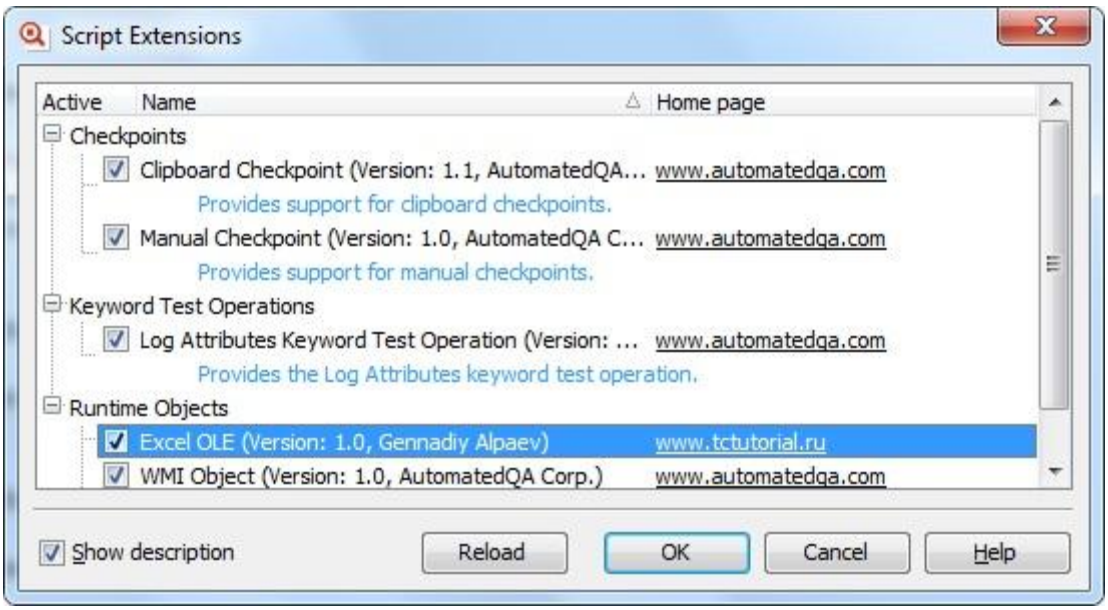
Рассмотрим по очереди все секции этого xml-файла.

- **ScriptExtensionGroup** – указывает на начало специального файла-описания настройки TestComplete
- **Category** – вид надстройки (в нашем случае - Runtime Objects)
- **ScriptExtension** – информация о надстройке (имя, автор и т.п.)
- **Script** – имя файла, в котором хранится исполняемый код
- **RuntimeObject** – здесь хранится имя объекта, как оно будет отображаться в окне автозаполнения редактора TestComplete

Далее идут перечисления методов (Method) и свойств (Property). Для каждого метода указывается его имя (Name=) и имя функции, которая будет выполняться при вызове этого метода (Routine=), а также описание метода (внутри тега Method). Для каждого свойства указываются его имя, get-аксессор, set-аксессор (Name, GetRoutine, SetRoutine) и также описание внутри тега Property.
Последним идет тег Description с описанием объекта Обратите внимание, что тег description должен быть именно последним!
Теперь нам необходимо создать собственно файл скриптовой надстройки из имеющихся двух файлов (файл кода и файл description.xml). Для этого оба файла надо поместить в одну папку, заархивировать в ZIP-архив любым архиватором и сменить расширение на tcx.
Надстройка создана и готова для инсталляции! Чтобы установить надстройку, просто щелкните по ней двойным щелчком в Проводнике и нажмите OK в открывшемся диалоговом окне.



Затем зайдите в TestComplete в меню *File – Install Script Extension*, нажмите кнопку Reload и включите чекбокс напротив имени подключенной надстройки.



Теперь у вас есть новый объект, который вы можете использовать точно так же, как и любой встроенный объект TestComplete-а.

11 Другие возможности

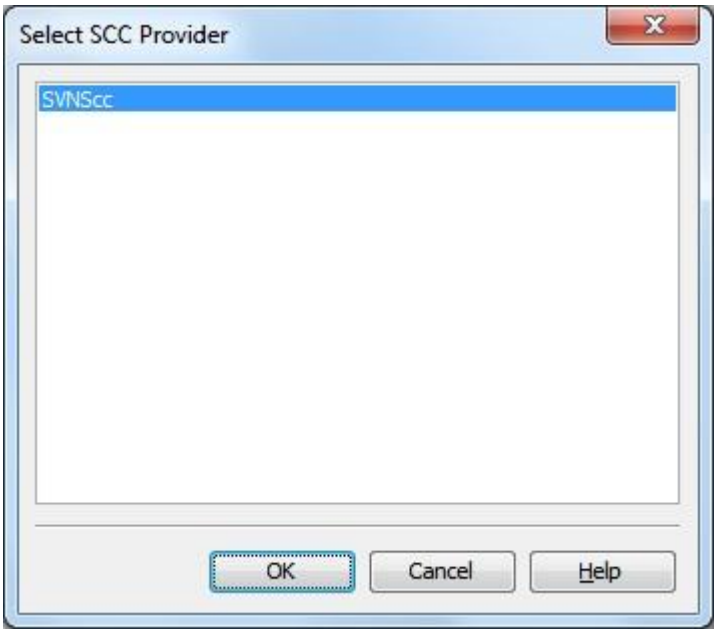
В этой главе рассматриваются различные возможности TestComplete, которые не были описаны в других главах.

11.1 Интеграция с системами контроля версий

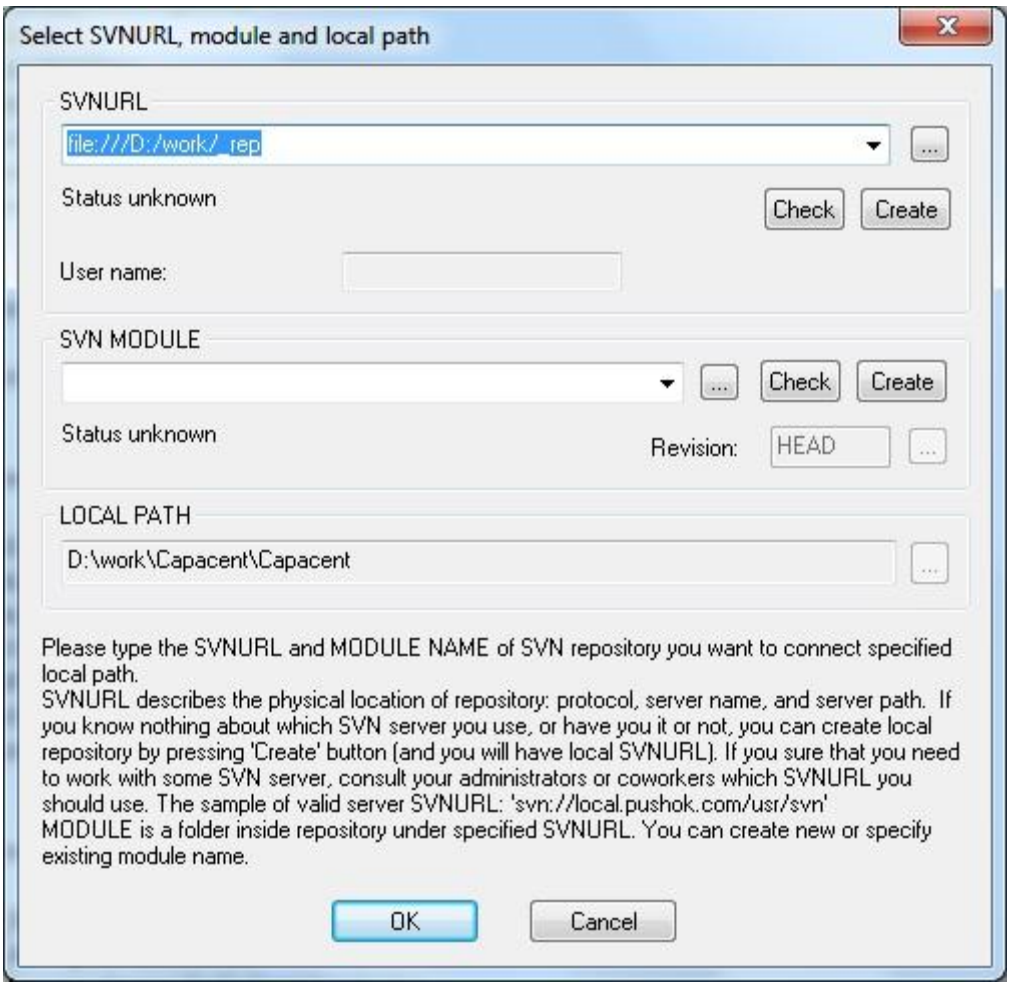
TestComplete поддерживает работу с системами контроля версий, позволяя автоматически выполнять действия по добавлению файлов в репозиторий, забирать последнюю версию файлов и т.д.

Если система контроля версий является совместимой с MS Source Code Control API (SCC), TestComplete может с ней работать (полный список поддерживаемых систем можно найти в справочной системе TestComplete, раздел Integration With Source Code Control Systems). Для некоторых систем (например, SVN) необходимо устанавливать дополнительные модули, которые добавляют поддержку SCC API системе контроля версий (SCC proxy provider).

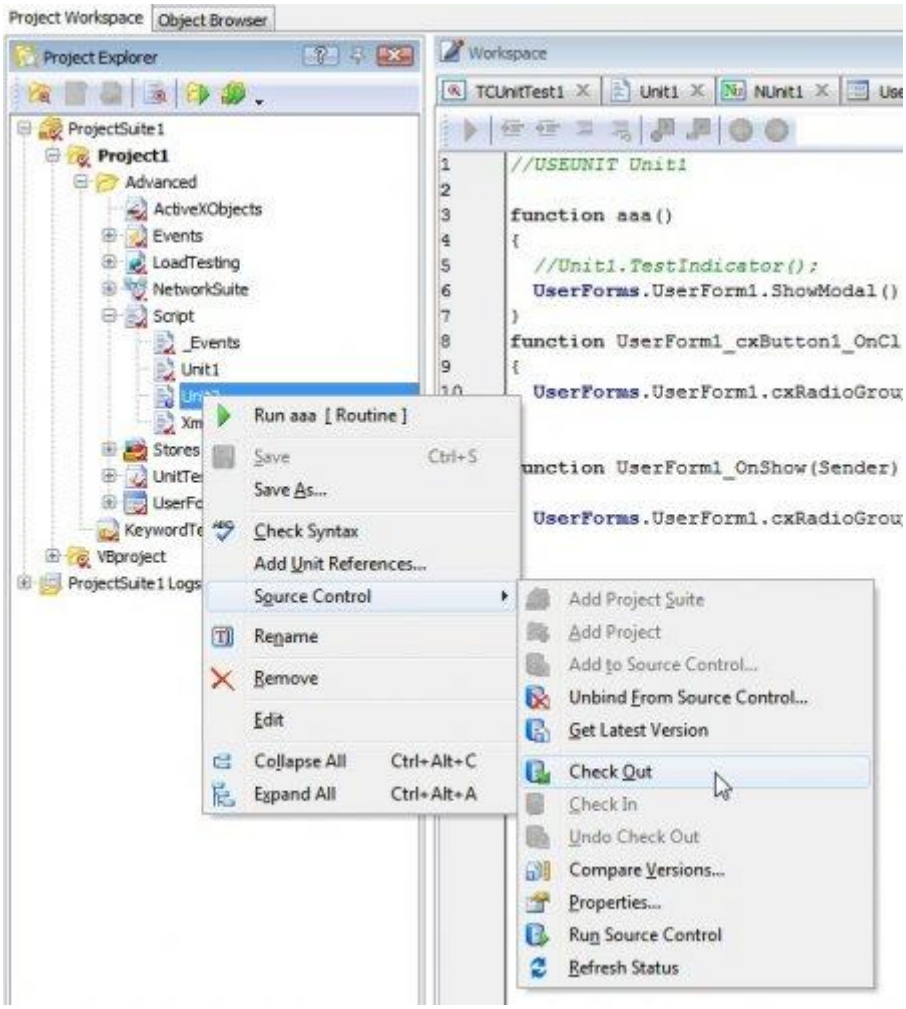
Чтобы добавить набор проектов в репозиторий, необходимо щелкнуть правой кнопкой мыши в Project Explorer и выбрать пункт Source Control – Add Project Suite, после чего в открывшемся диалоговом окне Select SCC Provider выбрать подходящий провайдер.



После чего откроется окно SCC провайдера, в котором необходимо указать параметры добавления проекта. Например, такое:

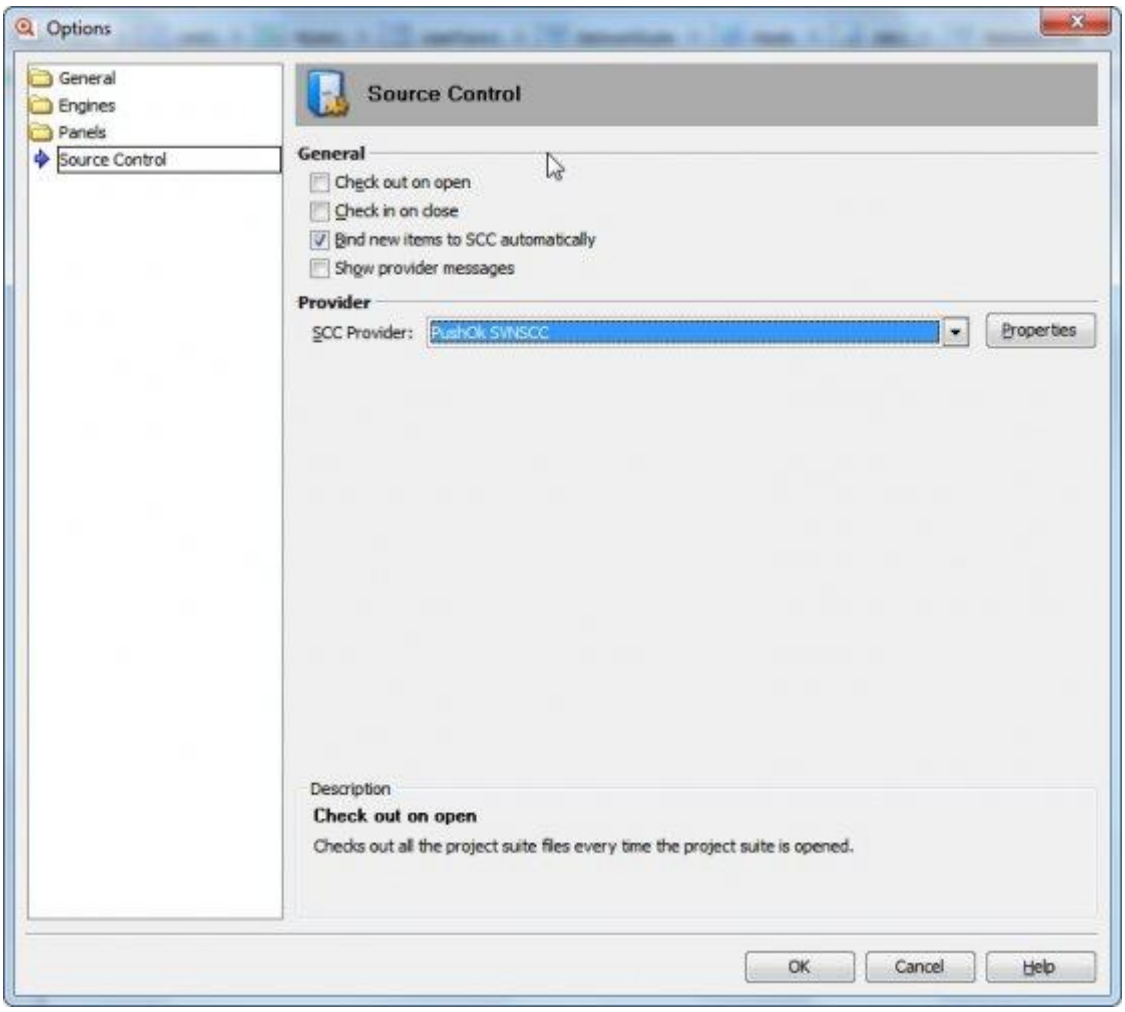


После того, как проект добавлен в систему контроля версий, все элементы в Project Explorer будут иметь дополнительные значки, указывающие на статус файла (checked-in, checked-out и т.п.). Соответственно в зависимости от статуса файла будут меняться доступные пункты меню. Например:

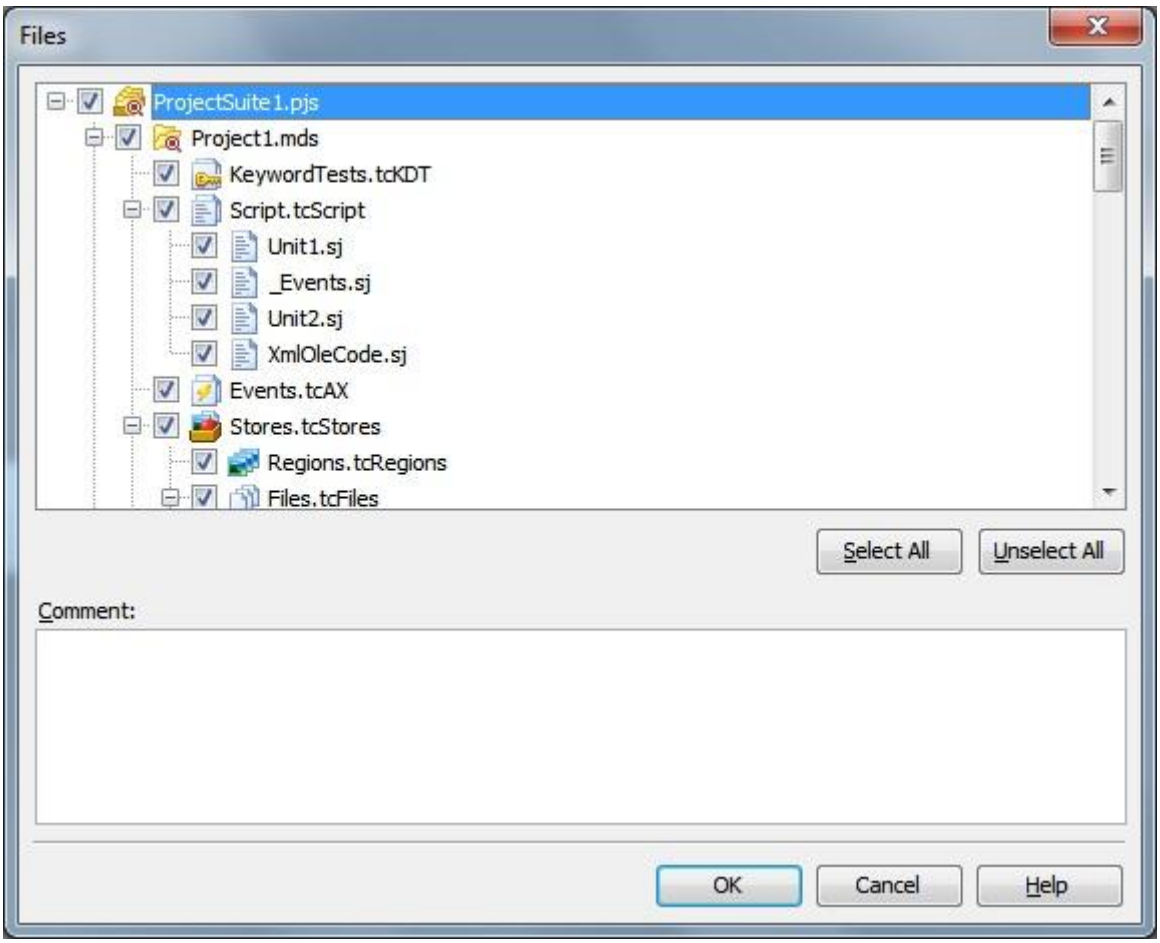


При попытке отредактировать залоченный файл, TestComplete выдаст сообщение, предлагая сначала выполнить операцию check-out для этого файла.

Для настройки параметров работы с системами контроля версий необходимо выбрать пункт меню Tools – Options и выделить элемент Source Control в дереве опций.



- **Check out on open/Check in on close** – автоматически делать check-out/check-in при открытии/закрытии проекта. При этом при закрытии TestComplete будет появляться окно для ввода комментария и выбора файлов для операции check-in



- **Bind new items to SCC automatically –** автоматически добавлять новые файлы проекта в систему контроля версий
- **Show provider messages –** показывать сообщения об ошибках от провайдера, которые получает TestComplete
- **SCC provider –** выбор провайдера SCC. Нажав на кнопку Properties можно настроить параметры провайдера

Несколько замечаний по использованию систем контроля версий в TestComplete:

- Если репозиторий находится на удаленной машине, связь с которой медленная, лучше не связывать TestComplete с SCC, так как при этом навигация по проекту может замедлится
- Файлы *.tcCfgExtender и *.tcLS не стоит добавлять в систему контроля версий, так как эти файлы содержат локальные настройки, которые различаются для каждого пользователя

11.2 Запуск TestComplete из командной строки

При запуске TestComplete можно передать ему определенные параметры через командную строку. Эту возможность можно использовать, например, для регулярных запусков скриптов ночью или на выходных, когда за компьютером нет человека.

Общий вид командной строки TestComplete:

TestComplete.exe имя_файла */параметр:значение /параметр:значение*

- **имя_файла** – это либо имя набора проектов (например, ProjectSuite1.pjs), либо имя проекта (например, VBproject.mds). Если указан этот параметр, TestComplete автоматически откроет указанный файл
- **/run** (или **/r**) – если указан этот параметр, TestComplete откроет проект (или набор проектов) и запустит тесты на выполнение. Если не указывать, что именно запускать, TestComplete запустит все включенные Test Item-ы из всех включенных наборов проектов проектов (т.е. все, у которых включена опция Enabled). Или же можно явно указать, что именно надо запустить:
    - **/project:**имя_проекта (или **/p:** ...) – запустить все включенные Test Item-ы проекта
    - **/project:**имя_проекта **/projectitem:**имя_элемента (или **/p:** ... **/pi:** ...) – запустить все тесты, принадлежащие указанному элементы проекта. Элемент проекта – это Scripts, NetworkSuite, LoadTesting, KeywordTests
    - **/project:**имя_проекта **/unit:**имя_модуля **/routine:**имя_функции (или **/p:** ... **/u:** ... **/r:** ...) – запустить какую-то конкретную функцию или процедуру

  Обратите внимание на следующие особенности:

    - нет возможности запустить какой-то отдельный Test Item, только процедуру или функцию
    - между параметрами и их значениями необходимо ставить двоеточие (например, /p:MyProject). Никаких пробелов там быть не должно
- **/exit** (или **/e**) – этот параметр указывает TestComplete-у, что после окончания запуска всех скриптов сам TestComplete должен быть закрыт
- **/SilentMode** – это параметр заставляет TestComplete выполнять скрипты в так называемом "тихом режиме", при котором подавляется вывод любых сообщений, требующих реакции пользователя. При этом все сообщения будут помещены в файл *<TestComplete>\Bin\Silent.log*. Эта опция очень полезна, если скрипты запускаются без присутствия пользователя за компьютером
- **/ns** – не отображает splash screen при запуске (картинка с логотипом TestComplete, которая первой появляется при запуске TestComplete и висит на экране до тех пор, пока не загрузится главное окно)

Приведем несколько примеров командной строки с объяснениями:

"C:\Program Files\Automated QA\TestComplete 7\Bin\TestComplete.exe" "C:\My Projects\MySuite.pjs" /r /p:MyProj
Открывает набор проектов MySuite.pjs и запускает все включенные Test Item-ы из проекта MyProj

"C:\Program Files\Automated QA\TestComplete 7\Bin\TestComplete.exe" "C:\Work\My Projects\MySuite.pjs" /r /e
Открывает набор проектов MySuite.pjs, запускает все включенные проекты из этого набора проектов, а затем закрывает TestComplete

"C:\Program Files\Automated QA\TestComplete 7\Bin\TestComplete.exe" "C:\Work\My Projects\MySuite.pjs" /r /p:MyProj /u:Unit1 /rt:Main
Открывает набор проектов MySuite.pjs и запускает функции Main из модуля Unit1 проекта MyProj.

11.3 Использование библиотеки распознавания текста (OCR)

Enterprise версия TestComplete содержит плагин **OCR (Optical Character Recognition)** – система оптического распознавания текста. Возможности OCR позволяют распознавать 52 латинских символа в верхнем и нижнем регистрах, 10 цифр и 31 специальный символ практически любых стилей, размеров и начертаний (жирный, курсив и т.п.). Не поддерживаются национальные символы (например, русские или китайские), а также символы, написанные с помощью специальных шрифтов (например, Wingdings, Webdings и т.д.). Практически невозможно распознавать так называемые капчи (т.е. картинки, которые используются для отличия программ-роботов от живых людей).

Еще один важный аспект использования системы распознавания текстов – это сглаживание шрифтов. Обычно в Windows используется один из способов сглаживания экранных шрифтов, что может повлиять на результаты распознавания текста. Поэтому сглаживание рекомендуется отключить в Панели управления (*Start – Settings – Control Panel – Display – Appearance – Effects – Use the following method to smooth edges of screen fonts*).

Система распознавания текста может быть полезна при распознавании отсканированных документов или при работе с самописными элементами управления, к свойствам и методам которых невозможно получить доступ в TestComplete.

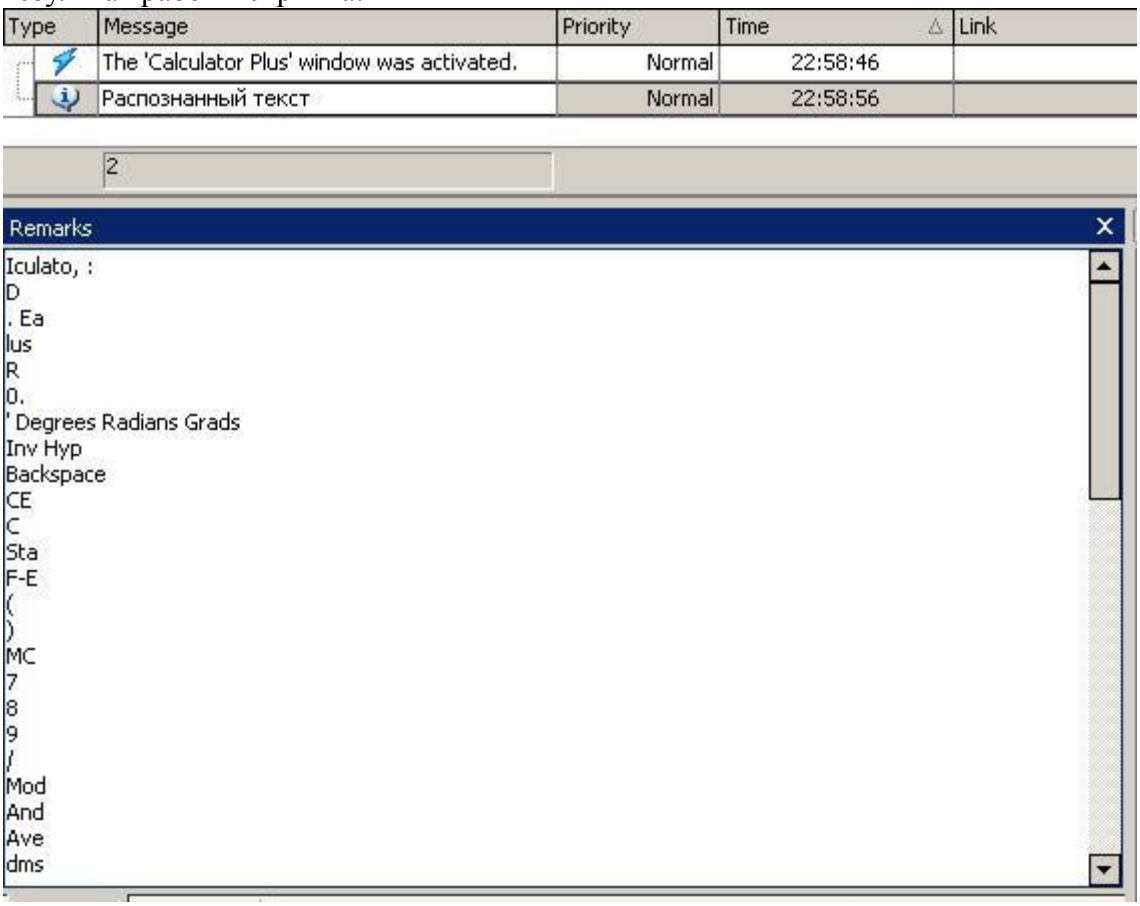
Теперь рассмотрим пример работы с распознаванием текста. Для этого используется объект OCR, у которого есть только один метод – CreateObject, который принимаетв качестве параметра любой экранный объект и возвращает объект OCRObject. Например, в следующем примере мы создадим OCRObject для окна Калькулятора:

```
function TestOCR()
{
  var  wnd;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate()
  var OCRobj = OCR.CreateObject(wnd);
}
```

Теперь мы можем распознать весь текст внутри окна Калькулятора с помощью метода GetText:

```
  var OCRobj = OCR.CreateObject(wnd);
  var sText = OCRobj.GetText();
  Log.Message("Распознанный текст", sText);
```

Результат работы скрипта:



Как видно из результата, практически все надписи на элементах управления распознались хорошо, независимо от цвета текста (а также при том, что было включено сглаживание экранных шрифтов). Проблемы возникли только с заголовком Калькулятора, а также незначительные ошибки вроде буквы I(i заглавная) вместо l (L прописная), однако система OCR в TestComplete не универсальна и подобные ошибки неизбежны.

В следующем примере мы воспользуемся методом FindRectByText для поиска координат текста в окне Калькулятора. Мы будем искать отдельные цифры (например, 7,8 и 9), получать их координаты и затем щелкать мышью по этим координатам. Если все пройдет как надо, то в результате мы введем цифры 789 в Калькулятор. Скрипт:

```
function TestOCR()
{
  var arr = new Array("7", "8", "9");
  var  wnd, i;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate()

  var OCRobj = OCR.CreateObject(wnd);

  for(i = 0; i < arr.length; i++)
  {
    if(OCRobj.FindRectByText(arr[i]))
    {
      wnd.Click(OCRobj.FoundX, OCRobj.FoundY)
    }
    else
    {
      Log.Error("Text '" + arr[i] + "' not found");
```

```
        }
    }

}
```

Результат:

Как видим, все прошло успешно!

Здесь мы не рассматриваем более тонкие настройки для системы распознавания текста OCROptions, в которых можно более точно задавать параметры распознавания, однако обязательно ознакомьтесь с ними в справочной системе TestComplete, если вы планируете часто пользоваться системой OCR.

Кроме того, обратите внимание на то, что хотя TestComplete и не распознает кириллические символы, у нас все же есть возможность частичного распознавания русских текстов в приложении, так как многие символы в русском и английском языках очень похожи. Конечно, такое распознавание будет далеко не полным, однако в некоторых случаях (например, когда надо найти текст в каком-то элементе управления и щелкнуть на нем) этого может оказаться вполне достаточно.

11.4 Вызов API-функций и функций из DLL

Вызов функций из DLL

Вызов DLL-функций в TestComplete – процесс сложный. Мы рассмотрим самый простой пример: функция **LockWorkStation** из файла **user32.dll,** которая блокирует компьютер.

В общем случае, для того, чтобы вызвать функцию из DLL, необходимо выполнить следующие действия:

- Определить тип DLL
- Описать параметры типов данных
- Описать функцию, которую мы вызываем
- Заполнить структуры данных передаваемых параметров
- Загрузить DLL в память

- Вызвать описанную функцию

В нашем примере функция **LockWorkStation** не принимает никаких параметров, что существенно упрощает задачу. Ниже показан пример вызова этой DLL-функции.

```
function TestDLL()
{
  // определяем новый тип DLL
  user32dll = DLL.DefineDLL("MyDLL");

  // описываем вызываемую функцию
  // последний параметр - возвращаемое значение, должен быть указан
обязательно
  // в нашем случае функция не возвращает никакого значения, поэтому
  // указан тип vt_void
  proc = user32dll.DefineProc("LockWorkStation", vt_void);

  // загружаем DLL в память, связывая имя DLL-файла с созданным ранее типом
DLL
  lib = DLL.Load("C:\\Windows\\System32\\USER32.DLL", "MyDLL");

  // вызываем функцию
  lib.LockWorkStation();
}
```

Подробнее о вызове функций из DLL можно прочитать в справочной системе TestComplete, раздел **How to Call a DLL Routine From Your Script.**

Вызов API-функций

Для вызова API-функций в TestComplete существует объект **Win32API**. С его помощью можно вызывать API-функции, общие для всех версий Windows начиная с версии 2000 и выше. Объект **Win32API** не поддерживает специфические функции для определенных платформ.

Кроме того, так как Microsoft OLE не поддерживает указатели, в TestComplete нет возможности вызывать API-функции, которые работают с указателями, принимая их в качестве параматров или возвращая указатели (например, ShellExecute).

Пример вызова API-функции:

```
function TestWinAPI()
{
  var res = Win32API.GetCommandLine();
  Log.Message(res);
}
```

Этот пример выведет в лог TestComplete-а значение «C:\Program Files\Automated QA\TestComplete 7\Bin\TestComplete.exe».

При вызове API-функций таким образом, обращение Win32API можно не указывать. Таким образом, следующий пример будет работать так же, как и предыдущий.

```
function TestWinAPI()
{
  var res = GetCommandLine();
  Log.Message(res);
```

```
}
```

11.5 Вызов функций из .NET сборок

Существует 2 способа вызова функций из .NET сборок (assemblies):

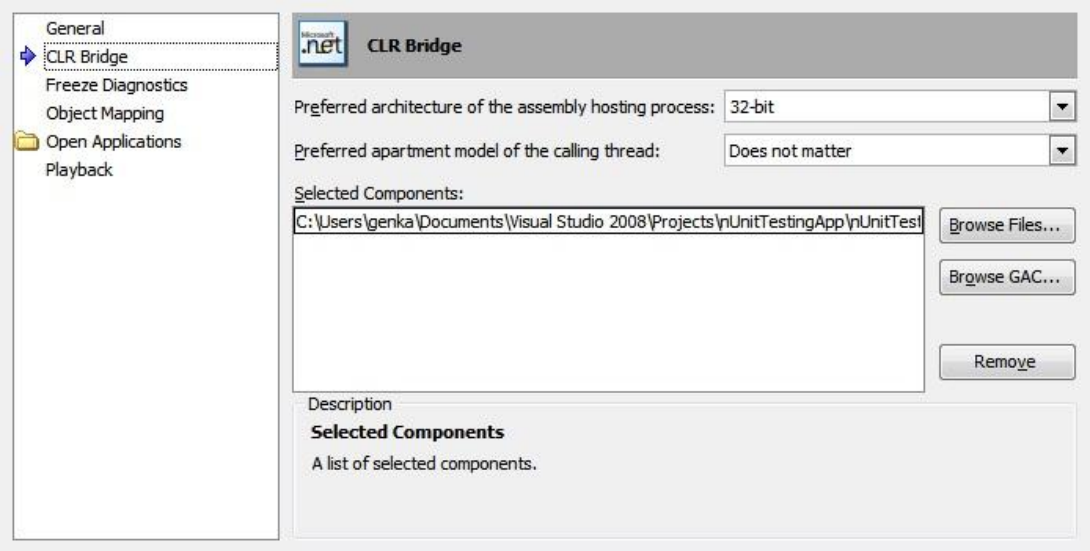
1. Используя объект dotNET
2. Используя домен приложения

Мы рассмотрим оба этих примера.

В качестве примера .NET-приложения мы возьмем приложение nUnitTestingApp, которое можно найти в архиве с проектом и которое мы создавали специально для главы 17 Модульное тестирование.

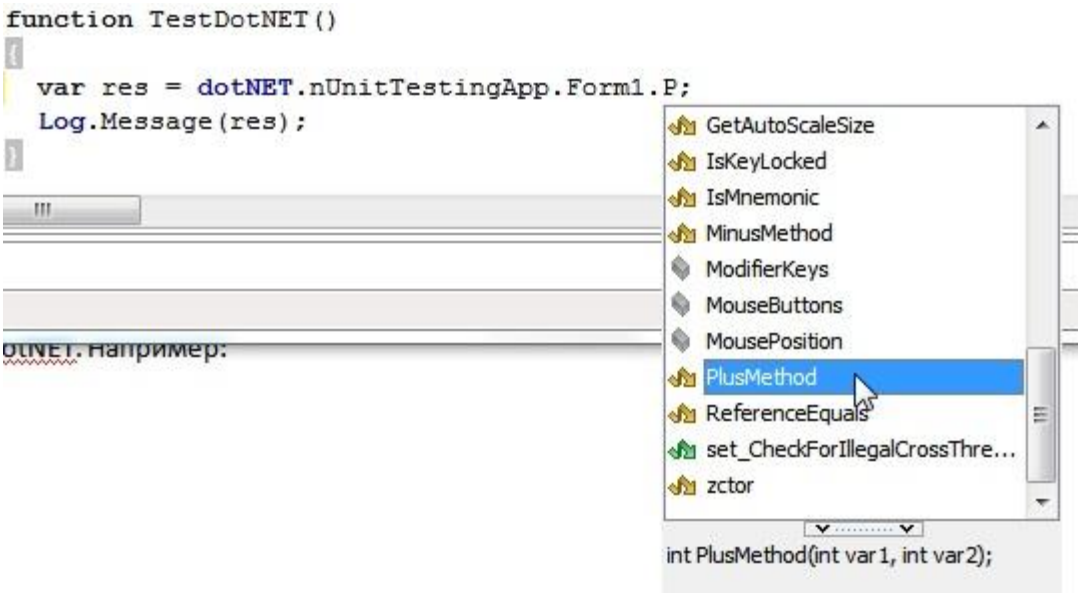
Использование объекта dotNET

Прежде, чем получить доступ к .NET методам из скриптов TestComplete, необходимо добавить эти методы в список **CLR Bridge.** Для этого щелкнем правой кнопкой мыши на имени проекта в Project Explorer и выберем пункт меню *Edit – Properties*, после чего выберем элемент **CLR Bridge.**



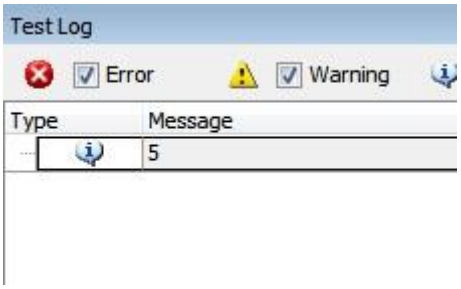
Здесь мы можем добавлять различные методы как из произвольных файлов (*Browse Files*), так и из глобального кэша сборок (*Browse GAC*).

Нажмем кнопку *Browse Files* и выберем файл nUnitTestingApp.exe. Теперь методы из нашего класса **Form1** можно видеть и вызывать непосредственно из скриптов TestComplete с помощью объекта **dotNET**.




```
function TestDotNET()
{
 var res = dotNET.nUnitTestingApp.Form1.PlusMethod(2, 3);
 Log.Message(res);
}
```

Результат работы данной функции:



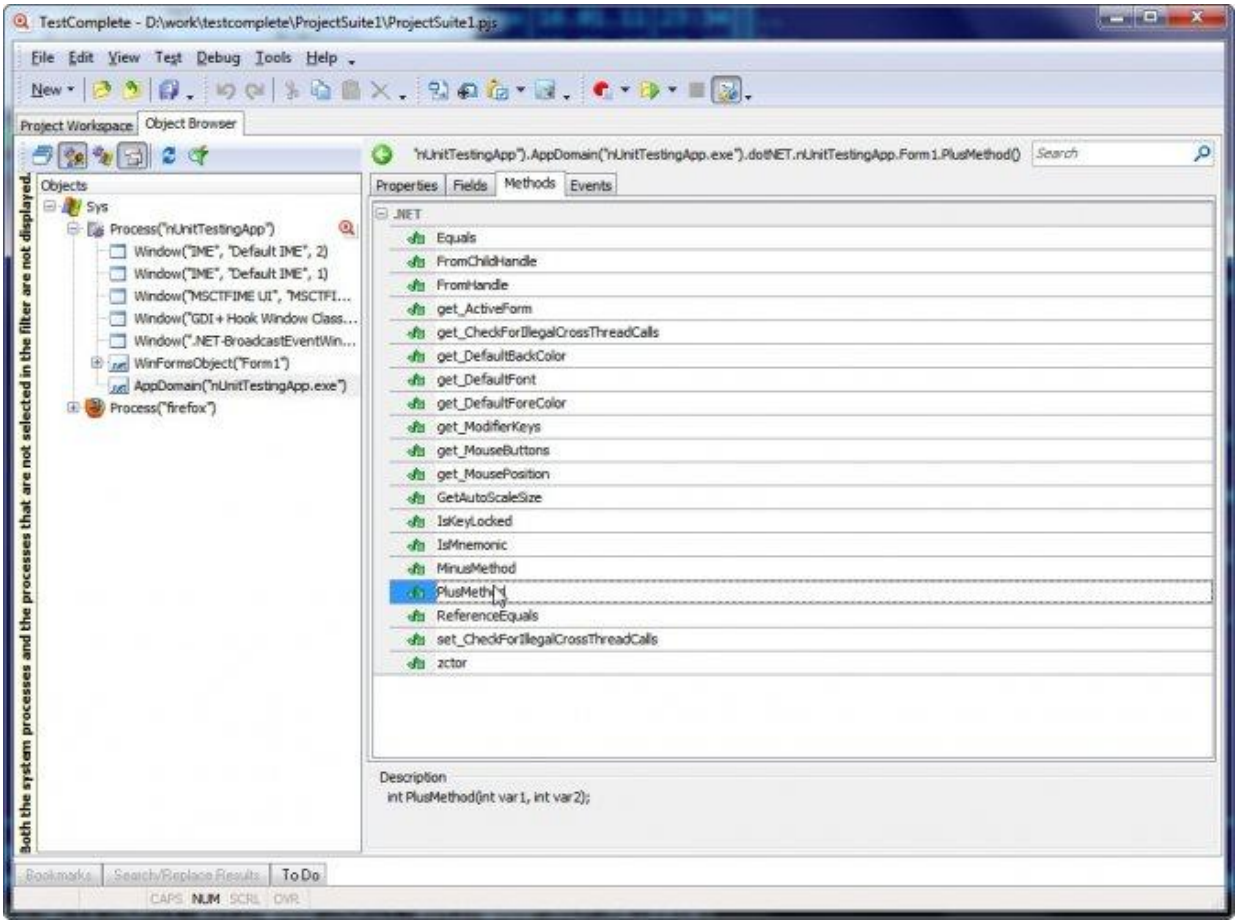
Этот способ позволяет вызывать статические методы. Если вам нужно вызвать нестатический метод, сначала необходимо создать объект соответствующего класса с помощью конструктора. Все конструкторы имеют имена вида ztor, ztor_1, ztor_2 и т.д.

Обратите внимание, что запускать само приложение нам не нужно. Также обратите внимание на параметры **Preferred architecture…** и **Preffered apartment model…** на странице CLR Bridge. Эти параметры необходимо установить правильно в зависимости от сборки. Подробнее об этих опциях можно прочитать в разделе справки **CLR Bridge Options.**

Также обратите внимание, что если в имени namespace используется точка (например, System.Collection), то точку необходимо заменить на знак подчеркивания (System_Collection).

Использование домена приложения

Любой запущенный .NET процесс имеет метод **AppDomain**, который позволяет обращаться к методам сборок без добавления их в список **CLR Bridge.**



Метод AppDomain содержит свойство dotNET, через которое можно обращаться как к свойствам и методам приложения, так и всех сборок, которые загружены в домен этого приложения. Например:

```
res = 
Sys.Process("nUnitTestingApp").AppDomain("nUnitTestingApp.exe").dotNET.nUnitTesting
App.Form1.MinusMethod(10,3);

 Log.Message(res);
```

Очевидным минусом этого подхода является то, что приложение должно быть запущено.

11.6 Remote Desktop, Virtual PC, VMware

Так как для запуска скриптов довольно часто использует удаленное подключение (Remote Desktop Connection) и виртуальные машины (Virtual PC и VMware), мы бы хотели в этой главе рассмотреть некоторые особенности их использования.

Основной компьютер

НЕ БЛОКИРУЙТЕ компьютер и НЕ ВЫЛОГИНИВАЙТЕСЬ из текущей сессии, когда работают скрипты TestComplete! При блокировке компьютера Windows перестает отрисовывать элементы пользовательского интерфейса (GUI), с которым взаимодействует TestComplete.

Это правило не касается типов тестирования, которые не связаны с GUI (например, HTTP Load Testing).

Remote Desktop

Если вы запускаете скрипты на удаленном компьютере, к которому подключаетесь через remote Desktop:

- НЕ ЗАКРЫВАЙТЕ окно Remote Desktop на основном компьютере!
- НЕ СВОРАЧИВАЙТЕ окно Remote Desktop на основном компьютере!
- НЕ БЛОКИРУЙТЕ основной компьютер!

Когда вы закрываете или сворачиваете окно Remote Desktop, Windows на удаленном компьютере перестаёт отрисовывать GUI-элементы, с которыми работает TestComplete.

Это касается только тестов, которые взаимодействуют с GUI приложения (окна, элементы управления и т.п.).

Для других типов тестирования (например, HTTP Load Testing), когда TestComplete не взаимодействует с пользовательским интерфейсом, эти два правила необязательны.

Для того, чтобы иметь возможность работать на основном компьютере, параллельно выполняя тесты на другом, просто уменьшите размер окна Remote Desktop и спокойно работайте с другими приложениями, переместив окно Remote Desktop на задний план.

Из этого правила есть одно интересное исключение. Если, допустим, вы подсоединяетесь с помощью Remote Desktop с основного компьютера к компьютеру comp1, а затем с компьютера comp1 в удаленной сессии открываете еще одно подключение к comp2, и уже на comp2 запускаете TestComplete, то на основном компьютере можно закрыть окно Remote Desktop Connection.

Для того, чтобы узнать, работают ли скрипты на удаленной машине (в Remote сессии), можно воспользоваться свойством **Sys.OSInfo.RemoteSession.**

Виртуальные машины (Virtual PC, VMWare и пр.)

Если вы запускаете скрипты на виртуальной машине:

- НЕ СВОРАЧИВАЙТЕ окно программы виртуальной машины
- НЕ ОТКЛЮЧАЙТЕСЬ от сессии, где в данный момент работают скрипты

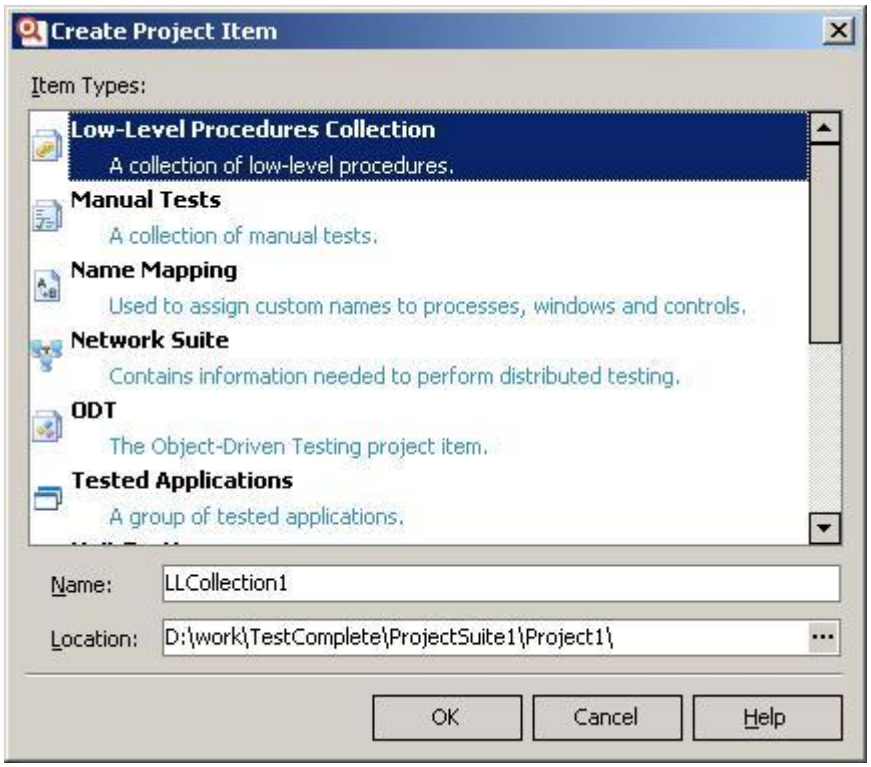
Результат будет такой же, как и в уже рассмотренной ситуации с Remote Desktop.

Вы можете, однако, переключаться между закладками VMWare, если у вас открыто несколько виртуальных машин.

Чтобы узнать, работают ли скрипты на виртуальных машинах, воспользуйтесь свойствами **Sys.OSInfo.VirtualPC** и **Sys.OSInfo.VMWare**.

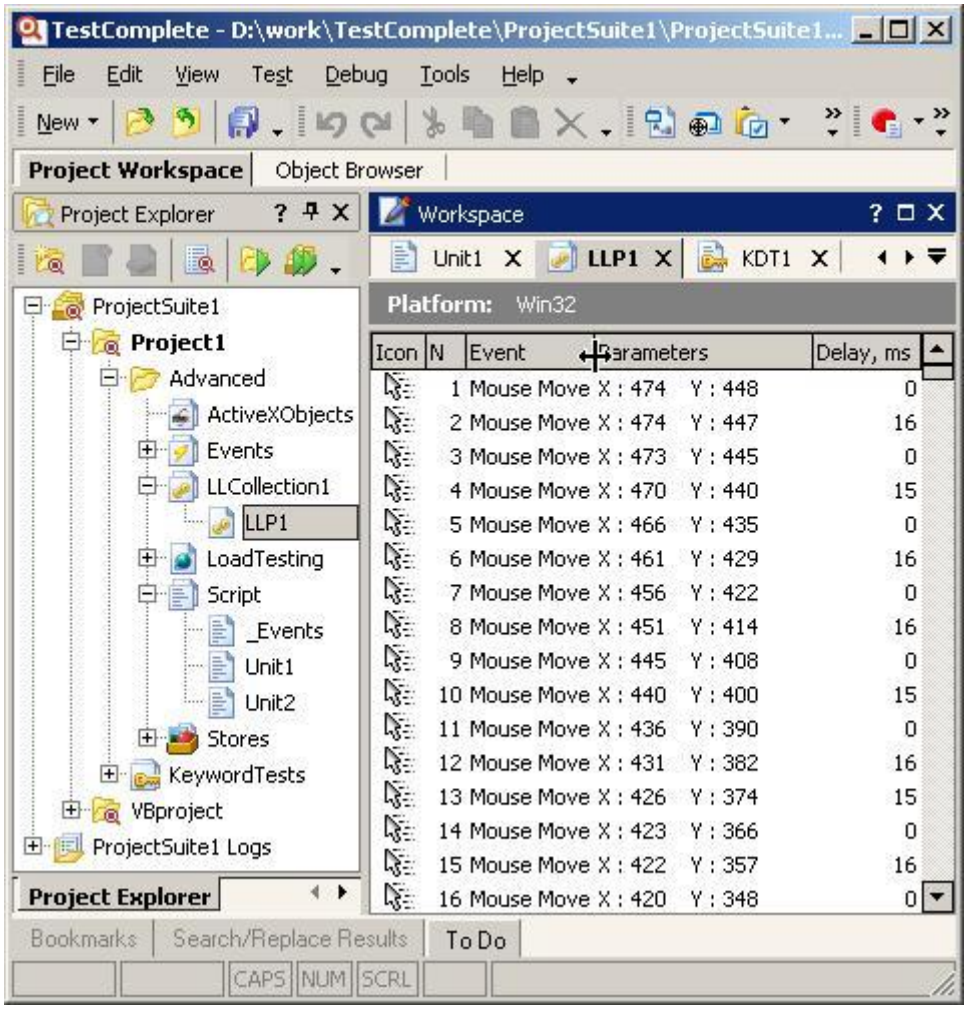
11.7 Использование низкоуровневых процедур

По умолчанию TestComplete записывает действия, которые интуитивно понятны (например, выбор элемента из выпадающего списка, нажатие на кнопку и т.п.). Однако иногда возникает необходимость в так называемых низкоуровневых процедурах, когда действия скрипта определяются не взаимодействием с элементами управления приложения, а простыми действиями, например «зажать левую кнопку мыши, переместить указатель мыши вверх на несколько пикселей, отпустить левую кнопку мыши, нажать ее снова, перетащить указатель на несколько пикселей влево, отпустить кнопку мыши». Такие действия обычно бывают нужны в графических приложениях (например, MS Paint), где необходимопротестировать рисование различных фигур.
В TestComplete для подобных действий существует объект **LLPlayer**. Чтобы получить доступ к низкоуровневым процедурам, необходимо сначала добавить в проект элемент **Low Level Procedures Collection.** Для этого надо щелкнуть правой кнопкой мыши на имени проекта, выберите пункт меню *Add – New Item* и в открывшемся диалоговом окне выберите элемент *Low-Level Procedures Collection.*

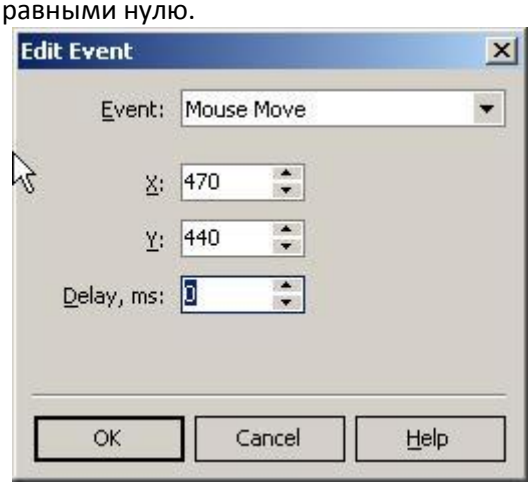


Теперь во время записи скрипта можно нажать кнопку *Start Recording a Low-Level Procedure* и TestComplete будет записывать низкоуровневую процедуру.

После того, как запись остановлена, в проекте появится записанная низкоуровневая процедура, которую можно найти в проекте в разделе *LLCollection1*.



Двойной щелчок по какой-то строке открывает диалоговое окошко для редактирования действия. Например, записанные задержки (колонка Delay) обычно не нужны и их можно установить равными нулю.



В следующем примере показан пример запуска записанной низкоуровневой процедуры из тестового скрипта.

```
function Test3()
{
  LLCollection1.LLP1.Execute();
}
```

Обратите внимание, что TestComplete может использовать записанные координаты относительно экрана или относительно какого-то конкретного окна (так называемые *Screen Related u Window Related* координаты). В данном случае мы использовали Screen Related координаты, т.е. смещение

курсора мыши отсчитывается от левого верхнего угла экрана. Чтобы запустить ту же самую процедуру относительно какого-то конкретного окна, достаточно передать это окно в качестве параметра методу Execute:

```
function Test3()
{
  var wnd = Sys.Process("mspaint").Window("MSPaintApp", "untitled -
Paint", 1);
  LLCollection1.LLP1.Execute(wnd);
}
```

Если вы не записываете тестовые скрипты с помощью инструментов записи TestComplete, а пишете их вручную, вы можете использовать объект **LLPlayer** для выполнения низкоуровневых процедур. Иногда встречаются ситуации, когда обычные методы (Click, Drag и т.п.) по какой-то причине не работают в тестируемом приложении, тогда необходимо воспользоваться объектом LLPlayer. Другой пример использования **LLPlayer** – работа с несколькими окнами (например, несколько дочерних окон MDI-приложения).

Ниже показан пример нажатия на кнопку «4» в окне Калькулятора, используя методы объекта LLPlayer.

```
function TestLLPlayer2()
{
   var wCalc = Sys.Process("calc").Window("SciCalc", "Calculator Plus", 1);
   wCalc.Activate();
   wCalc.HoverMouse(1, 1);

   var iX, iY;
   iX = wCalc.ScreenLeft;
   iY = wCalc.ScreenTop;
   LLPlayer.MouseMove(iX + 260, iY + 188, 0);
   LLPlayer.MouseDown(MK_LBUTTON, iX + 260, iY + 188, 0);
   LLPlayer.MouseUp(MK_LBUTTON, iX + 260, iY + 188, 0);

}
```

Обратите внимание, насколько усложняется при этом код скрипта – все использованные параметры являются обязательными. Однако, как уже было сказано, в некоторых случаях без этих возможностей не обойтись (например, тестирование графических приложений типа MS Paint).

11.8 Перехват событий

Событие (Event) в TestComplete – это действие, происходящее во время работы скриптов. Каждый раз, когда стартует и завершается работа скриптов, когда в лог отправляется сообщение (ошибка, предупреждение и т.п.), когда отправляется запрос нагрузочного теста и когда получен ответ, - во всех этих и в других случаях срабатывают события TestComplete.

По умолчанию эти события никак не обрабатываются, однако иногда могут возникать задачи, в которых потребовалась бы возможность управлять такими событиями.

Приведем 2 примера.

Первый. Предположим, что в процессе работы с тестируемым приложением произошло отсоединение от базы данных и в приложении выскочило сообщение, предупреждающее об этом. TestComplete в этом случае выдаст сообщение об ошибке "Unexpected window", и попытается закрыть окно одним из предопределенных способов (нажатие кнопки OK, нажатие клавиши Esc и т.п.). Даже если ему удастся закрыть окно, проблему с

отключением базы данных это не решит, поэтому было бы неплохо отловить подобную ситуацию и перед закрытием окна восстановить подключение к базе. Это возможно сделать с помощью события OnUnexpectedWindow.

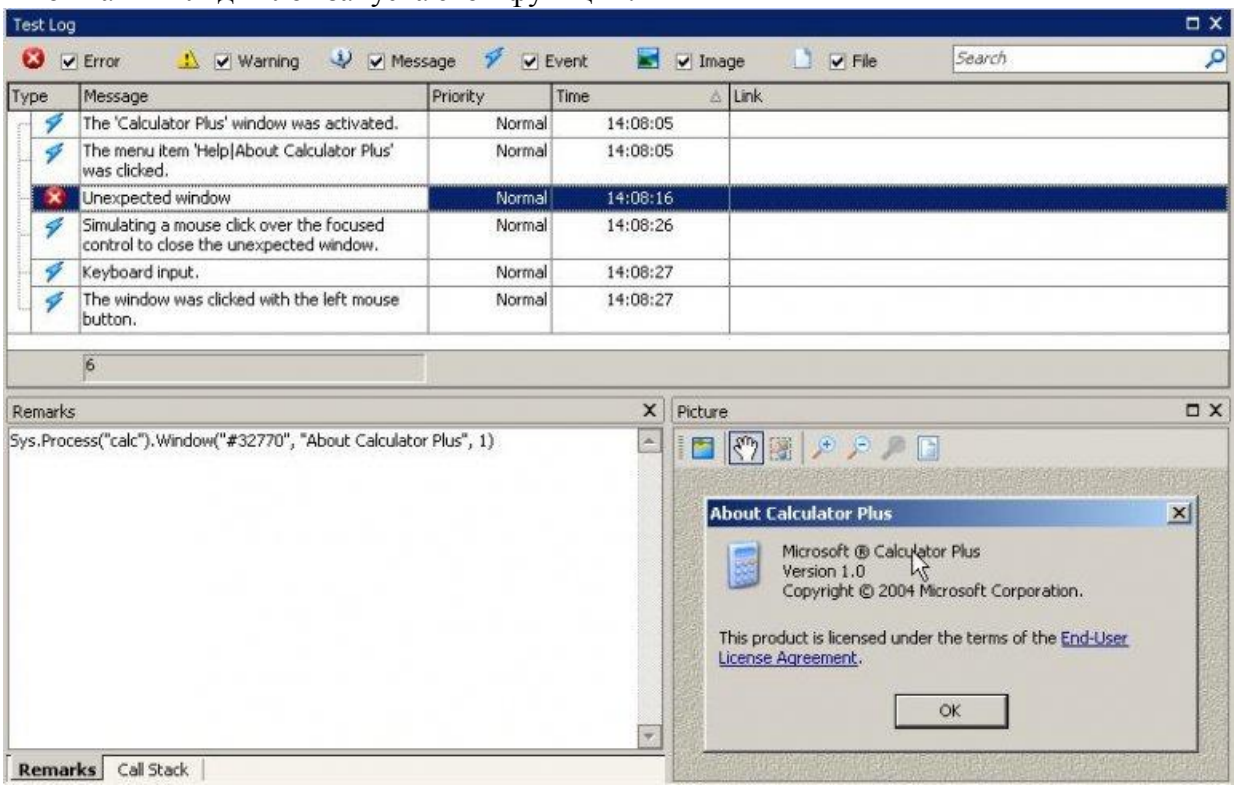
Второй. Это реальная ситуация, время от времени возникающая в TestComplete. При работе с некоторыми нестандартными элементами управления (в частности с элементами управления Infragistics), иногда TestComplete генерировал ошибку "Improper command". Это случалось потому, что при обработке действий пользователя эти контролы, как кажется TestComplete-у, перестают отвечать на запросы, хотя на самом деле они прекрасно работают. В результате после прогона скриптов мы имеем десяток ошибок "Improper command", хотя все скрипты отработали правильно. Для решения подобной проблемы мы можем перехватывать событие OnLogError и проверять текст ошибки. Если это наш "Improper command" – то мы попросту игнорируем такую ошибку, в противном случае ошибка заносится в лог как обычно.

В качестве примера перехвата события OnEnexpectedWindow рассмотрим пример с Калькулятором. Мы откроем окно About Calculator, а затем попытаемся работать с главным окном Калькулятора, что вызовет ошибку Unexpected window. В обработчике этой ошибки мы напишем проверку, которая будет проверять, является ли мешающее нам окно окном About и в случае, если это так, будем его просто закрывать и помещать сообщение в лог о том, что окно закрылось.

Вот как выглядит изначально наш пример:

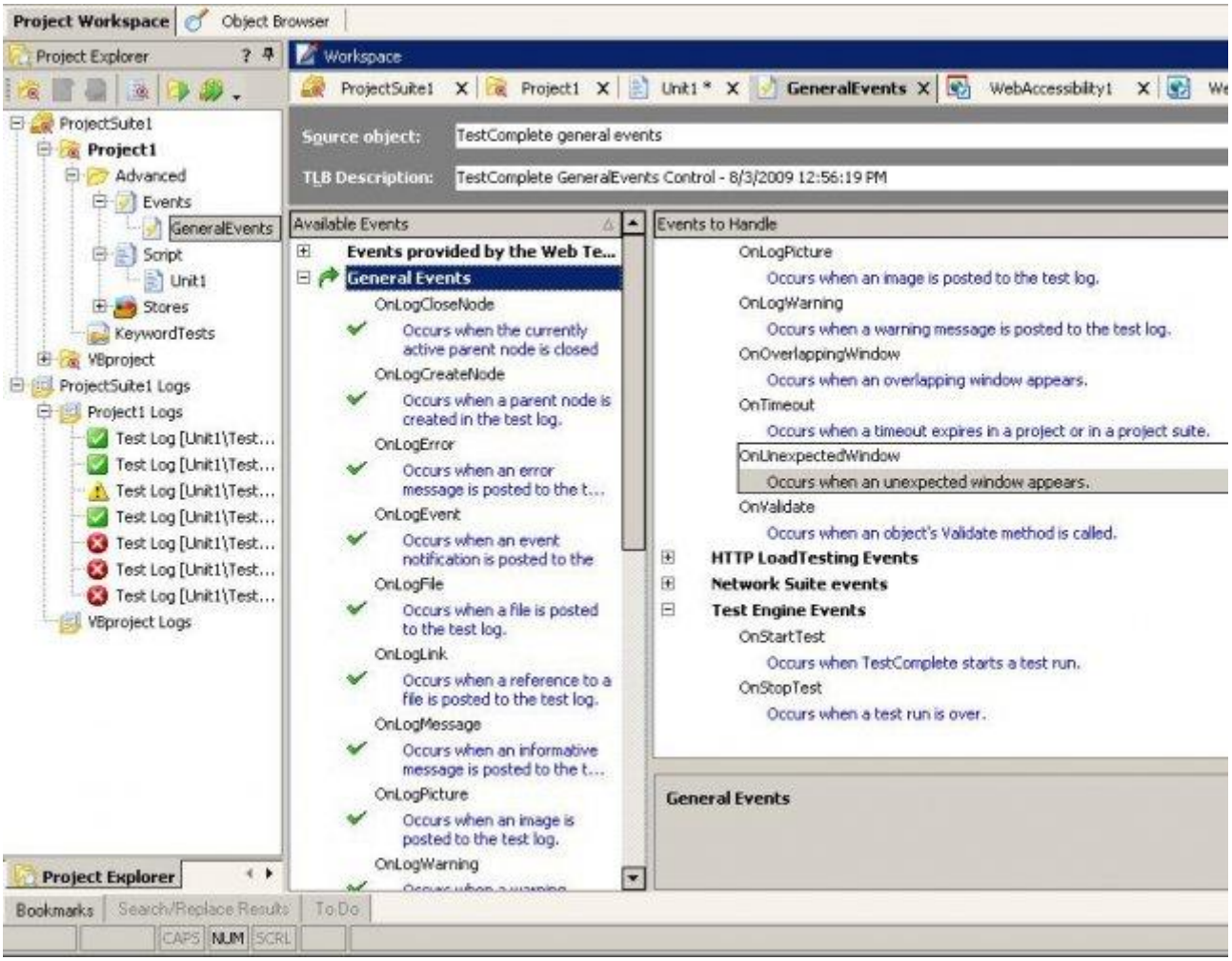
```
function TestUnexpectedWindow()
{
  var  wnd;
  wnd = Sys.Process("calc").Window("SciCalc", "Calculator Plus");
  wnd.Activate();

  wnd.MainMenu.Click("Help|About Calculator Plus");
  wnd.Keys("123");
  wnd.Window("Button", "+").Click();
}
```

И вот как выглядит лог запуска этой функции:

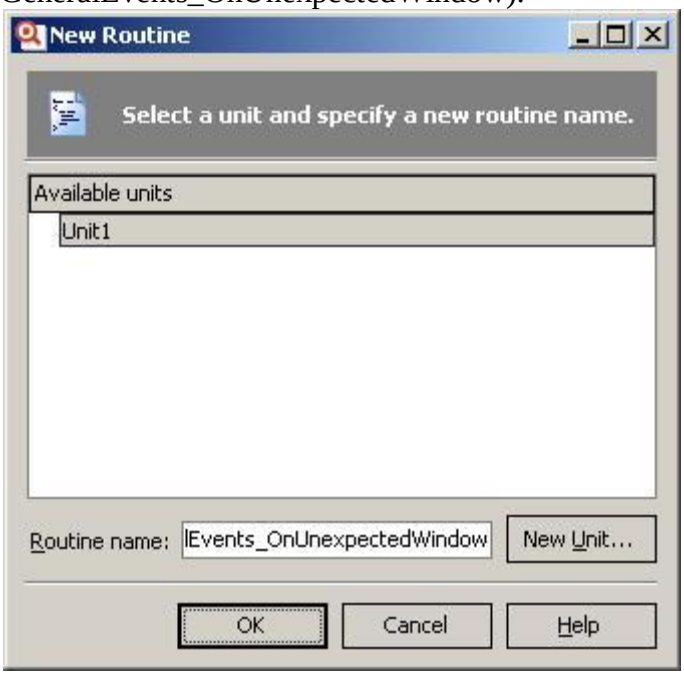


TestComplete сгенерировал сообщение об ошибке, поместил в лог скриншот мешающего окна и его полное имя, а затем закрыл окно, щелкнув мышью по сфокусированному элементу управления (в данном случае это кнопка OK). После этого TestComplete продолжил выполнение скрипта, однако в логе появилась ошибка, от которой нам хочется избавиться.

Теперь создадим обработчик события OnUnexpectedWindow. Для этого в дереве проекта выберем пункт Events – GeneralEvents, а затем в правой части окна TestComplete дважды щелкнем на пункте OnUnexpectedWindow, как показано на скриншоте ниже.



При этом откроется окно New Routine. Здесь мы должны выбрать модуль (Unit), в котором хотим хранить наш обработчик события, а также дать ему имя (по умолчанию GeneralEvents_OnUnexpectedWindow).



Обратите внимание! Если вы не выполните все вышеперечисленные действия, а просто создадите функцию с именем GeneralEvents_OnUnexpectedWindow, - это не создаст обработчик события! Чтобы событие перехватывалось, необходимо связать какую-то функцию (новую или существующую) с этим событием, что мы сейчас и делаем.

Мы рекомендуем хранить все обработчики событий в отдельном модуле, например _Events (подробнее о работе с несколькими модулями в проекте можно прочитать в главе 3.10 Работа с несколькими модулями).

Так как у нас нет сейчас отдельного модуля, мы его создаем с помощью кнопки New Unit, а затем жмем ОК в окне New Routine. При этом в модуле _Events у нас появляется функция-обработчик события OnUnexpectedWindow:

```
function GeneralEvents_OnUnexpectedWindow(Sender, Window, LogParams)
{

}
```

В этой функции 3 параметра:

- **Sender** (отправитель) – в нашем случае это GeneralEvents
- **Window** – окно, которое мешает TestComplete-у
- **LogParams** – текущие параметры лога (шрифт, цвет и т.п.)

Теперь нам необходимо внести такие изменения, которые позволят не выводить сообщение об ошибке в случае окна About, а просто закрывать его, а в остальных случаях действовать как обычно. Для этого мы будем проверять заголовок окна About, а также воспользуемся свойством LogParams.Locked для блокировки отправки сообщения.

```
function GeneralEvents_OnUnexpectedWindow(Sender, Window, LogParams)
{
  if(Window.WndCaption == "About Calculator Plus")
  {
    LogParams.Locked = true;
    Window.Close();
    Log.Message("Окно About закрыто");
  }
}
```

Результат работы скрипта:

Type	Message	Priority
	The 'Calculator Plus' window was activated.	Normal
	The menu item 'Help|About Calculator Plus' was clicked.	Normal
	The 'About Calculator Plus' window was closed.	Normal
	Окно About закрыто	Normal
	Keyboard input.	Normal
	The window was clicked with the left mouse button.	Normal

Аналогичным образом создаются обработчики других событий, потому мы не будем их рассматривать подробно, а лишь перечислим некоторые из них и дадим общие рекомендации.

В группе **Events provided by the We-Testing plug-in** находятся события, специфичные для тестирования веб-приложений:

1. *OnWebBeforeNavigate* – возникает перед тем, как браузер осуществляет переход на новую станицу
2. *OnWebDownloadFinished* – возникает когда страница открылась либо ее открытие было прервано
3. *OnWebDownloadStarted* – возникает сразу после того, как началась загрузка страницы
4. *OnWebPageDownloaded* – возникает после того, как страница или фрейм загрузились
5. *OnWebQuit* – возникает перед тем, как браузер закрывается

В группе **General Events** находятся специфичные для нагрузочного тестирования события: *OnLoadTestingRequest и OnLoadTestingResponse*, возникают, соответственно, когда TestComplete отправляет HTTP запрос и получает ответ от сервера.

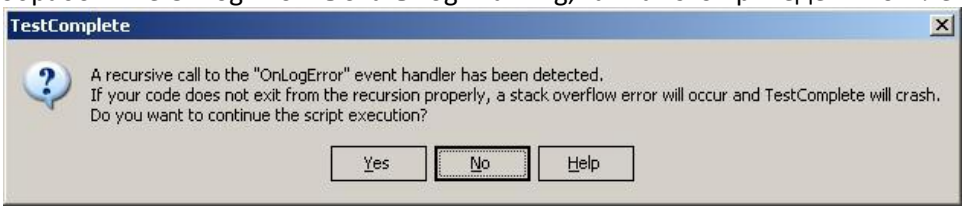
В группе **Network Suite events** находятся специфичные для распределенного тестирования события.

Test Engine Events – здесь находятся только два события, *OnStartTest и OnStopTest*, которые возникают при запуске и остановке работы скриптов.

Группа **General Events** самая большая, здесь содержатся события, связанные с логом TestComplete: *OnLogCloseNode*, *OnLogCreateNode*, *OnLogError*, *OnLogWarning* и т.п.. Здесь же находятся события, связанные с появлением неожиданных для TestComplete окон (*OnOverlappingWindow и OnUnexpectedWindow*), *OnTimeout* – возникает когда истекает время, отведенное для работы скриптов.

И несколько советов по работе с перехватом событий

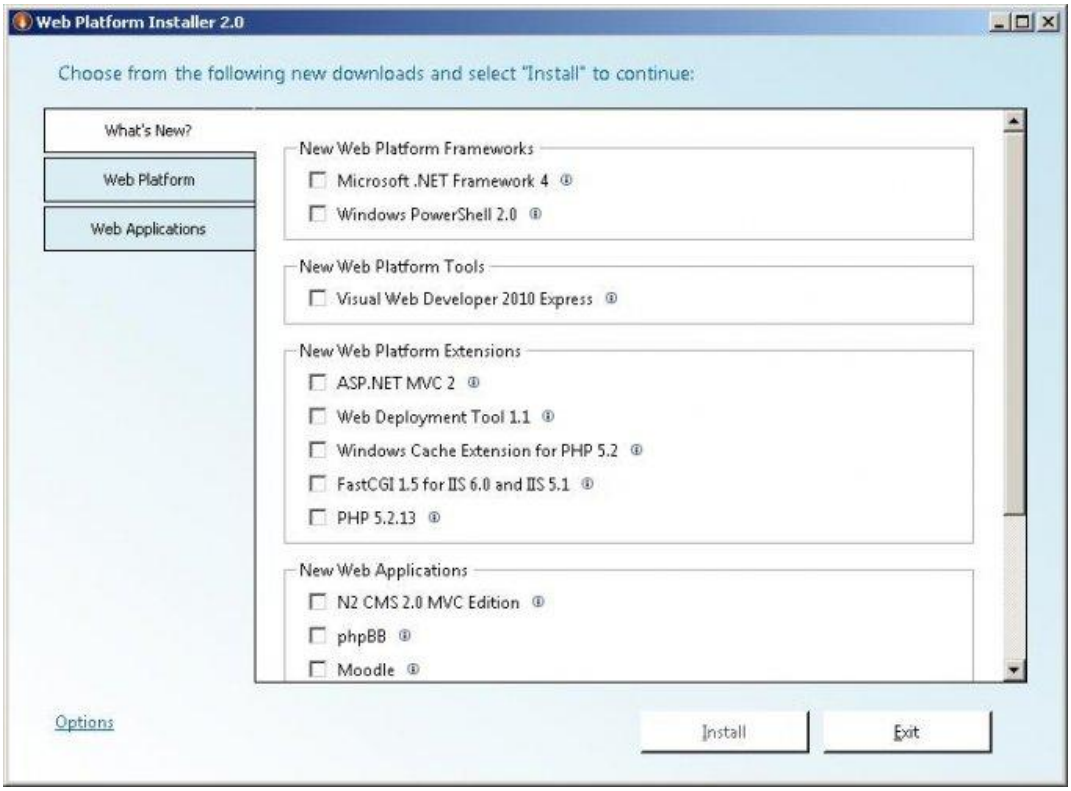
1. *Перехват событий – очень полезный и мощный инструмент TestComplete, однако с ним надо быть очень осторожным.* Например, в использованном примере достаточно вынести строку LogParams.Locked = true за пределы условия if и вы больше не увидите ни одного сообщения о неизвестном окне, хотя они будут появляться и TestComplete будет пытаться их закрыть. Будьте внимательны, когда используете перехват событий
2. *Не зацикливайте события друг на друга и на самих себя.* То есть, не пытайтесь внутри обработчика события OnLogError вызвать метод Log.Error и т.п., а также не пытайтесь, например, вызвать Log.Error внутри обработчика OnLogWarning и одновременно с этим в обработчике OnLogError вызвать Log.Warning, так как это приведет к появлению ошибки



3. Помните, что *события, связанные с логом (OnLogError, OnLogWarning, OnLogEvent и т.д.)* возникают очень часто в ходе работы скриптов, а значит *не стоит перегружать их большим количеством кода*, так как это может существенно увеличить время работы тестовых скриптов

11.9 Ассоциации объектов (object mapping)

В некоторых приложениях по разным причинам разработчики используют собственноручно написанные элементы управления, которые, однако, по функциональности аналогичным стандартным контролам. Обычно это используется для расширения возможностей этого элемента управления и новый элемент делается на базе существующего, однако TestComplete не может распознать его и считает просто объектом, с которым может работать только кликая мышкой и вводя текст при помощи метода Keys. Чтобы решить подобные проблемы, в TestComplete существует возможность «связать» неизвестный TestComplete-у элемент управления с уже известным. Эта возможность называется **Object Mapping** (Ассоциации объектов). Для этого необходимо в настройках проекта в разделе **Object Mapping** выбрать существующий класс и добавить в список имен классов новое имя класса вашего элемента управления.
В качестве примера рассмотрим приложение Web Platform Installer. Внешне оно выглядит так:

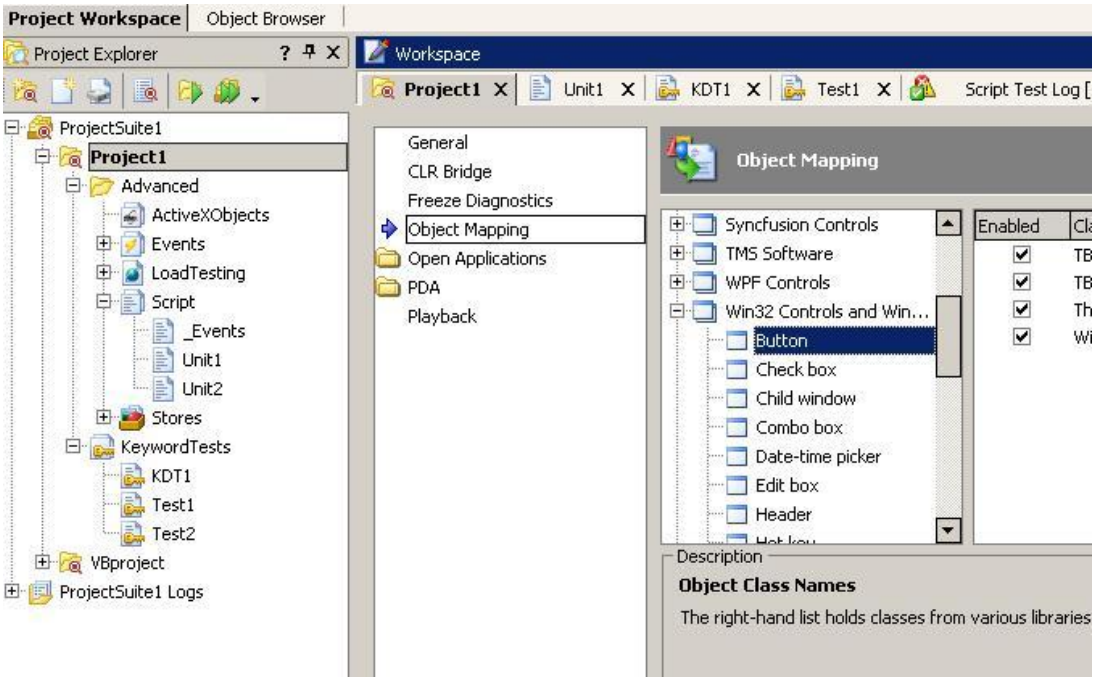


В левой нижней части окна вы видите ссылку **Options**, которая на самом деле является статическим текстом с определенными свойствами, однако работает фактически как кнопка, так как при нажатии на нее открывается новое окно. Если в TestComplete начать записывать действия, нажать на эту ссылку и остановить затем запись, то мы получим следующий записанный скрипт:

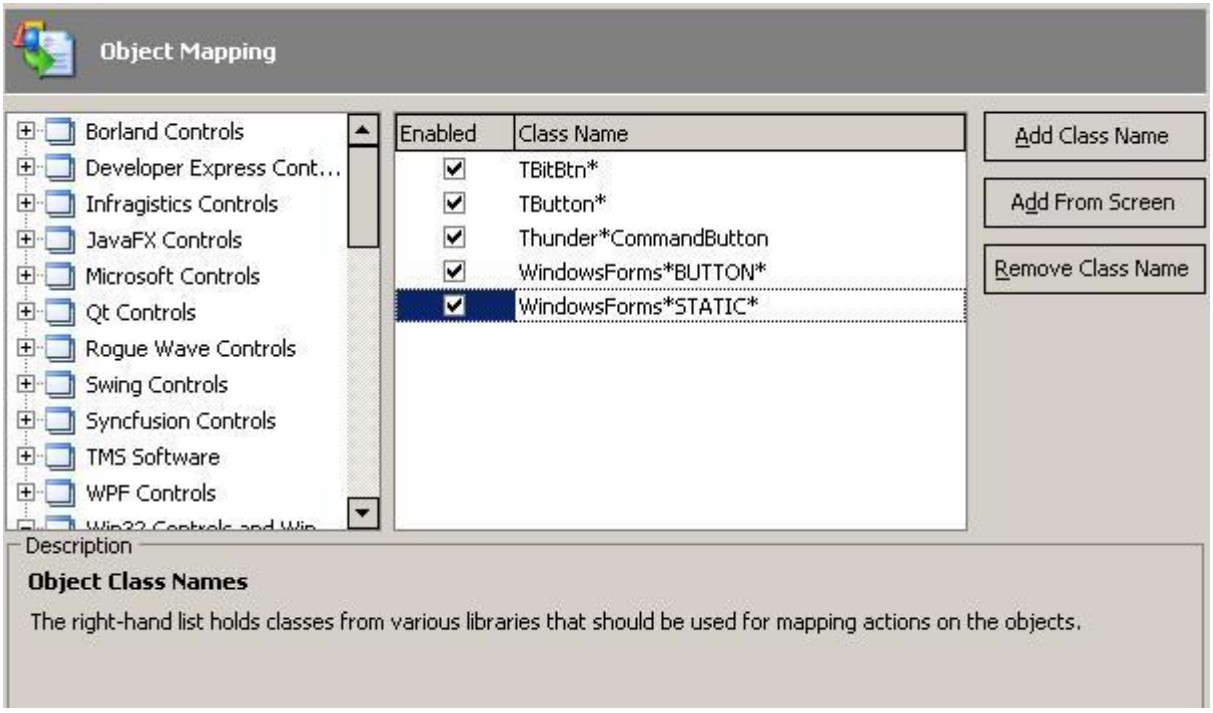
```
function Test1()
{

Sys.Process("WebPlatformInstaller").WinFormsObject("ShellForm").WinFormsObjec
t("_changeSettingsLinkLabel").Click(36, 5);
}
```

Так как TestComplete не знает, что это за объект, он записывает нажатие на ссылку с помощью метода Click с конкретными координатами.
Теперь в TestComplete щелкнем правой кнопкой по имени проекта и выберем пункт меню *Edit – Properties*, затем выберем в списке раздел *Object Mapping*, а в списке классов –

элемент *Win32 Controls and Windows – Button*.



Теперь, если вы знаете имя нового класса (например, предварительно скопировали значение свойства WndClass в Object Browser-е), можете нажать кнопку *Add Class Name* и в добавленном элементе списка вписать имя своего класса. Или же можете просто нажать на кнопку *Add From Screen* и затем выделить на экране нужный объект с помощью *Finder Tool*. В некоторых случаях (как в нашем примере) имя класса строится динамически (например, **WindowsForms10.STATIC.app.0.33c0d9d**), поэтому лучше динамические части заменить на символ групповой замены *, чтобы в итоге получилось **WindowsForms*STATIC***.



Если теперь снова перейти в режим записи и нажать на ту же ссылку Options, мы получим следующий результат:

```
function Test2()
{
```

```
Sys.Process("WebPlatformInstaller").WinFormsObject("ShellForm").WinFormsObject("_chang
eSettingsLinkLabel").ClickButton();
}
```

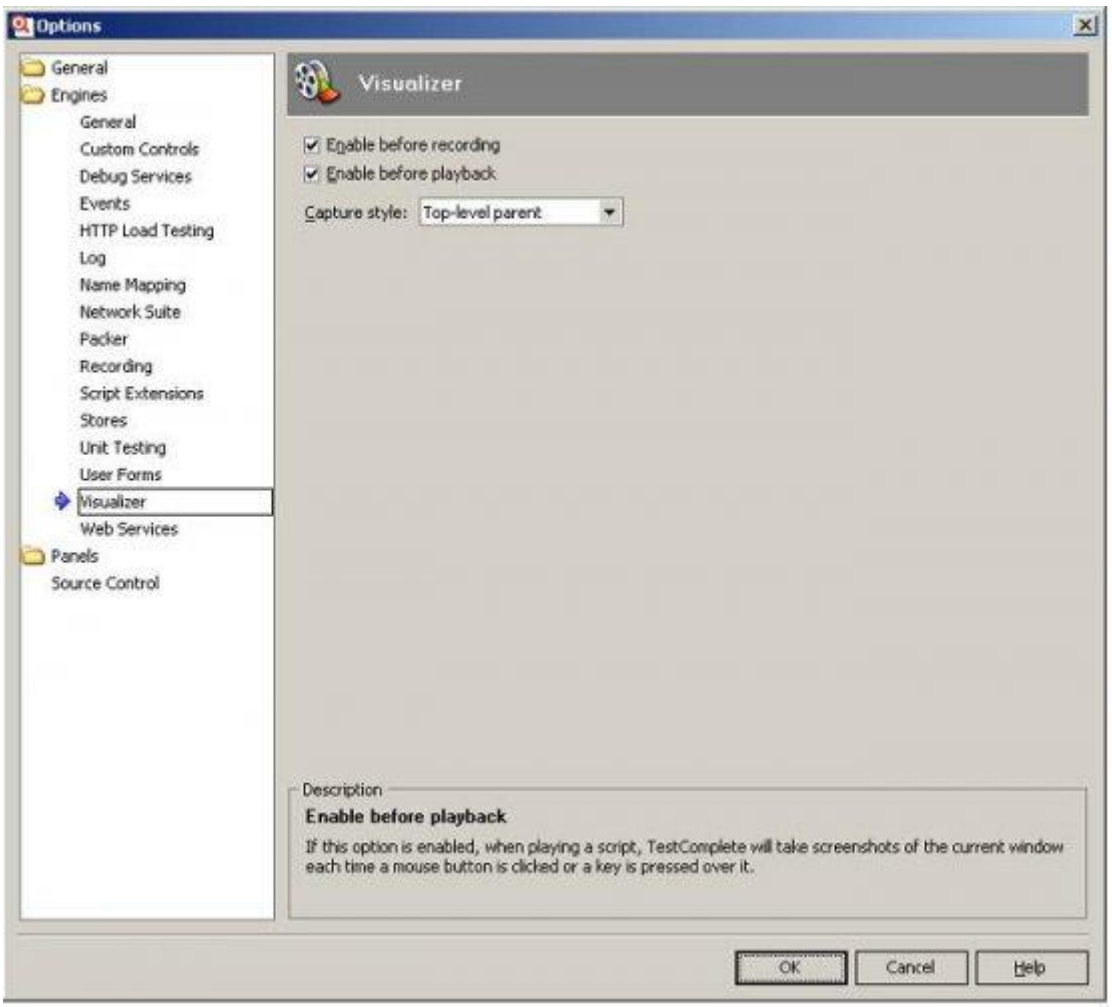
Как видите, метод *Click* с координатами заменился на *ClickButton* без координат, так как теперь TestComplete считает, что работает с обычной кнопкой.
Сразу оговоримся, что приведенный выше пример не совсем удачный с точки зрения маппирования, так как фактически мы заставляем TestComplete работать со статическим текстом как с кнопкой, однако он хорошо показывает основы работы с Object Mapping. Разработчики TestComplete уже предусмотрели ассоциации со множеством известных классов, в чем можно убедиться, просмотрев настройки разных существующих классов, поэтому подобные ситуации у пользователей должны возникать нечасто.

11.10 Использование Визуализатора

Визуализатор (Visualizer) – это простое и удобное средство для сопоставления кода скриптов с соответствующими экранными объектами. С помощью Визуализатора вы можете всегда посмотреть на скриншот окна в тот момент, когда выполнялось то или иное действие (нажатие кнопки, ввод текста, в общем любого действия, связанного с работой с экранными объектами). При этом объект, с которым работал скрипт в определенный момент, будет обведен на скриншоте, что упрощает поиск элементов управления на снимке экрана.

Чтобы включить Визуализатор, необходимо открыть настройки *TestComplete – Tools – Option – Engines – Visualizer* и в открывшейся панели Visualizer включить чекбоксы *Enable before recording и Enable before playback*. Это заставит TestComplete делать снимки экрана как при записи, так и при воспроизведении (но не каждый раз, а только лишь в том случае, если новый объект отличается от предыдущего).

В выпадающем списке можно выбрать, снимок чего именно должен производиться:



- **Window** – будет делаться снимок только элемента управления, с которым ведется работа (например, кнопка или текстовое поле)
- **Top-level** parent – окно-родитель, т.е. диалоговое или главное окно — потомок процесса, — в котором находится элемент управления
- **Desktop** – будет делаться снимок всего экрана
- **User-defined region** — регион, указанный пользователем

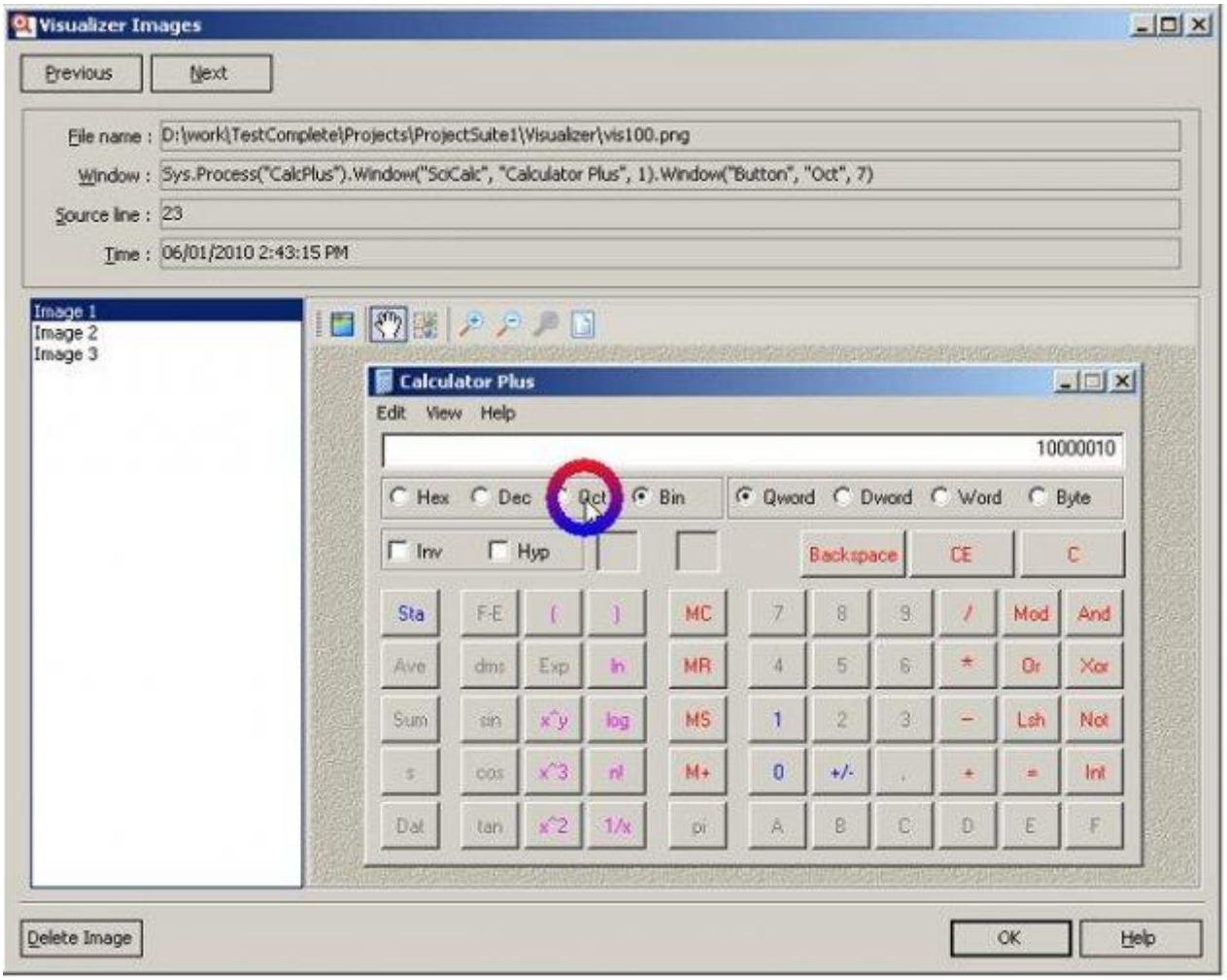
По умолчанию включена опция Top-level parent, что рекомендуется для большинства случаев.

Теперь запишем простой пример скрипта, который щелкает по разным кнопкам в Калькуляторе и посмотрим, что из этого получится.

```
13  function Test1()
14  {
15    var  wndSciCalc;
16    wndSciCalc = Sys.Process("CalcPlus").Window("SciCalc", "
17    wndSciCalc.Activate();
      wndSciCalc.Window("Button", "5").ClickButton();
      wndSciCalc.Window("Button", "8").ClickButton();
      wndSciCalc.Window("Button", "+").ClickButton();
      wndSciCalc.Window("Button", "7").ClickButton();
      wndSciCalc.Window("Button", "=").ClickButton();
      wndSciCalc.Window("Button", "Hex").ClickButton();
      wndSciCalc.Window("Button", "Bin").ClickButton();
25  }
26
```

Как видите, после записи скрипта возле некоторых строк у нас появились пиктограммки, означающие, что для объекта, используемого в данной строке, есть ассоциированный скриншот.

Если навести курсор на эту пиктограммку, то появится список всех изображений, ассоциированных с этой строкой, а если щелкнуть по пиктограмме — появится окно Visualizer Images.



В списке слева отображаются все скриншоты, соответствующие этой строке кода. Это удобно в том случае, когда внешний вид элемента управления изменяется, всегда можно посмотреть, как он выглядел раньше и сравнить с текущим изображением.

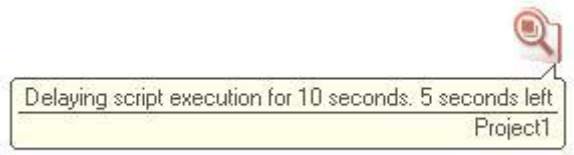
В верхней части окна можно посмотреть полное имя элемента управления, номер строки, с которой он ассоциирован и время, когда он был сохранен.

Единственное неудобство Визуализатора состоит в том, что скриншоты ассоциируются не с конкретной строкой кода, а с номером строки модуля, хотя при этом учитываются конкретные функции. Т.е. если вы вставите новую функцию перед приведенной функцией, то все скриншоты переместятся и будут находиться возле правильных строк. Однако если вы вставите новые строки внутри текущей функции, то скриншоты сместятся относительно строк, для которых они были сохранены.

Все скриншоты Визуализатора хранятся в папке <Project Suite>\Visualizer.

11.11 Работа с Индикатором

Индикатор – это небольшое информационное окошко, которое появляется в правом верхнем углу экрана во время работы скриптов и отображает текущие действия скрипта.



Для работы с Индикатором предназначен объект **Indicator** с несколькими свойствами и методами:

- Свойство *Text* – позволяет получить текст, в данный момент отображаемый на Индикаторе

Методы:

- *Hide/Show* – скрыть/отобразить Индикатор
- *Clear* – очистить текст в Индикаторе
- *PushText* – задает новый текст для Индикатора
- *PopText* – позволяет восстановить в Индикаторе текст, который был в нем до вызова метода PushText.

Ниже показан простой пример работы с Индикатором: функция Sleep, приостанавливающая выполнение скрипта на заданное количество секунд и помещающая в Индикатор информацию о том, на сколько приостановлено выполнение скрипта и сколько еще осталось ждать. Для запуска функции просто вызовите функцию Sleep с любым целочисленным параметром (например, 10) из другой функции.

Обратите внимание, что в последних версиях TestComplete метод BuiltIn.Delay считается устаревшим и вместо него необходимо использовать метод aqUtils.Delay!

```
function Sleep(iSeconds)
{
  i = iSeconds;
  while(i > 0)
```

```
  {
    BuiltIn.Delay(500);
    Indicator.PushText("Delaying script execution for " + iSeconds + " seconds. " + i + " seconds
left");
    BuiltIn.Delay(500);
    i -= 1;
  }
  Indicator.Clear();
}
```

12 Работа с графическими объектами

Прежде, чем начать эту главу, напомним, что любые операции с картинками в автоматизации тестирования нужно производить только в случае крайней необходимости (например, в случае тестирования графического редактора или когда нет никакой другой возможности работать с объектом), так как операции сравнения изображений:

- во-первых, производятся долго по сравнению с другими операциями
- во-вторых, требуют частого обновления тестовых скриптов

Это происходит потому, что обычно даже самые незначительные изменения в приложении, незаметные для пользователя, сразу же отражаются на результатах проверок, отчего в логах появляются ошибки, которые на самом деле ошибками приложения не являются, а являются лишь недоработкой или плохой реализацией проверок. Поэтому, прежде чем начинать использовать графические возможности TestComplete, убедитесь, что вы рассмотрели все прочие возможности и они действительно вам не подходят.

В отличие от многих других инструментов, предназначенных для автоматизации тестирования, TestComplete обладает огромным количеством встроенных средств для работы с изображениями. Ниже мы вкратце рассмотрим наиболее полезные из них.

Захват изображения и вывод его в лог

У любого видимого объекта в TestComplete есть метод Picture(), который позволяет захватить изображение элемента управления для дальнейшей работы с ним. Этот метод возвращает объект типа Picture. Например:

```
function TestImages()

{

  var wPaint = Sys.Process("mspaint").Window("MSPaintApp", "*");

  wPaint.Activate();



  var myPicture = wPaint.Picture();



}
```

В этом примере мы сохраняем изображение окна MS Paint в переменной myPicture.

Для веб-страниц также можно использовать метод PagePicture(), который позволяет сохранить в изображении всю страницу, даже если она выходит за видимые пределы экрана. При этом TestComplete сам прокрутит страницу и склеит куски изображения в одну картинку.

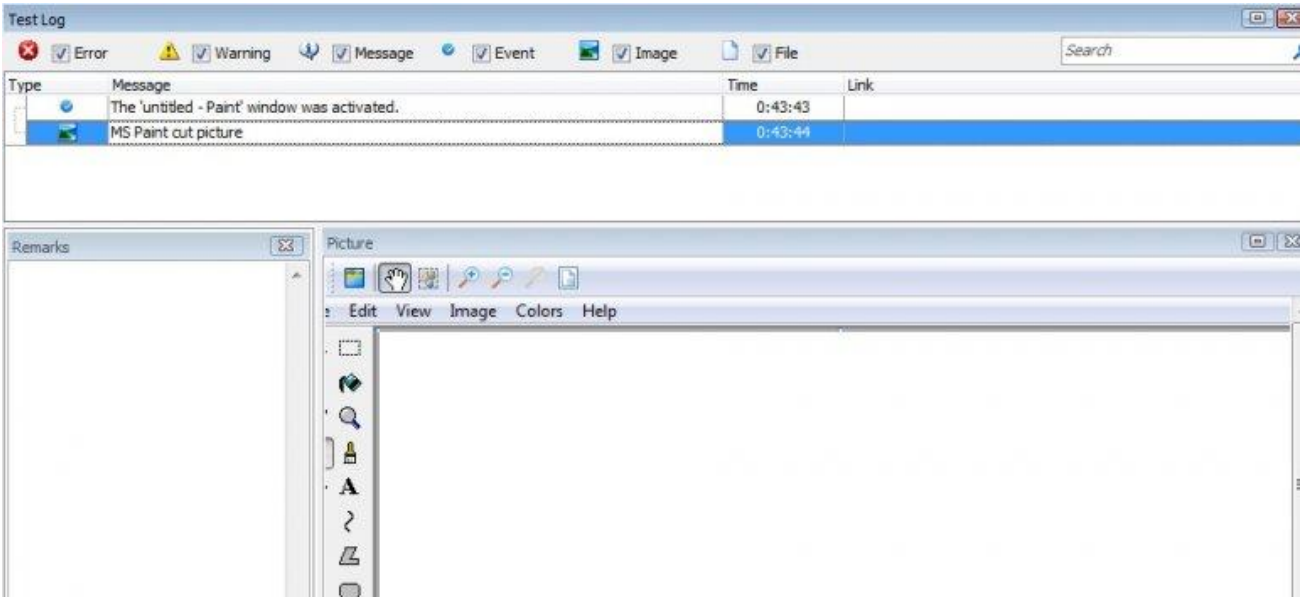
У метода Picture() есть несколько параметров, позволяющие захватить не всё изображение, а только его часть. При этом нам необходимо задать координаты начала (X и Y), а также ширину и высоту захватываемого изображения. Например, в следующем примере мы захватим не все окно Paint, а «обрежем» его на 10 пикселей с каждой стороны.

```
function TestImages()

{

  var wPaint = Sys.Process("mspaint").Window("MSPaintApp", "*");

  wPaint.Activate();



  var myPicture = wPaint.Picture(30, 30, wPaint.Width-60, wPaint.Height-60);

}
```

Для вывода изображения в лог предназначен медот Log.Picture, в который передаются собственно изображение и его описание. Например, если в предыдущий пример дописать следующую строку

```
Log.Picture(myPicture, "MS Paint cut picture");
```

То в результате мы получим следующий лог:

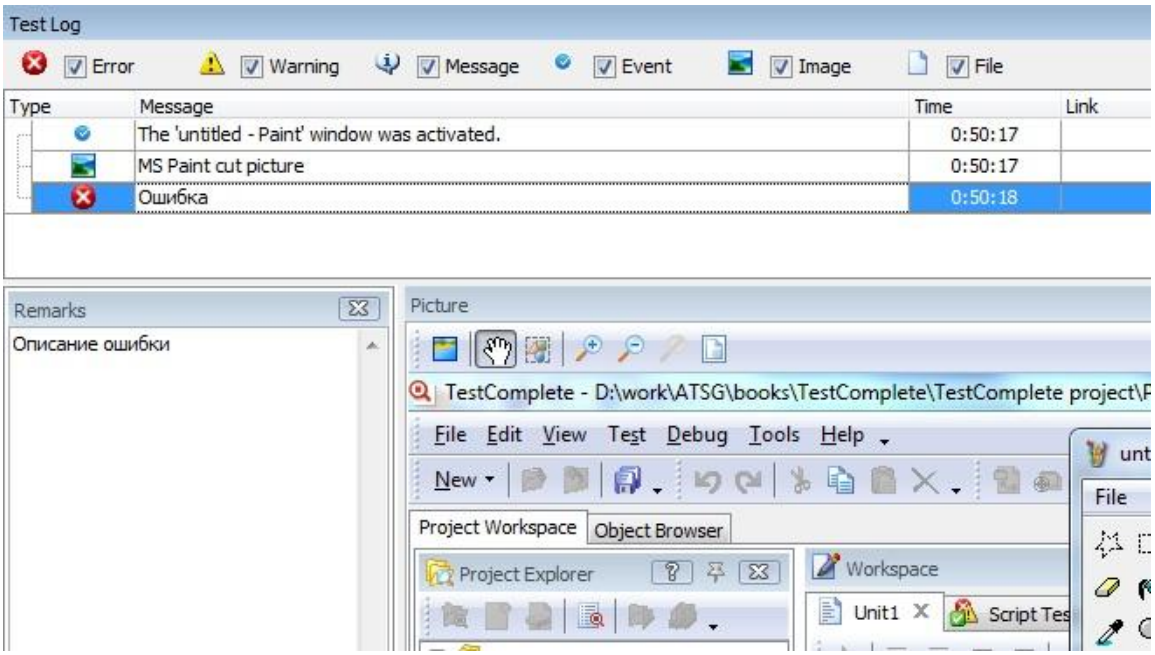


При выделении соответствующей строки лога, в его правой нижней части появляется панель с изображением, которое мы поместили в лог. На этой панели есть несколько кнопок, с помощью которых изображение можно открыть в новом окне, перемещать, изменять его масштаб и т.п.

Кроме метода Log.Picture(), можно поместить изображение в лог и при логгировании ошибок или предупреждений. Например, вы можете захотеть поместить в лог скриншот экрана во время возникновении ошибки. Для этого можно использовать следующий код:

Log.Error("Ошибка", "Описание ошибки", **undefined**, **undefined**, **Sys**.Desktop.Picture());

Результат:



Загрузка и выгрузка изображений

Захваченное изображение можно сохранить в файл или поместить в буфер обмена, а также сделать обратные операции. В следующем примере мы захватим картинку окна, сохраним ее в файл и поместим в буфер обмена, а затем вычитаем в другие переменные из файла и буфера обмена.

```
function TestImages2()

{

  var wPaint = Sys.Process("mspaint").Window("MSPaintApp", "*");

  wPaint.Activate();



  var myPicture = wPaint.Picture();

  myPicture.SaveToFile("c:\\1.png");

  Sys.Clipboard = myPicture;
```

```
  var Pic1 = Utils.Picture;

    Pic1.LoadFromFile("c:\\1.png");

  var Pic2 = Sys.Clipboard;



  Log.Picture(Pic1);

  Log.Picture(Pic2);

}
```

Обратите внимание на то, как с помощью метода Utils.Picture мы создали пустую переменную типа Picture (**var** Pic1 = **Utils**.Picture;).

Параметры изображений и их модификация

У объекта Picture есть несколько полезных свойств и методов, предназначенных для работы с самим изображением:

- свойство Size позволяет получать и задавать размеры изображения
- свойство Pixels позволяет получить или установить значение цвета для заданного пиксела в изображении
- метод Stretch – позволяет масштабировать изображение

В следующем примере мы захватим изображение окна MS Paint, затем с помощью масштабирования (метод Stretch) уменьшим его в 4 раза, затем с помощью свойства Size оставим от полученной картинки только верхнюю левую часть, а в заключении с помощью свойства Pixels нарисуем в изображении черную линию.

```
function TestImages3()

{

  var wPaint = Sys.Process("mspaint").Window("MSPaintApp", "*");

  wPaint.Activate();

  var myPicture = wPaint.Picture();



  // масштабируем

  myPicture.Stretch(myPicture.Size.Width/2, myPicture.Size.Height/2);

  Log.Picture(myPicture);
```

```
  // обрезаем
  myPicture.Size.Width = myPicture.Size.Width/2;
  myPicture.Size.Height = myPicture.Size.Height/2;
  Log.Picture(myPicture);


  // рисуем горизонтальную линию черного цвета длиной 99 пикселей
  for(i = 1; i < 100; i++)
  {
    myPicture.Pixels(i, 10) = 0x000000;
  }
  Log.Picture(myPicture);
}
```

Сравнение изображений

Сравнение изображений – это, пожалуй, самая важная часть этого раздела, так как именно для сравнения обычно и приходится их захватывать.

Сравнивать изображения можно двумя способами:

- используя объект Picture
- с помощью объекта Regions

Так как эти два подхода похожи и дают в принципе одинаковые результаты, мы рассмотрим подробно только первый из них, а второй оставим на рассмотрение читателям, перечислив лишь основные методы объекта Regions.

Существует 2 способа сравнения изображений:

- прямое сравнение двух картинок
- поиск области внутри картинки

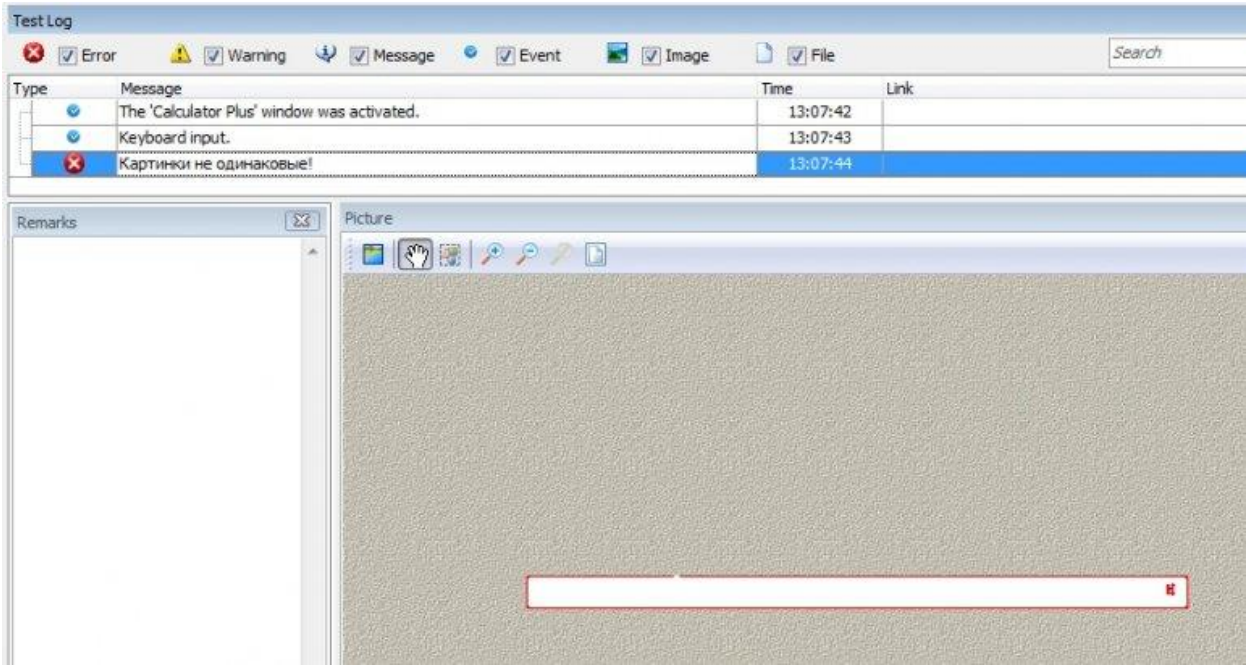
Для прямого сравнения картинок используется метод Difference объекта типа Picture. Этот метод сравнивает изображение, для которого вызывается метод, с другим изображением, которое передается методу в параметрах. Метод Difference возвращает null, если сравнение прошло успешно, и картинку-разницу, в случае если картинки неодинаковые.

Картинка-разница – это изображение, на котором белым цветом отмечаются одинаковые места из двух сравниваемых изображений, а красным – различия между ними.

В следующем примере мы сохраним изображение поля ввода Калькулятора (когда в нем находится цифра 0), затем введем цифру 9 и сравним первое изображение с полученным, а результат сравнения выведем в лог.

```
function TestImages4()

{

  var pic1, pic2, pic3;

  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");

  var wEdit = wCalc.Window("Edit");



  wCalc.Activate();

  pic1 = wEdit.Picture();

  wCalc.Keys("9");

  pic2 = wEdit.Picture();

  pic3 = pic1.Difference(pic2);

  if(pic3 != null)

  {

    Log.Error("Картинки не одинаковые!", undefined, undefined, undefined, pic3)

  }

}
```

Результат работы функции:



В некоторых случаях при сравнении картинок разница будет всегда. Например, если вы сравниваете окно, в котором присутствует текущая дата, то каждый день дата будет разной, а значит каждый день необходимо обновлять картинку. Чтобы этого не делать, метод Difference предоставляет 2 параметра:

- **Transparent** – позволяет отметить на рисунке область, которую TestComplete будет считать «прозрачной» и пропустит при сравнении. Для того, чтобы указать такой цвет, необходимо открыть сохраненную картинку в любом редакторе и отметить каким-то одним цветом (например, красным) самый первый пиксель (верхний левый) и всю область, которую следует пропускать при сравнении, после чего установить параметр Transparent в true. В нашем примере необходимо будет таким образом закрасить всё поле ввода. Первый пиксель необходимо закрашивать потому, что именно по нему TestComplete определяет «прозрачный» цвет при сравнении
- **Tolerance** – позволяет указать максимальное количество пикселей, которое может быть разным при сравнении и это не будет считаться ошибкой. Этот параметр может оказаться полезным в случае, когда элементы управления помещаются в окно динамически и их размеры могут иногда отличаться на 1-2 пикселя, однако при этом всё остальное остается неизменным. Учтите, что использование этого параметра может замедлить работу метода Difference.

Следующий пример будет сложным. В нём мы продемонстрируем работу обоих параметров. Для этого мы во время работы скрипта динамически закрасим красным цветом первый пиксель и всю область, где находится цифра (в правой части), а затем рассчитаем значение Tolerance, исходя из размеров поля ввода.

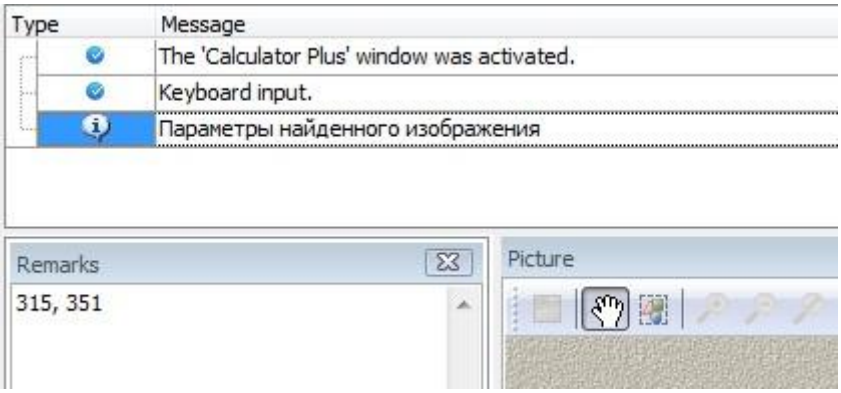
```
function TestImages5()
{
  var pic1, pic2, pic3;
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  var wEdit = wCalc.Window("Edit");

  wCalc.Activate();
  wCalc.Keys("[Esc]");
  pic1 = wEdit.Picture();
```

```
  wCalc.Keys("9");
  pic2 = wEdit.Picture();

  // закрашиваем первый пиксель и всю область возле первой цифры поля ввода
  pic1.Pixels(0, 0) = 0x0000FF;
  for(var i = 440; i <= 450; i++)
  {
    for(var j = 3; j <= 19; j++)
    {
      pic1.Pixels(i, j) = 0x0000FF;
    }
  }

  // рассчитываем значение Tolerance
  var iTol = wEdit.Height*2 + wEdit.Width*2;

  pic3 = pic1.Difference(pic2, true, iTol);
  if(pic3 != null)
  {
    Log.Error("Картинки не одинаковые!", undefined, undefined, undefined,
pic3)
  }
}
```

Для поиска области внутри имеющейся картинки используется метод Find объекта Picture. В качестве параметра ему необходимо передать другой объект типа Picture, который мы хотим найти внутри данной картинки. Кроме этого параметра можно задать координаты X и Y, начиная с которых будет осуществляться поиск, а также уже известные нам параметры Transparent и Tolerance. Метод Find возвращает объект типа Rect, из которого можно узнать координаты искомой картинки, или null если изображение не найдено. Естественно, искомое изображение должно быть меньше, чем изображение, в котором производится поиск.

В примере ниже мы покажем, как внутри картинки Калькулятора найти картинку, соответствующую кнопке 9.

```
function TestImages6()
{
  var pic1, pic2;
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  var wEdit = wCalc.Window("Edit");

  wCalc.Activate();
  wCalc.Keys("[Esc]");
  pic1 = wCalc.Picture();
  pic2 = wCalc.Window("Button", "9").Picture();

  var rect = pic1.Find(pic2);
  if(rect == null)
  {
    Log.Error("Картинка кнопки 9 не найдена внутри картинки Калькулятора");
  }
  else
  {
    Log.Message("Параметры найденного изображения", rect.Left + ", " +
rect.Right);
  }

}
```



Использование Regions

Если вы используете в своем проекте объект Regions, вы можете использовать его методы для сравнения изображений. Например, метод Compare сравнивает объект, хранящийся в Regions с другим изображением, а для поиска используются методы Find и FindRegion (эти методы одинаковы, разница лишь в порядке передаваемых параметров). Преимущество использования, например, метода Compare в том, что в случае, когда сравнение закончилось неудачей, TestComplete сам помещает в лог 3 картинки: две сравниваемых картинки и картинку-разницу, что существенно облегчает как написание скриптов, так и последующее чтение логов. Однако некоторые вещи (например, масштабирование) выполнить с помощью Regions не удастся.

Прочие особенности

При захвате изображений и сравнении картинок есть еще один параметр, который мы не рассматривали, – Mouse. Этот параметр позволяет при захвате и сравнении изображений учитывать также и курсор мыши (который по умолчанию игнорируется).

Кроме того, обратите внимание на то, что TestComplete поддерживает 4 формата изображений: BMP, GIF, JPG и PNG. В случае использования последних трех форматов при сравнении изображений TestComplete преобразует их к формату BMP. Для большинства случаев мы рекомендуем использовать формат PNG с установленным Compression level = 0. Этот формат сохраняет максимальную точность изображений и при этом занимает сравнительно немного места. Установить нужный формат можно в разделе *Tools – Options – General*.

Еще один важный момент: изображения рекомендуется захватывать и сохранять либо с помощью самого TestComplete, либо с помощью клавиши PrtScr и программы MS Paint. В случае использования любых других сторонних приложений возможны различия в сохраненных изображениях с теми, которые делает TestComplete.

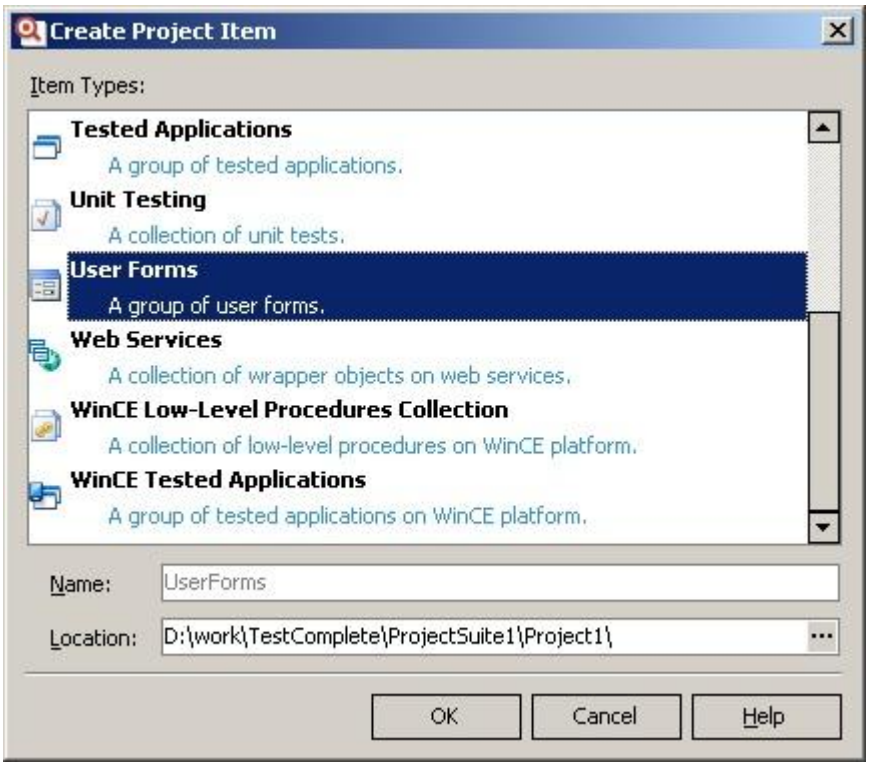
13 Пользовательские формы

Иногда перед запуском тестов бывает необходимо установить какие-то предварительные настройки. Например, задать логин и пароль, который будет затем использоваться в приложении, выбрать сервер, на котором будут запускаться тесты, указать человека, запустившего тесты на выполнение и т.п. и т.п.

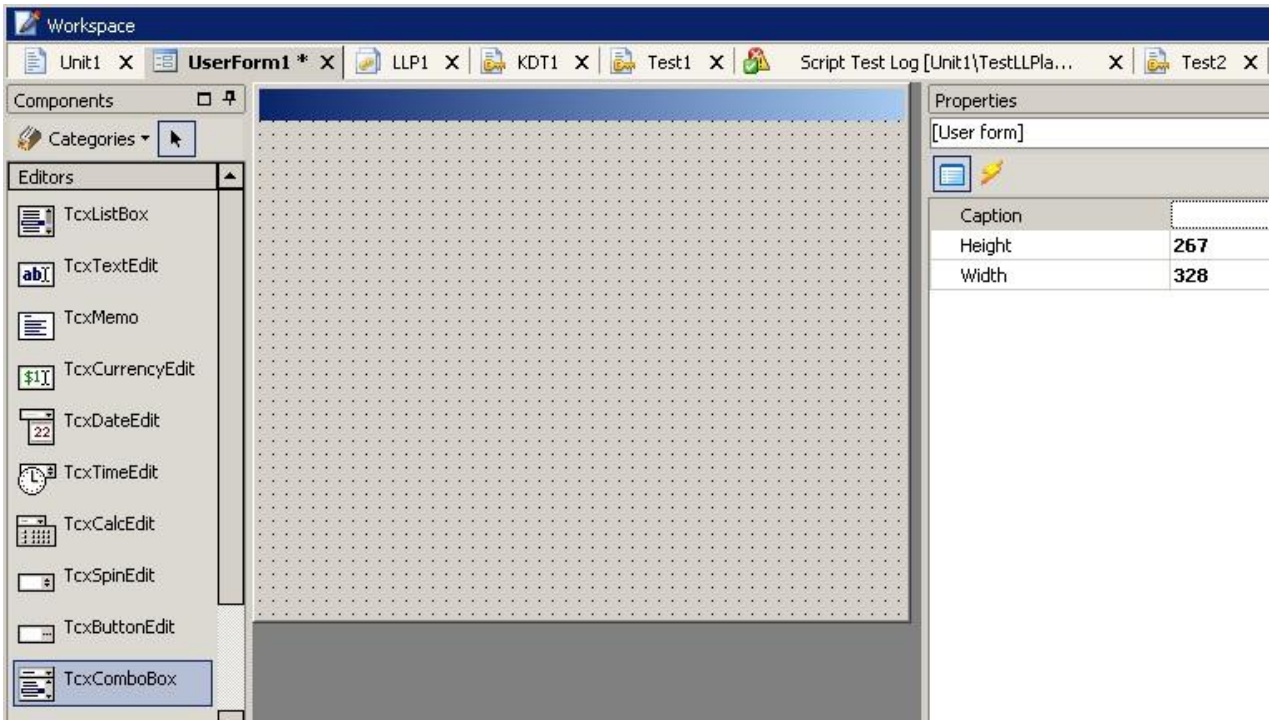
Конечно, все эти настройки можно хранить в INI или XML файлах, а можно оформить выбор параметров более красиво с помощью пользовательских форм.

Создание пользовательских форм в TestComplete не отличается от аналогичных конструкторов в визуальных системах разработки (например, Visual Studio или Delphi).

Для того, чтобы работать с пользовательскими формами, необходимо сначала добавить элемент User Forms в проект. Для этого щелкните правой кнопкой мыши по имени проекта, выберите пункт *Add – New Item* и в открывшемся диалоговом окне выберите элемент **User Forms**.



После этого таким же образом добавляем новую пользовательскую форму (правый щелчок на элементе **UserForms**, *Add – New Item*), задаем форме любое имя и нажимаем **ОК**. После чего нам открывается конструктор форм.



Мы не будем рассматривать все доступные элементы управления, которые можно использовать в пользовательских формах, так как их очень много: поля, списки, кнопки, флажки, панели, стандартные диалоги и прочие. Мы просто рассмотрим несколько примеров использования пользовательских форм.

Пример 1 – обработка возвращаемого формой результата

Предположим, у нас есть несколько серверов, на которых можно запускать скрипты, и непосредственно перед запуском необходимо выбрать необходимый сервер. Имена серверов можно хранить в массиве и отображать пользователю на форме в виде выпадающего списка.

Создадим новую пользовательскую форму и поместим на нее 3 элемента: ComboBox и 2 кнопки.

Для элемента **ComboBox** зададим следующие свойства:

- **Name** = cxServer
- **Text** = Select server…

Кнопка **OK** будет иметь следующие свойства:

Caption = OK

ModalResult = mrOk

Default = True

Кнопка **Cancel**:

- **Cancel** = True

- **ModalResult** = mrCancel

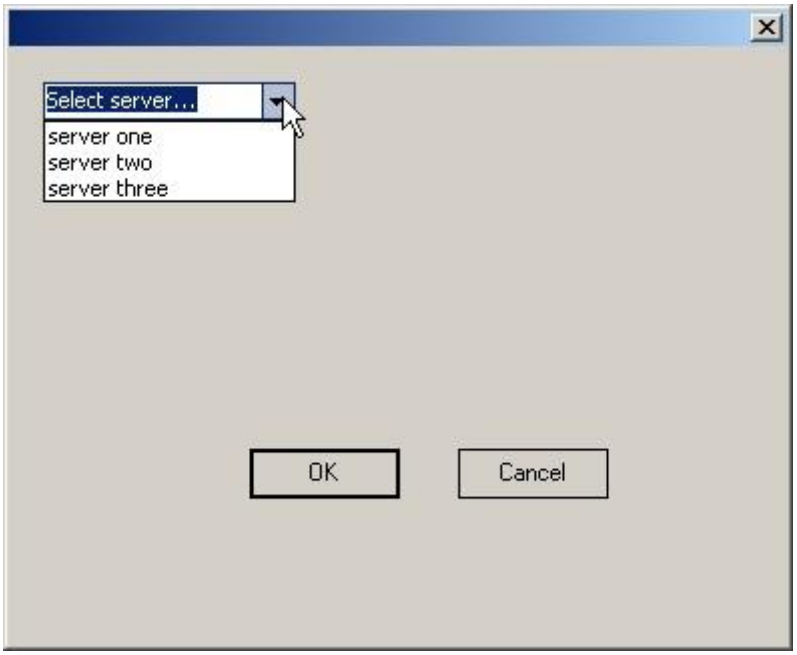
Свойство **ModalResult** очень важно: именно по нему мы сможем определить, какая именно кнопка была нажата – **OK** или **Cancel**, и дальше выполнить необходимые для каждой ситуации действия.

Для того, чтобы заполнить список элементами массива, необходимо изменить его свойство Properties.Items.Text, где перечислить элементы списка, разделив их символом перевода строки («\n»). Затем с помощью метода **ShowModal** отобразим форму на экране и проверим результат, с которым закроется форма (**mrOk** или **mrCancel**). Если пользователь нажмет кнопку **Cancel**, то мы прервем выполнение теста с помощью метода **Runner.Halt.**

Вот код скрипта, который выполняет все указанные действия:

```
function TestUserForms()
{
  var arrServers = new Array("server one", "server two", "server three");
  var i, mr, server;
  var fForm1 = UserForms.UserForm1;
  //заполняем список серверов
  for(i = 0; i < arrServers.length; i++)
  {
    fForm1.cxServer.Properties.Items.Text += arrServers[i] + "\n";
  }
  // Показать форму
  mr = fForm1.ShowModal();
  // проверяем результат, с которым закрылась форма
  if(mr == mrOk)
  {
    Log.Message("OK button is pressed");
    server = fForm1.cxServer.Text;
    Log.Message(server + " server selected")
  }
  else
  {
    Runner.Halt("Operation was aborted by user");
  }
}
```

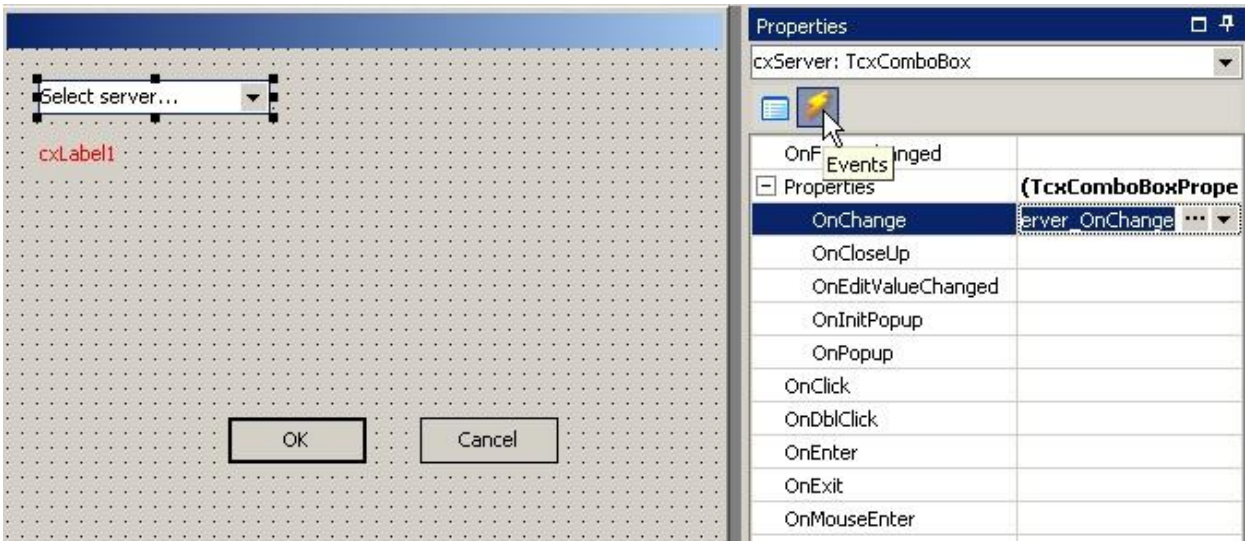
А вот как будет выглядеть наша форма для пользователя:



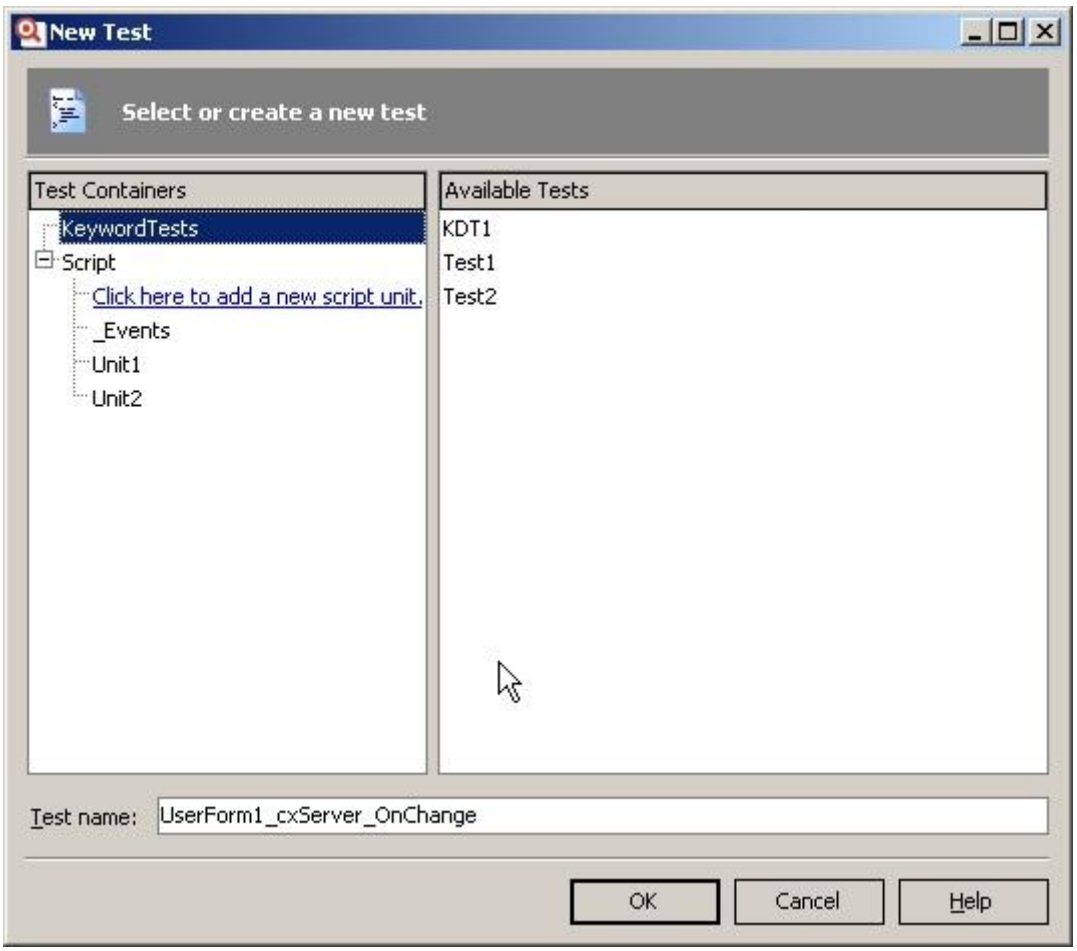
Пример 2 – обработка событий

Пример выше очень простой, так как пользователь может не просто выбрать элемент из списка, а ввести какой-то некст, который будет неправильным именем сервера, в результате чего все последующие тесты выдадут ошибку.

Предположим, что мы хотим предупреждать пользователя о том, что введенное значение не является значением из списка. Для этого поместим на форму дополнительный элемент управления – **Label**, свойство **Visible** ему установим в **False**. Затем выделим элемент список и на панели свойств перейдем на вкладку **Events**.



Выделим событие **OnChange**, как показано на скриншоте выше и нажмем кнопку с изображенным троеточием. При этом откроется диалоговое окно *New Test*, в котором мы укажем имя и расположение функции, которая будет связана с событием **OnChange**.

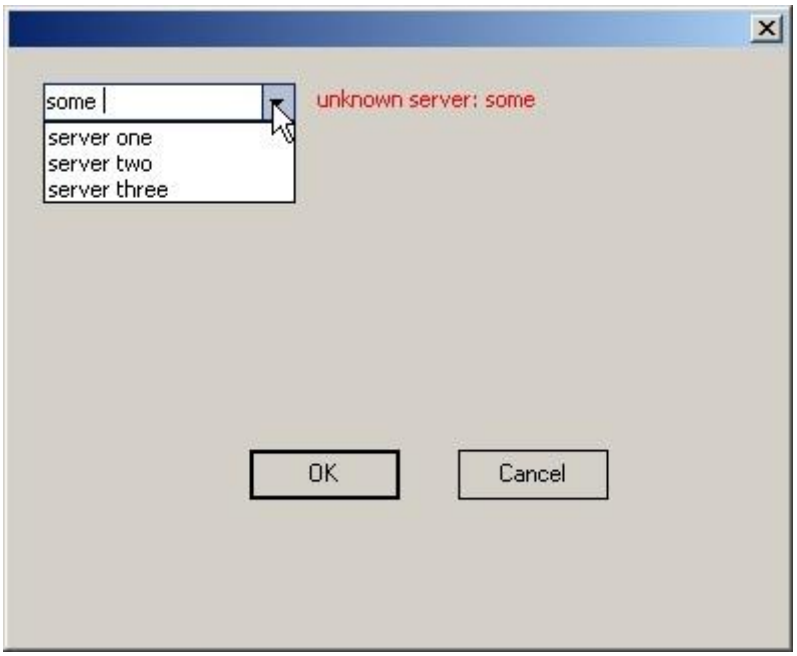


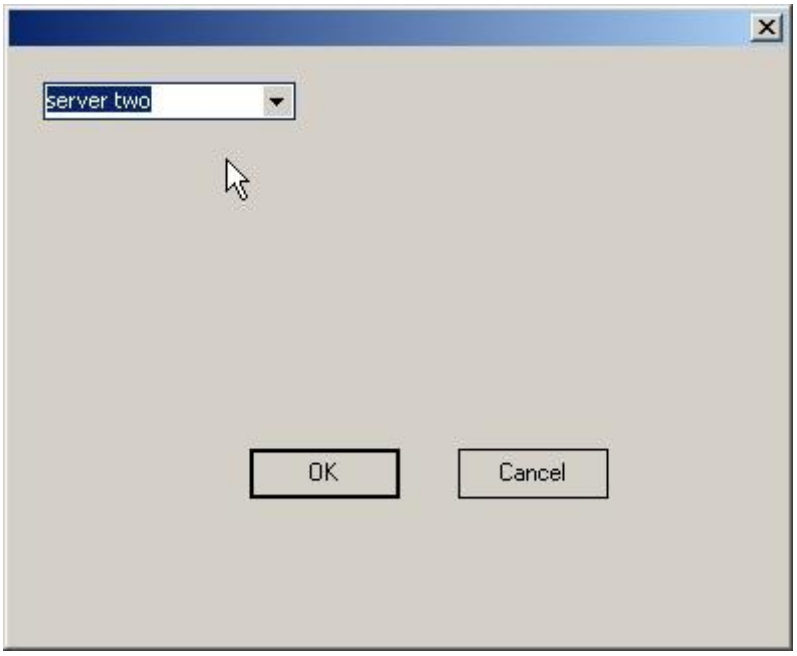
Нажмем ОК и напишем следующую функцию:

```
function UserForm1_cxServer_OnChange(Sender)
{
  var cbox = UserForms.UserForm1.cxServer;
  if(cbox.Properties.Items.Text.match(cbox.Text) == null)
  {
    UserForms.UserForm1.cxLabel1.Visible = true;
    UserForms.UserForm1.cxLabel1.Caption = "unknown server: " + cbox.Text;
  }
  else
  {
    UserForms.UserForm1.cxLabel1.Visible = false;
  }
```

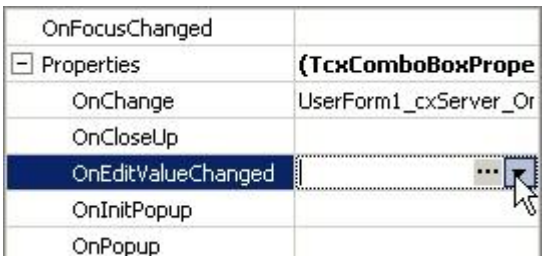
```
}
```

Теперь снова запустим созданную ранее функцию **TestUserForms** и убедимся, что при введении в поле списка любого значения, которое отсутствует в списке, у нас появляется красное предупреждение об этом, а при выборе элемента из списка надпись исчезает.





Естественно, одна и та же функция может использоваться как обработчик сразу нескольких событий разных элементов управления. Для этого достаточно при создании обработчика нажать не кнопку с троеточием, а кнопку с изображенным на ней треугольничком и в появившемся окне **Select Test** выбрать существующую функцию.



Возможно, приведенные примеры не очень хороши, так как на практике обычно встречаются более сложные пользовательские формы, однако мы решили не усложнять примеры, а сконцентрироваться на главных моментах.

Фактически, с помощью TestComplete User Forms можно написать довольно мощные приложения, однако помните, что TestComplete – это все же средство автоматизации тестирования, а не разработки :)

Хотя возможности TestComplete-а по созданию пользовательских форм велики, все же они не бесконечны, и вполне вероятно, что вам может понадобиться что-то, что невозможно сделать стандартными средствами. В таких случаях нужно постараться пораскинуть мозгами и придумать, как обойти имеющуюся проблему.

Например, вы хотите поместить на форму *radio list*, в котором по умолчанию выбран какой-то элемент. Например, так:

Однако среди списка свойств элемента **RadioGroup** нет свойства **SelectedValue**, **DefaultItem** или чего-то подобного, есть только свойство **ItemIndex**, в котором хранится выбранный в данный момент элемент. Чтобы обойти эту проблему и иметь выделенный по умолчанию элемент, достаточно создать обработчик события OnShow для нашей формы и написать там следующий код:

```
UserForms.UserForm1.cxRadioGroup1.ItemIndex = 1;
```

Теперь при открытии формы у нас будет по умолчанию выделен второй элемент списка.

14 TestComplete и COM

Изначально эта глава планировалась довольно большой и разбитой на подглавы: работа с файловой системой, работа с реестром, работа с MS Excel и т.д. Однако за время написания учебника выходили всё новые и новые версии TestComplete (первые главы учебника были написаны когда только-только вышла 6-я версия продукта) и многое из задуманного стало неактуальным. Появились новые объекты (например, aqFile, aqTextFile, aqEnvironment), которые позволяют делать все то, что планировалось здесь написать. Поэтому в итоге от этой главы осталось только основное.

Подключение к COM-объектам

TestComplete позволяет работать с COM-объектами так же просто, как и со своими внутренними объектами. Единственное, что нужно знать, – это свойства и методы соответствующего объекта и как их использовать.

Для доступа к COM-объекту используется следующий синтаксис:

var obj = Sys.OleObject(OBJECT_NAME);

где OBJECT_NAME – имя OLE-объекта.

Например, для доступа к приложению Excel используется объект «Excel.Application», а для работы с XML-файлами – объект «Msxml2.DOMDocument.3.0».

Один из наиболее часто используемых OLE-объектов – это объект WScript.Shell. С его помощью можно работать с файловой системой и реестром Windows. Так, следующий пример демонстрирует запуск программы без использования объекта TestedApps:

```
function TestOLE()

{

  var obj = Sys.OleObject("WScript.Shell");

  obj.Run("notepad.exe");

}
```

А следующий – выводит в лог количество процессоров на текущем компьютере (значение переменной окружения NUMBER_OF_PROCESSORS):

```
var obj = Sys.OleObject("WScript.Shell");

Log.Message(obj.ExpandEnvironmentStrings("%NUMBER_OF_PROCESSORS%"));
```

Того же эффекта можно добиться стандартными средствами языка программирования, который вы используете. Например (JScript):

```
var obj = new ActiveXObject("WScript.Shell");
```

```
Log.Message(obj.ExpandEnvironmentStrings("%NUMBER_OF_PROCESSORS%"));
```

Или VBScript:

```
Set obj = CreateObject("WScript.Shell")

Log.Message obj.ExpandEnvironmentStrings("%NUMBER_OF_PROCESSORS%")
```

На этом наше знакомство с возможностями TestComplete и COM заканчивается. К счастью возможности TestComplete с каждой новой версией расширяются, благодаря чему потребность в использовании COM-объектов постоянно уменьшается. Однако, если возможности TestComplete вас не удовлетворяют, вы можете обратиться к следующим источникам:

http://www.podgoretsky.com/ftp/Docs/WSH/Chebotarev/ – краткое, но очень емкое руководство по WScript.Shell

http://msdn.microsoft.com/en-us/library/ms763742%28v=VS.85%29.aspx – спецификации MSXML

http://autotestgroup.com/ru/materials/17.html – пример работы с Excel через OLE

Естественно, рассмотрение свойств и методов этих объектов выходит далеко за рамки этого учебника.

TestComplete как COM-сервер

Сам TestComplete также является COM-сервером, что позволяет подключаться к нему и работать с ним из других программ. Например, в следующем примере демонстрируется, как подключиться к TestComplete из Visual Basic 6, открыть набор проектов и затем закрыть TestComplete.

```
Set TestCompleteApp = CreateObject("TestComplete.TestCompleteApplication")

Set IntegrationObject = TestCompleteApp.Integration
IntegrationObject.OpenProjectSuite "C:\Documents and Settings\All
Users\Documents\TestComplete 7 Samples\Scripts\Test Log\TestLog.pjs"

TestCompleteApp.Quit
```

Подробнее о подключении к TestComplete можно прочитать в справочной системе в разделе «Working With TestComplete via COM»

15 Распределенное тестирование

В этой главе рассматривается распределенное тестирование: возможность одновременного запуска тестов TestComplete на разных компьютерах.

В общем случае распределенное тестирование в TestComplete можно разделить на 2 типа: распределенное нагрузочное тестирование и распределенное функциональное тестирование.

Распределенное нагрузочное тестирование

О нагрузочном тестировании рассказывается в главе 4.2 Нагрузочное тестирование Web-приложений. Из-за некоторых ограничений операционной системы, TestComplete может эмулировать действия не более чем 300 пользователей с одного компьютера. Если вы хотите нагрузить свое приложение более чем тремястами соединениями, вам необходимо распределить нагрузку между несколькими компьютерами. Чтобы использовать компьютер для нагрузочного тестирования и подключать его к работе нагрузочных тестов, на этом компьютере должно быть установлено и запущено одно из приложений:

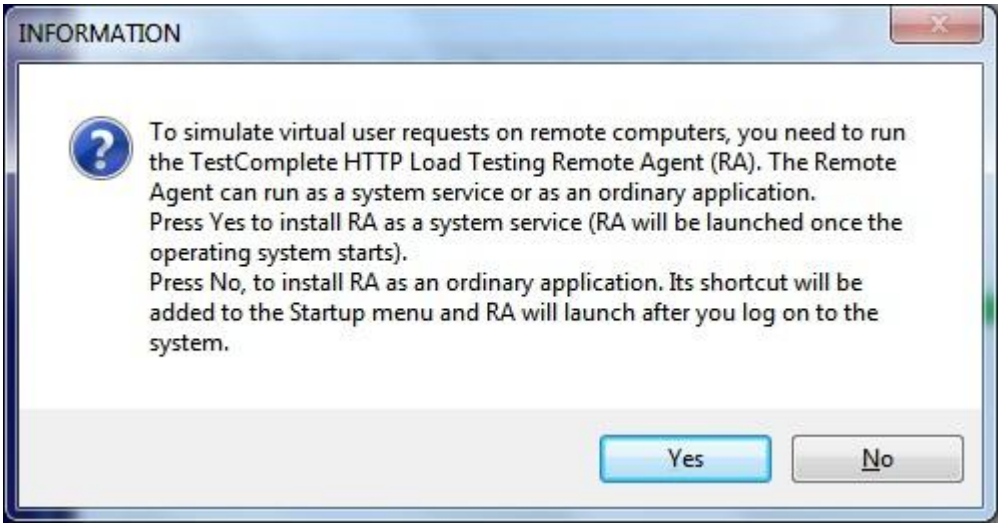
- TestComplete
- TestExecute
- Load Testing Remote Agent

Обратите внимание: TestComplete, TestExecute или Remote Agent, установленные на дополнительных компьютерах, должны быть той же версии, что и TestComplete или TestExecute на основной машине!

Мы рассмотрим пример распределенного нагрузочного тестирования на примере утилиты Remote Agent (RA).

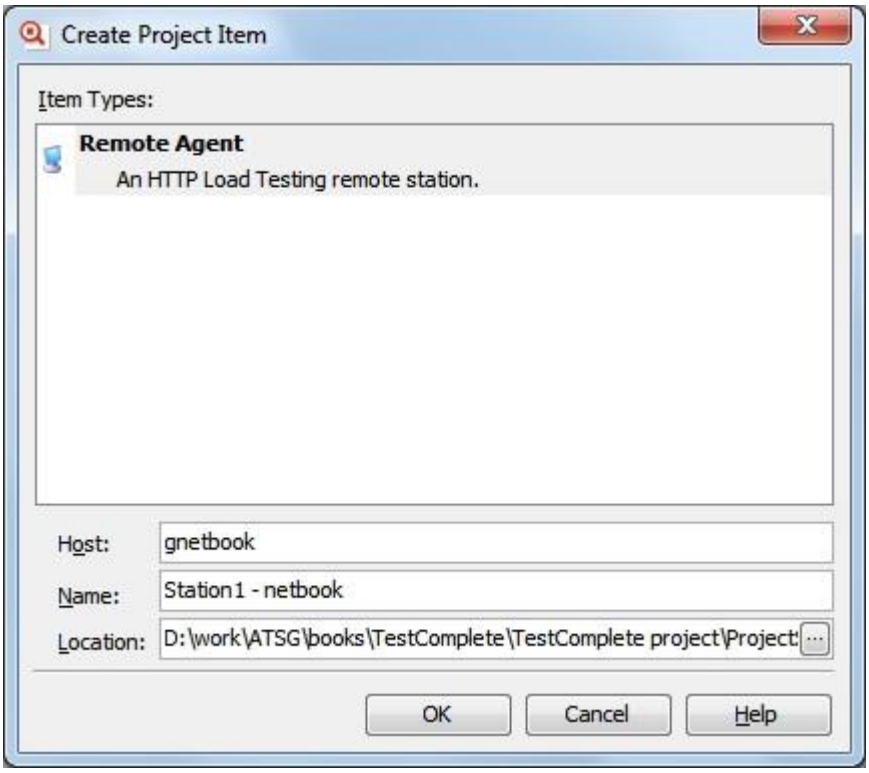
Remote Agent – это специальная утилита для эмуляции виртуальных пользователей. Скачать ее можно на портале сайта SmartBear, так же, как и TestComplete.

RA может запускаться как в обычном режиме, так и как сервисное приложение (Service). Выбор способа запуска происходит во время установки утилиты, когда появляется такое сообщение:



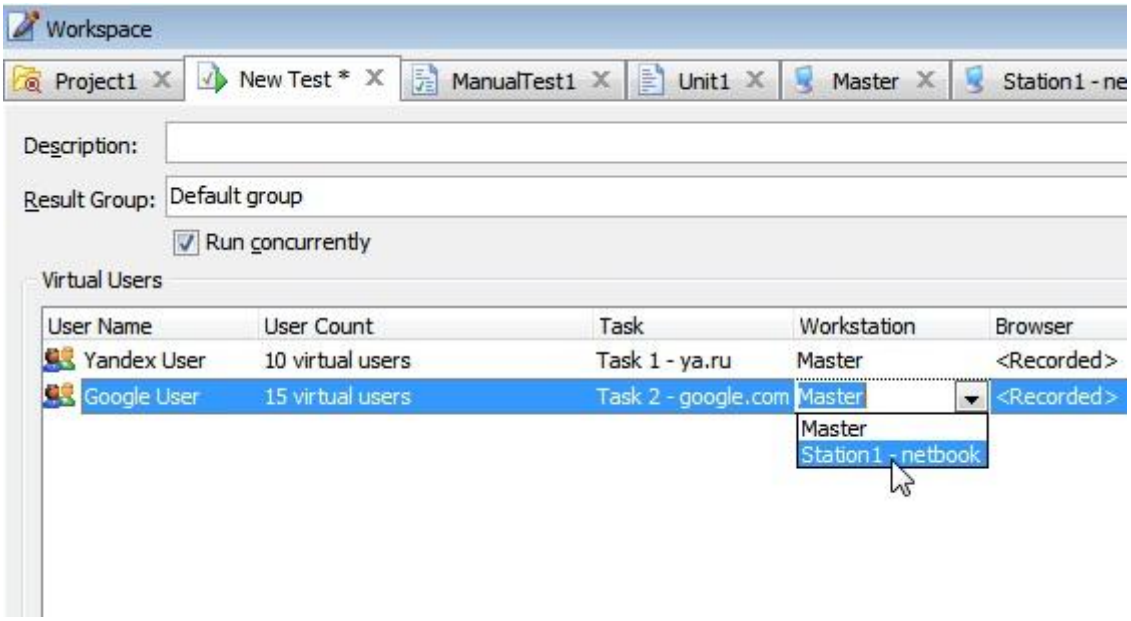
Если ответить Yes, приложение будет запускаться как сервис при старте операционной системы, иначе же RA будет установлено как обычное приложение и его надо будет запускать вручную.

Теперь можно добавить в проект дополнительные компьютеры, которые будут участвовать в нагрузочном тестировании. Для этого сделаем правый щелчок мышью на элементе проекта LoadTesting – Stations, выберем пункт меню Add – New Item и в открывшемся диалоговом окне введем параметры добавляемого компьютера.

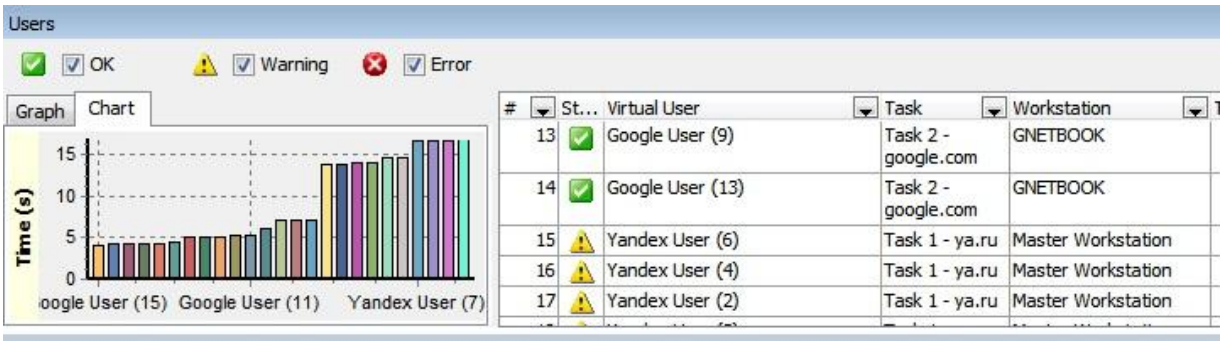


- Host – это имя или IP адрес подключаемого компьютера
- Name – имя станции, как оно будет отображаться в TestComplete-е

Дальше нам необходимо указать, какие тесты будут использовать добавленную рабочую станцию. Для этого откроем любой тест (New Test в нашем проекте) и укажем для каждого виртуального пользователя, на какой станции выполнять нагрузочный тест.



Если теперь запустить этот тест, то в результате мы увидим, что тесты для Яндекса выполнились на основной машине, а для Гугла – на дополнительной.



Если вы запускаете нагрузочные тесты из скриптов и хотите управлять использованием рабочих станций, вам необходимо модифицировать свойство Station объекта VirtualUser. В качестве примера модифицируем кусок кода из главы 4.2 Нагрузочное тестирование Web-приложений (изменения выделены полужирным):

```
// назначаем каждому пользователю задачу и тест
  for(i = 0; i < aUsers.length; i++)
  {
    aUsers[i] = LoadTesting.CreateVirtualUser("New created user " + i.toString());
    aUsers[i].Task = LoadTesting.HTTPTaskByName("Task 2 - google.com");
    aUsers[i].TestInstance = ti;

    aUsers[i].Station = LoadTesting.Stations.ItemByName("Station1 - netbook");

  }
```

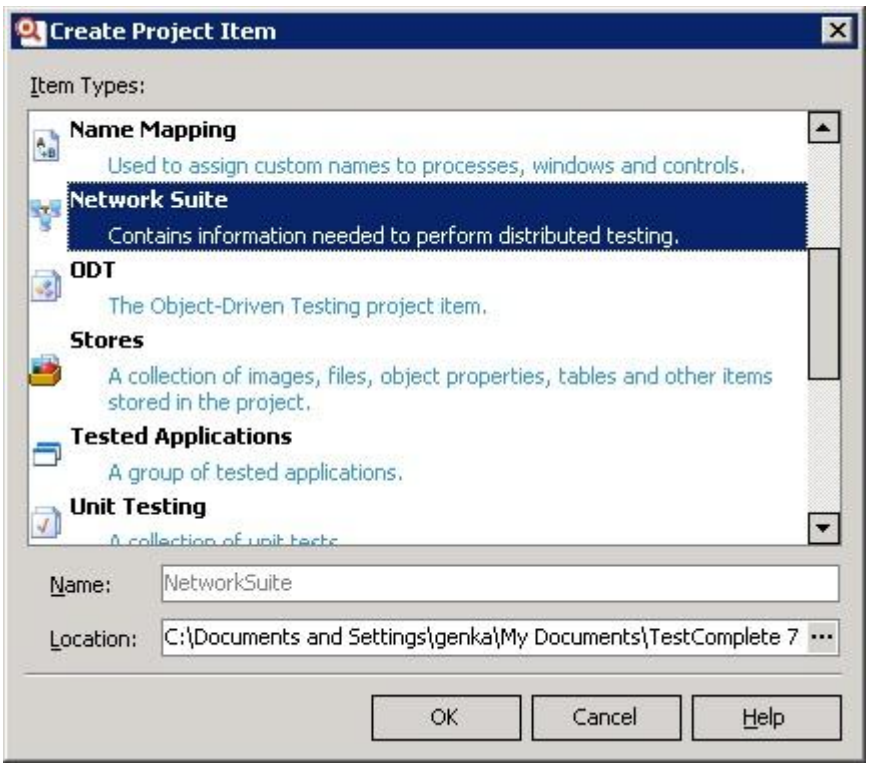
Кроме того, если вы запускаете нагрузочные тесты из скриптов, вам может пригодиться метод Rescan объекта LoadTesting.Stations, который проверяет доступность всех рабочих станций и генерирует исключение, если хотя бы одна из них недоступна.

Распределенное функциональное тестирование

TestComplete позволяет также распределять выполнение функциональных тестов по разным компьютерам. Это может быть полезно, например, в том случае, если запуск всех скриптов занимает слишком много времени и скрипты не успевают выполниться даже за ночь; или для запуска одних и тех же тестов на разных конфигурациях (разные операционные системы, пользователи с разным уровнем доступа и т.п.).

В распределенном функциональном тестировании используются 2 типа компьютеров: master (главный компьютер) и slave (подчиненные компьютеры). Master компьютер управляет запуском скриптов на подчиненных компьютерах.

Чтобы иметь возможность запускать скрипты на нескольких машинах, на каждой из них должен быть установлен либо TestComplete, либо TestExecute. Кроме того, в каждом проекте, который будет запускаться на slave-компьютере, должен быть добавлен элемент NetworkSuite. Чтобы добавить его, необходимо щелкнуть правой кнопкой мыши на имени проекта в дереве Project Explorer, выбрать пункт меню Add – New Item, и в открывшемся диалоговом окне выбрать Network Suite.



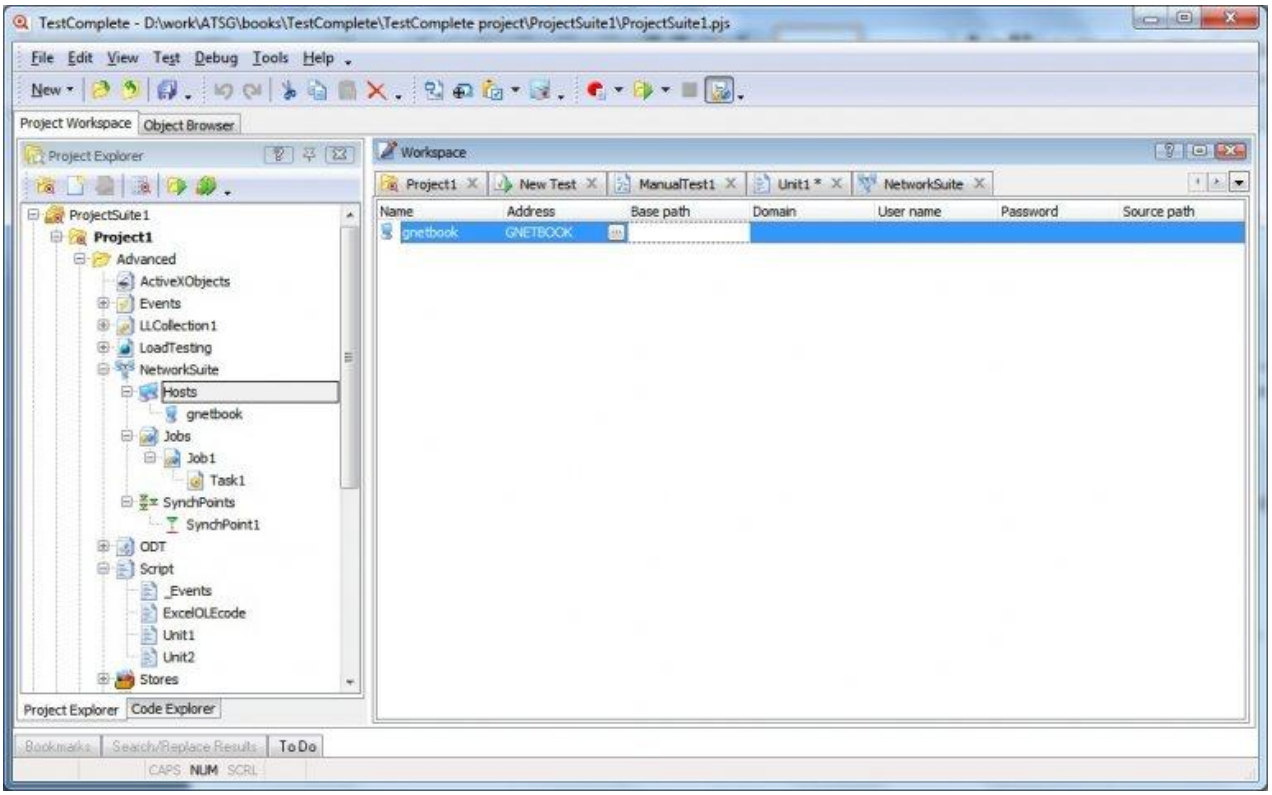
То же самое необходимо сделать на master-компьютере.

Теперь на master-компьютере необходимо указать компьютеры и задачи, которые будут запускаться удаленно. Для этого выполним правый щелчок на элементе Hosts и выберем пункт Add – New Item, затем в открывшемся диалоговом окне введем имя станции (оно не обязательно должно совпадать с именем компьютера). Затем выберем добавленный компьютер в списке Project Explorer и введем его параметры:

- **Address** – имя или IP-адрес компьютера
- **Domain, User name, Password** – параметры учетной записи компьютера, к которому будем подключаться

- **Base path** – общий путь для нескольких проектов, размещенных в одной папке slave-компьютера; этот параметр будет использоваться для формирования путей у задач (Tasks)
- **Source path** – путь к папке на master-компьютере, откуда будет копироваться проект на slave-компьютер

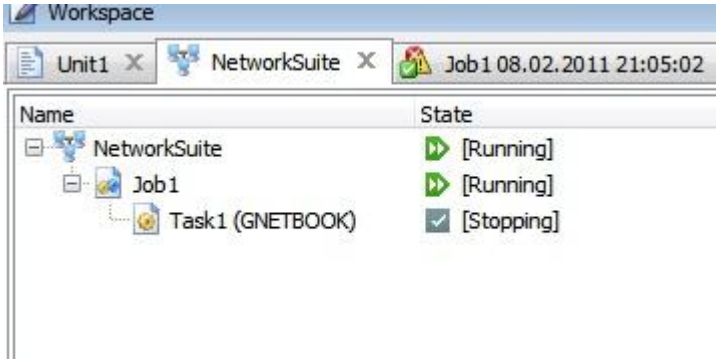
Из всех перечисленных параметров обязательным является только имя компьютера. Параметры учетной записи по умолчанию берутся с текущего компьютера; Base path может быть опущен и использоваться полные пути к проектам; Source Path может быть также опущен, если проект находится на удаленной машине.



Теперь добавим задачи. Прежде всего необходимо добавить Job. Job – это просто средство для упорядочивания задач (Tasks). В каждом элементе Job может быть несколько задач. Задача же, в свою очередь, содержит всю информацию о запускаемом тесте (имя проекта или функции, компьютер, на котором эта задача будет запускаться, приложение, которое будет использоваться на удаленной машине для запуска и т.д. Процесс добавления Job-ов и задач аналогичен процессу добавления Host-ов, рассмотренному выше. В итоге в нашем проекте добавляется задача, которую можно запустить на удаленном компьютере:

Можно запустить задачу прямо из редактора TestComplete, щелкнув по Job-е правой кнопкой мыши и выбрав пункт Run…

При этом TestComplete будет отображать ход текущих операций, производимых на удаленных компьютерах.



Для того, чтобы запустить на выполнение скрипты сразу на нескольких подчиненных компьютерах, необходимо прежде всего добавить эти компьютеры и запускаемые задачи в проект, как рассказано выше. После чего воспользоваться методом Run() объекта NetworkSuite.

```
NetworkSuite.Run(true);
```

У этого метода лишь один параметр – *WaitForCompletion* – который задает, нужно ли ждать окончания этой задачи, прежде чем переходить к следующей. Именно благодаря этому параметру можно обеспечить одновременный запуск скриптов на нескольких компьютерах, устанавливая его в false.

Здесь, однако, есть особенность. Если выполнение скрипта на master-компьютере завершится раньше, чем выполнение удаленных скриптов, то и выполнение скриптов на удаленных машинах тоже будет завершено. Это можно легко продемонстрировать следующим примером:

```
function TestNetworkSuite()

{

  NetworkSuite.Run(false);

  aqUtils.Delay(1000);

  Log.Message("Done");
```

```
}
```

При этом на удаленном скрипте используем задержку в 10 секунд:

```
aqUtils.Delay(10000);
```

Если запустить этот скрипт, то задержка 10 секунд на удаленном компьютере так и не дойдет до конца, удаленный TestComplete будет завершен раньше.

Чтобы избежать этой ситуации, в TestComplete предусмотрены точки синхронизации (SynchPoints). Точки синхронизации – это просто элементы, которые позволяют приостанавливать выполнение скриптов, которые по каким-либо причинам выполняются быстрее, до тех пор, пока этой же точки не достигнут другие скрипты. Как только все скрипты достигнут данной точки, выполнение на всех компьютерах продолжится. Таким образом, чтобы избежать проблемы, описанной выше, достаточно во всех проектах (включая master) добавить точку синхронизации (в нашем случае это “SynchPoint1”) и модифицировать скрипты таким образом, чтобы в конце выполнения ожидалась синхронизация. Делается это с помощью метода Synchronize:

```
function TestNetworkSuite()
{
  NetworkSuite.Run(false);
  aqUtils.Delay(1000);
  Log.Message("Done");
  NetworkSuite.Synchronize("SynchPoint1");
}
```

И то же самое на удаленной машине:

```
function TestNetworkSuite()
{
  Log.Message("This function is ran on remote PC");
  aqUtils.Delay(10000);
  NetworkSuite.Synchronize("SynchPoint1");
}
```

Еще одной важной особенностью распределенного тестирования является использование NetworkSuite Project Variables, т.е. переменных, которые доступны для всех запущенных скриптов, как на master, так и на slave-машинах. Добавлять и изменять эти переменные

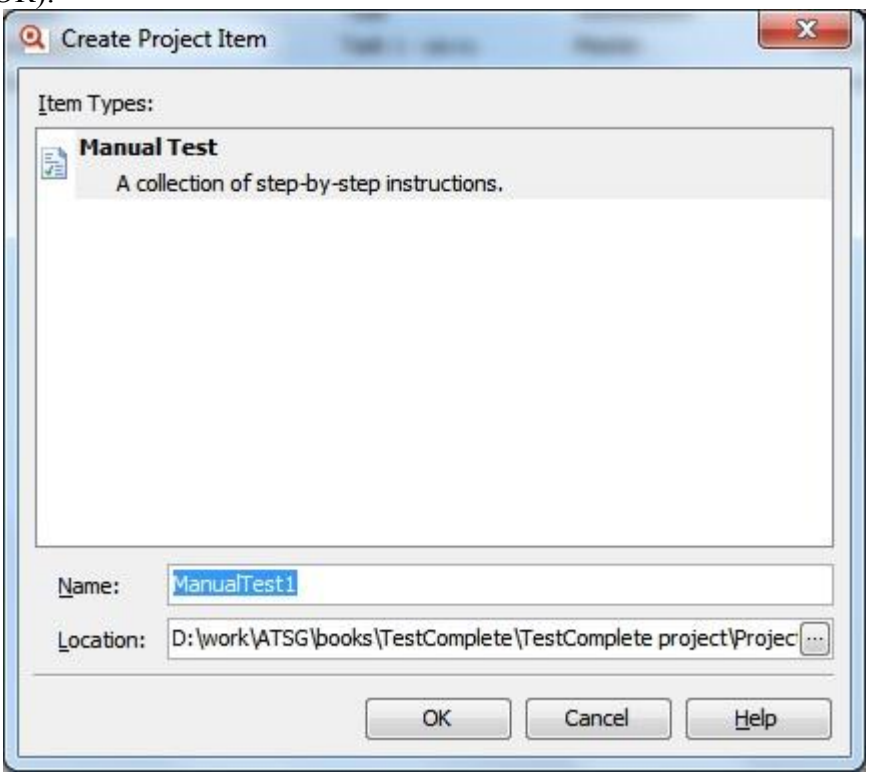
можно, дважды щелкнув на элементе NetworkSuite в Project Explorer. Эти переменные бывают двух типов:

- **Persistent** – сохраняют свои значения между запусками скриптов
- **Temporary** – заново инициализируются при каждом запуске

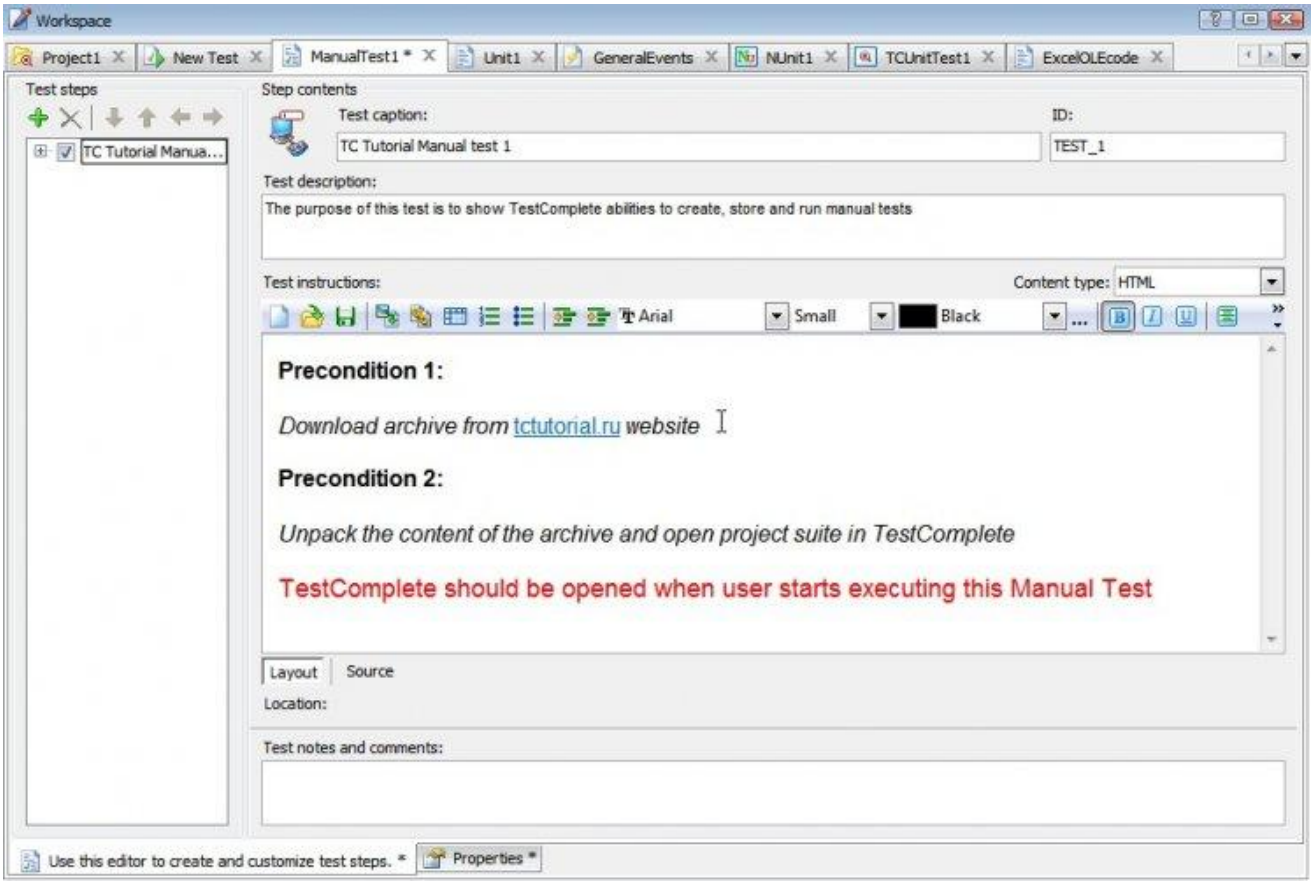
Если вам необходимо обмениваться данными между скриптами на разных компьютерах, воспользуйтесь NetworkSuite Project Variables. Подробное описание по работе с ними есть в справочной системе TestComplete, раздел «Network Suite Variables».

16 Ручное тестирование

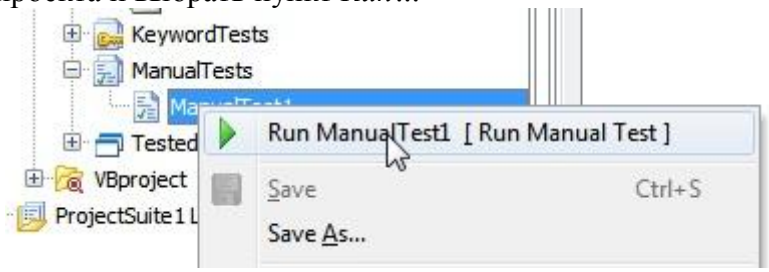
TestComplete позволяет создавать и хранить в проектах ручные тесты, т.е. тесты, которые пользователь должен выполнять самостоятельно, шаг за шагом, по ходу отмечая пройденные шаги и отмечая ошибки.
Ценность данного элемента в программе, предназначенной для автоматизации тестирования, представляется нам весьма сомнительной. На наш взгляд, тесткейсы необходимо хранить в специальных программах (например, тесттрекинговых или багтрекинговых системах), однако это мнение не помешает нам рассмотреть создание и запуск ручных тестов в TestComplete :)
Как обычно, сначала необходимо добавить в проект соответствующий элемент проекта (правый щелчок на проекте, *Add – New Item, Manual Tests*). Затем добавляем тест (правый щелчок на элементе **ManualTests** в проекте, *Add – New Item*, вводим имя теста и нажимаем OK).



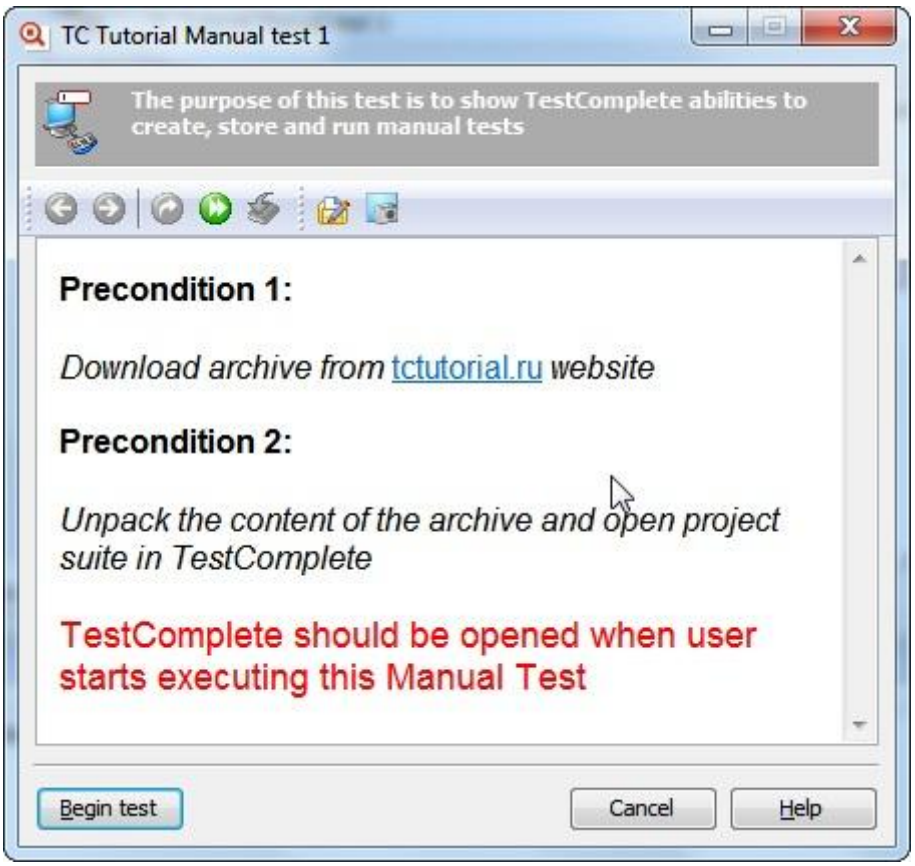
При этом у нас открывается панель **Manual Test,** где мы можем добавлять шаги тесткейса, менять его описание и название и т.д. Шаги редактируются в левой части панели (Test steps). В правой же части панели находятся поля **Caption, Description и Instructions** текущего теста или шага (в зависимости от того, что выбрано в списке слева).



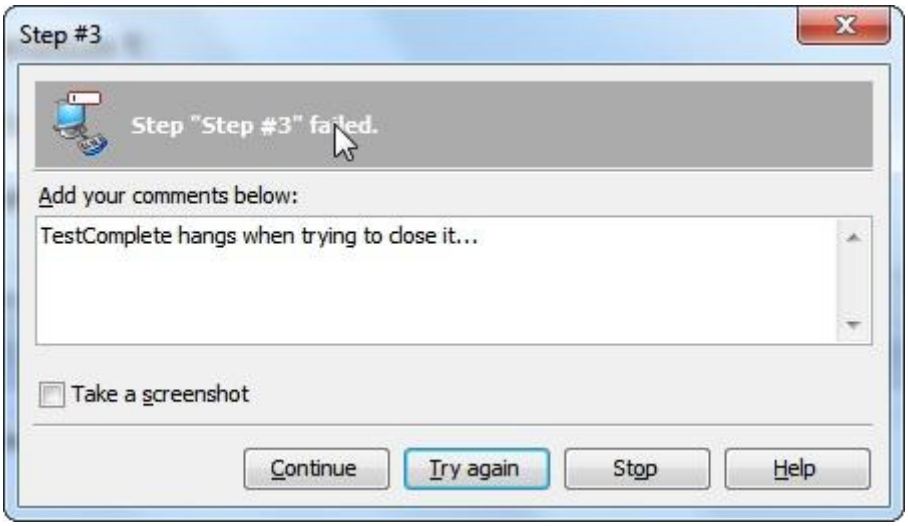
Как видно из примера, поля **Caption и Description** являются обычными, а поле **Instructions** позволяет использовать форматирование для оформления текста (размер, цвет шрифта, выравнивание на странице, списки, отступы, таблицы и т.д.). Мы не будем подробно останавливаться на этих вопросах, так как эти возможности простые и обычные. Теперь мы можем запустить созданный тест. Для этого, как и для запуска автоматического скрипта, необходимо щелкнуть правой кнопкой мыши на тесте в дереве проекта и выбрать пункт *Run*…



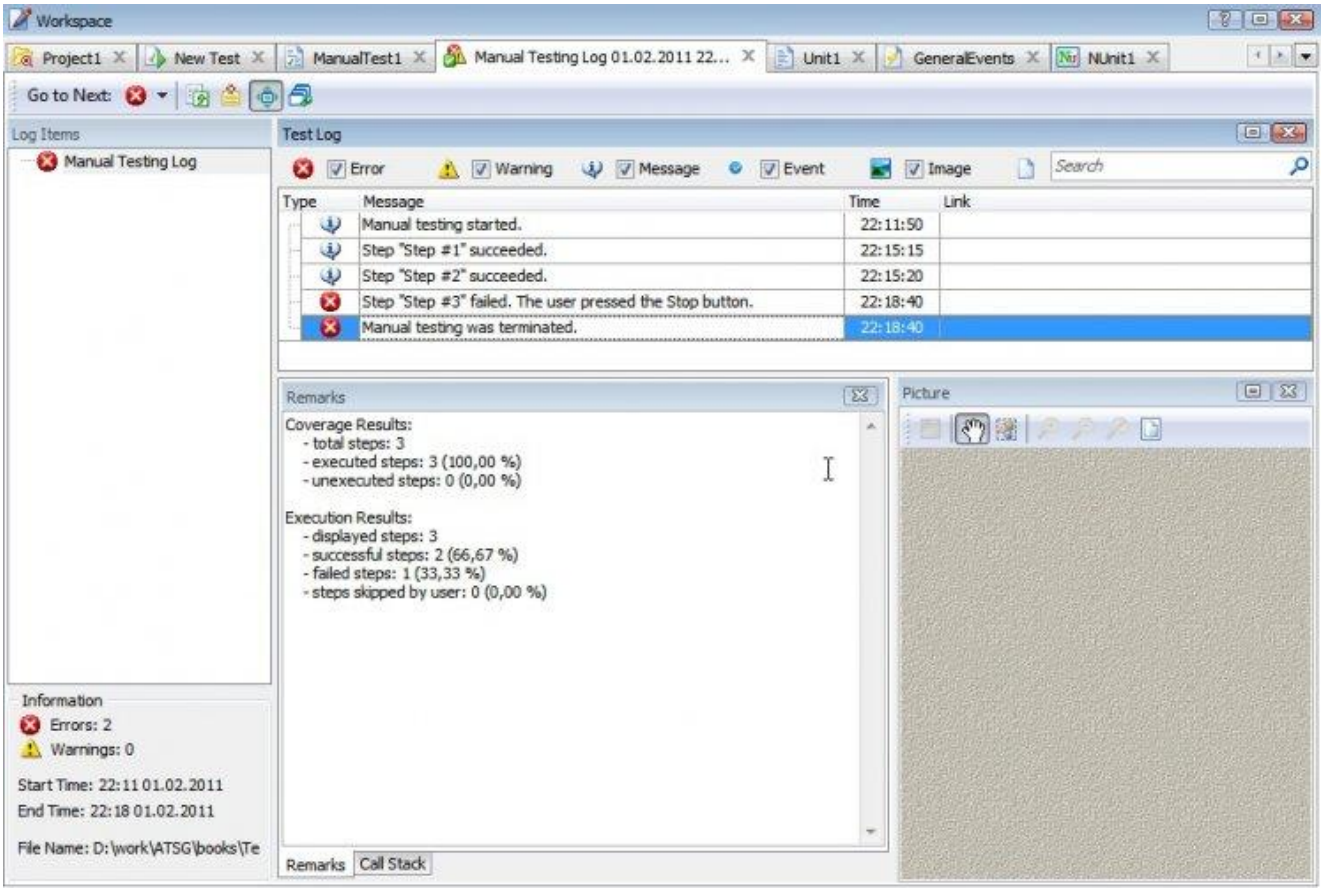
После чего на экране появится окно, которое перекрывает все остальные окна на экране (как это бывает в системах тест-трекинга, так как во время прохождения теста пользователь должен постоянно иметь доступ к тесткейсу).



Читаем инструкции и нажимаем кнопку **Begin test.** После этого TestComplete показывает нам шаги по порядку в том же окне. Для каждого шага предусмотрены варианты **Success** и **Fail** (соответственно, проделали ли мы указанные инструкции успешно или нет). При нажатии **Fail** открывается окно для ввода комментария, в котором мы объясняем, что именно в тесте пошло не так.



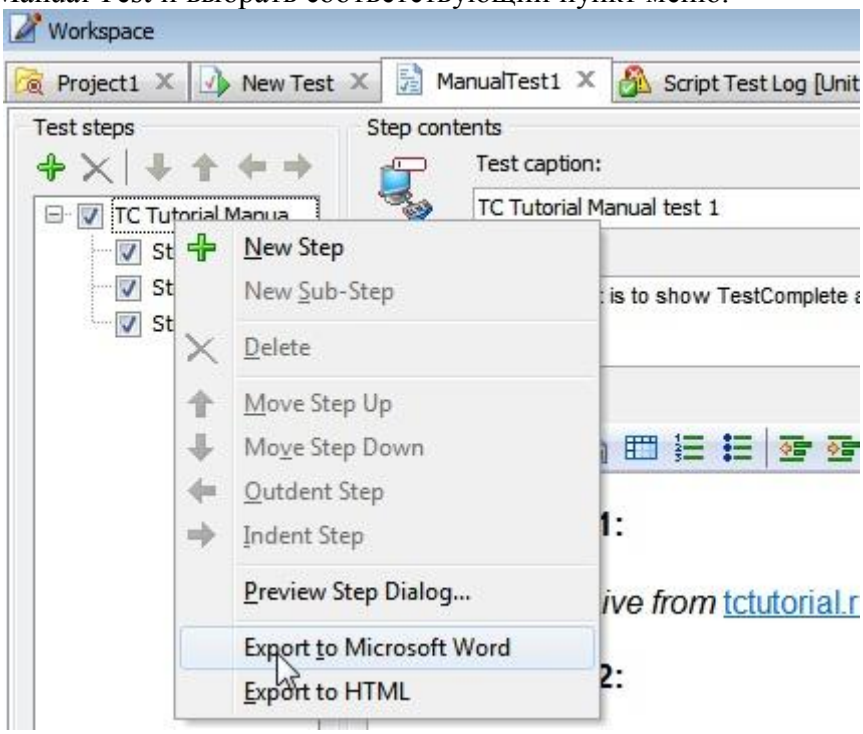
Из этого окна можно продолжить выполнение **(Continue),** попробовать пройти тот же шаг опять **(Try again)** или прервать выполнение теста **(Stop).**
В результате мы получим лог, похожий на логии автоматических тестов.



Вызвать ручной тест на выполнение можно также из скрипта. Например:

```
function TestManualTests()
{
  ManualTest1.Start();
}
```

Также любой ручной тест можно экспортировать в HTML или Word файл. Для этого необходимо открыть тест, щелкнуть правой кнопкой мыши по имени теста в панели Manual Test и выбрать соответствующий пункт меню.



Также ручные тесткейсы можно создавать непосредственно из скрипта TestComplete. Для этого используется объект **ManualTestBuilder**. Пример использования объекта **ManualTestBuilder**
можно найти в справочной системе TestComplete, раздел Creating Manual Tests From Scripts.

17 Модульное тестирование

Модульные тесты – это тесты, которые проверяют корректность работы отдельной функции или метода. Модульные тесты обычно пишутся программистами и служат для первичной проверки того, что внесенные изменения не изменили поведение отдельных компонентов системы.

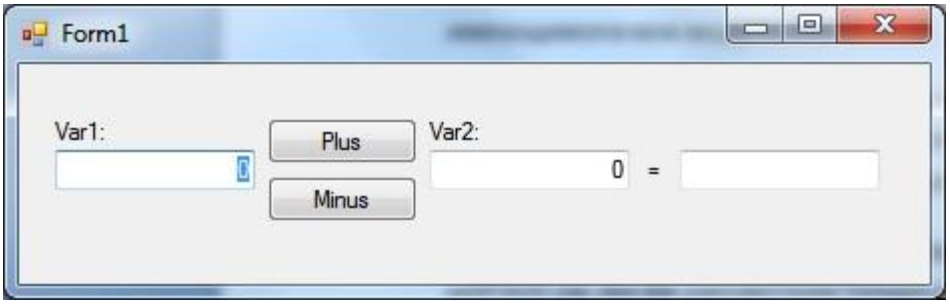
TestComplete позволяет запускать некоторые виды модульных тестов. Для этого либо используются сторонние библиотеки, к которым обращается TestComplete при запуске тестов, либо вносятся изменения в тестируемое приложение, чтобы TestComplete имел доступ к имеющимся в нем модульным тестам.

TestComplete поддерживает следующие типы модульных тестов: MSTest, JUnit, NUnit, DUnit и TCUnitTest. Первые четыре запускаются при помощи отдельных утилит, а последний тип тестов требует внесения изменений в тестируемое приложение.

В справочной системе TestComplete подробно рассмотрены все шаги по подключению и запуску unit test-ов, мы же рассмотрим только два примера запуска модульных тестов для .NET приложения: NUnit и TCUnitTest.

Тестовое приложение

Специально для этой главы мы создали небольшое .NET-приложение (его можно найти в архиве с примерами, папка nUnitTestingApp).



В поля Var1 и Var2 вводятся числовые значения, а затем нажимается кнопка Plus или Minus. В зависимости от нажатой кнопки в третье поле помещается результат сложения либо вычитания этих переменных соответственно. Вот как выглядит код сложения и вычитания, а также код нажатий на кнопки:

```
public static int PlusMethod(int var1, int var2)
    {
       return var1 + var2;
    }
```

```
    public static int MinusMethod(int var1, int var2)
    {
        return var1 - var2;
    }

    private void button1_Click(object sender, EventArgs e)
    {
        txtResult.Text = (PlusMethod(Convert.ToInt32(this.txtVar1.Text),
            Convert.ToInt32(this.txtVar2.Text))).ToString();
    }

    private void button2_Click(object sender, EventArgs e)
    {
        txtResult.Text = (MinusMethod(Convert.ToInt32(this.txtVar1.Text),
            Convert.ToInt32(this.txtVar2.Text))).ToString();
    }
```

Ниже мы поместили класс MyTests, который выполняет 3 проверки для метода PlusMethod:

```
// unit tests
    [TestFixture]
    public class MyTests
    {
        [Test]
        public void PlusTest1()
        {
            Assert.AreEqual(Form1.PlusMethod(2, 3), 5);
        }
```

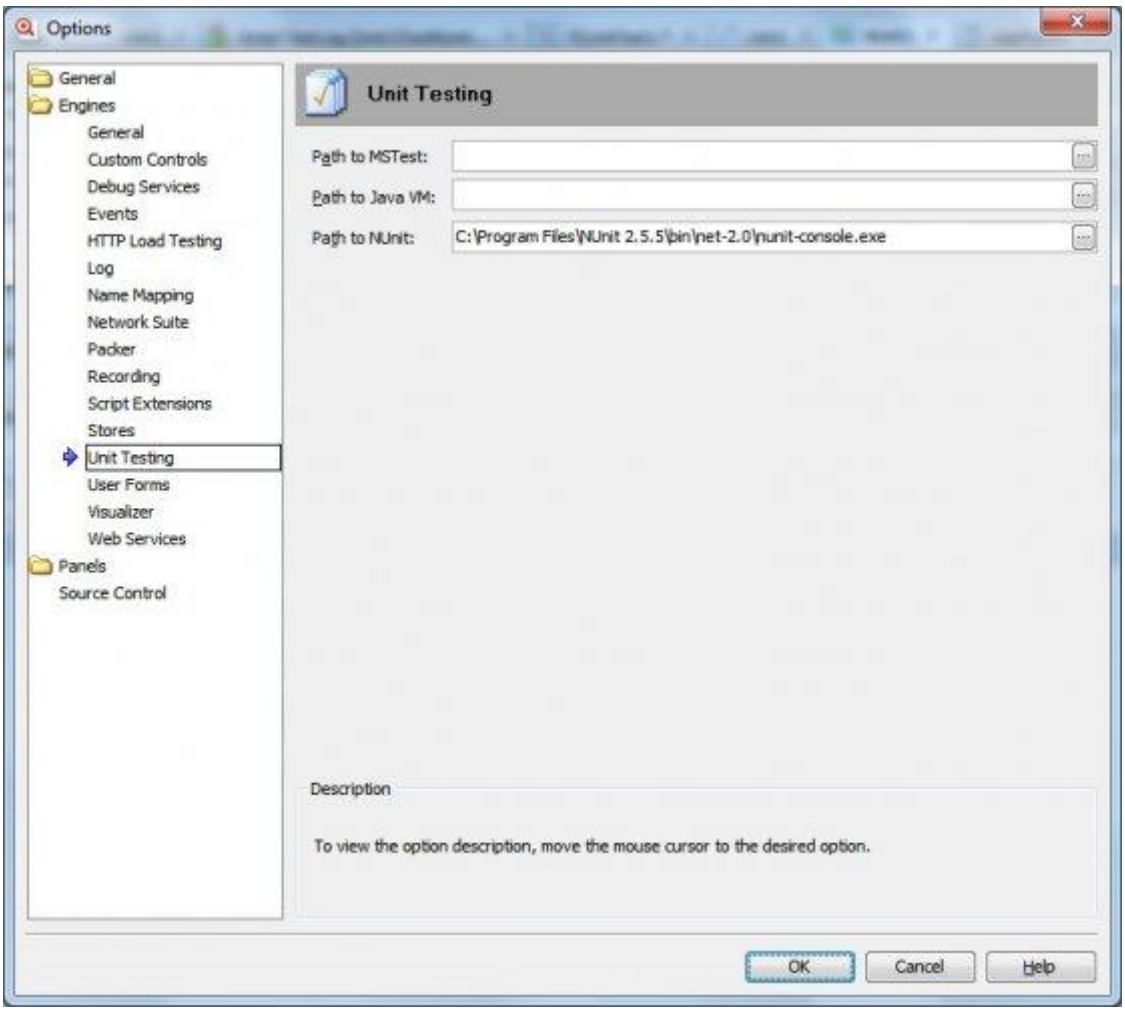
```
    [Test]
    public void PlusTest2()
    {
        Assert.AreEqual(Form1.PlusMethod(5, 6), 11);
    }
    [Test]
    public void PlusTest3()
    {
        Assert.AreEqual(Form1.PlusMethod(10, 3), 5);
    }

}
```

Обратите внимание, что последний тест изначально задан неверно (10 + 3 = 13, а не 5, как ожидается в тесте). Это сделано специально, чтобы сымитировать ошибку в юнит-тесте.

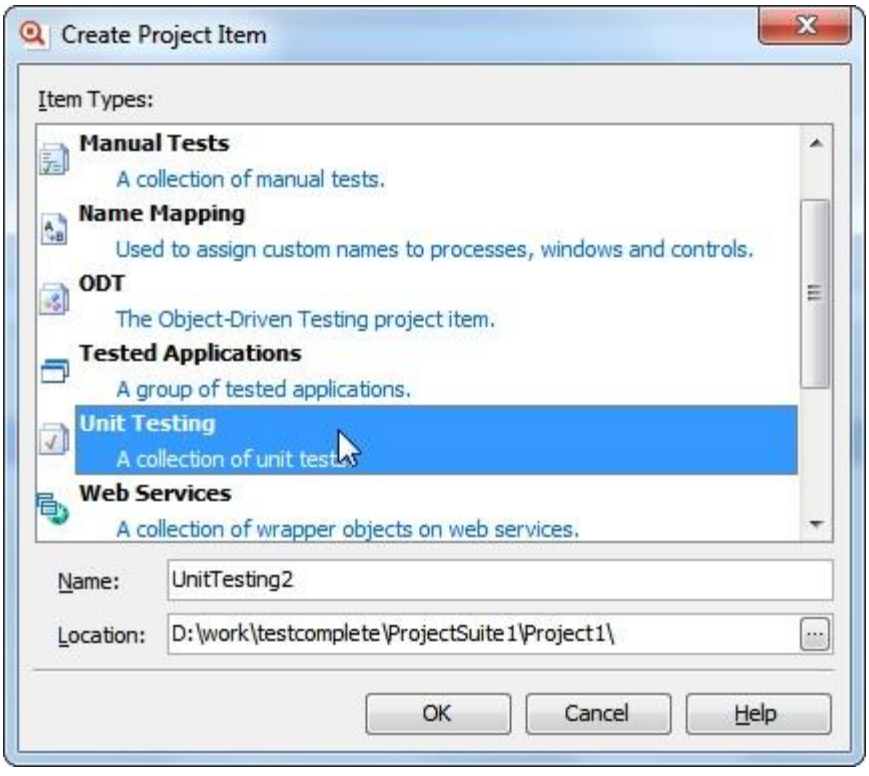
Запуск NUnit тестов с помощью внешней утилиты

Для запуска юнит-тестов используется утилита командной сроки nunit-console.exe.

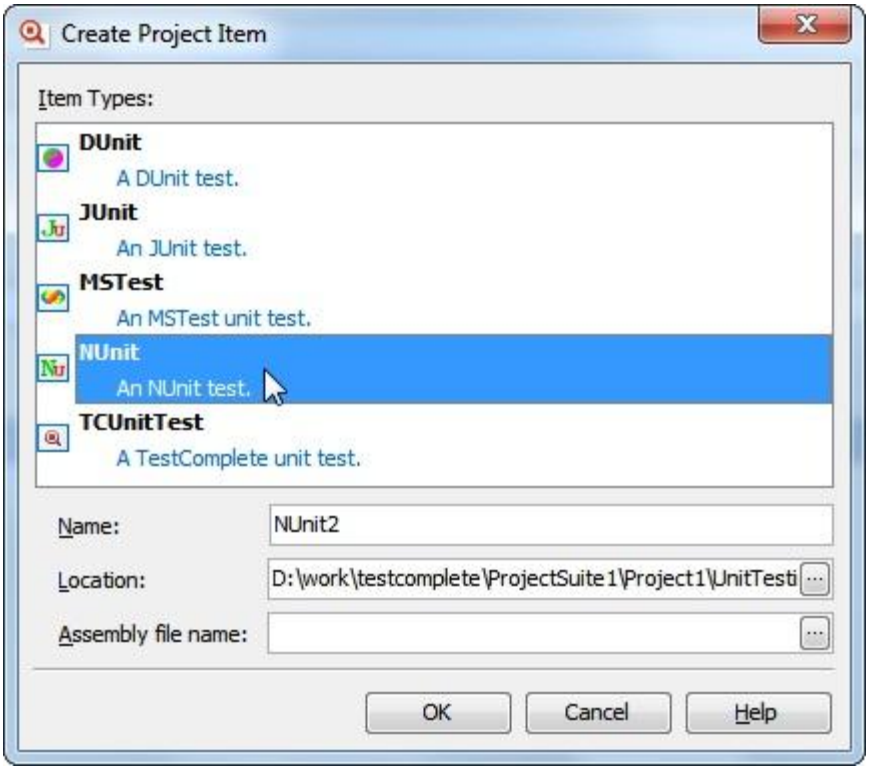
Прежде всего, необходимо указать путь к этой утилите в настройках TestComplete (Tools – Options – Engines – Unit Testing). Естественно, что для этого NUnit должен быть установлен на вашем компьютере.



Теперь добавим в проект элемент UnitTesting (правой кнопкой на имени проекта, Add – New Item – Unit Testing).

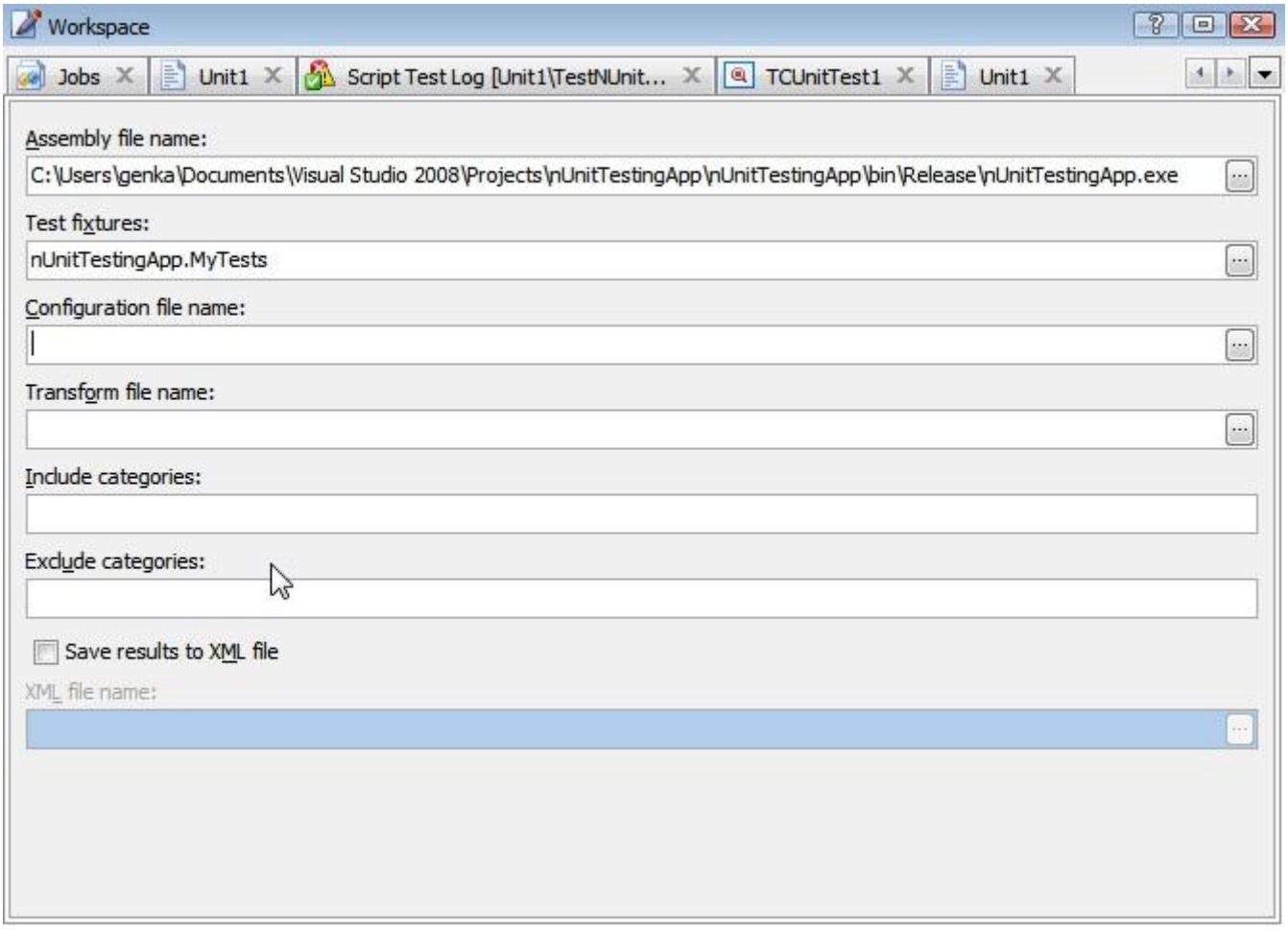


Затем правой кнопкой на добавленном элементе UnitTesting1, Add – New Item, и выберем элемент NUnit.



Теперь дважды щелкнем по добавленному элементу NUnit1 и заполним необходимые поля. В нашем случае это первые два поля:

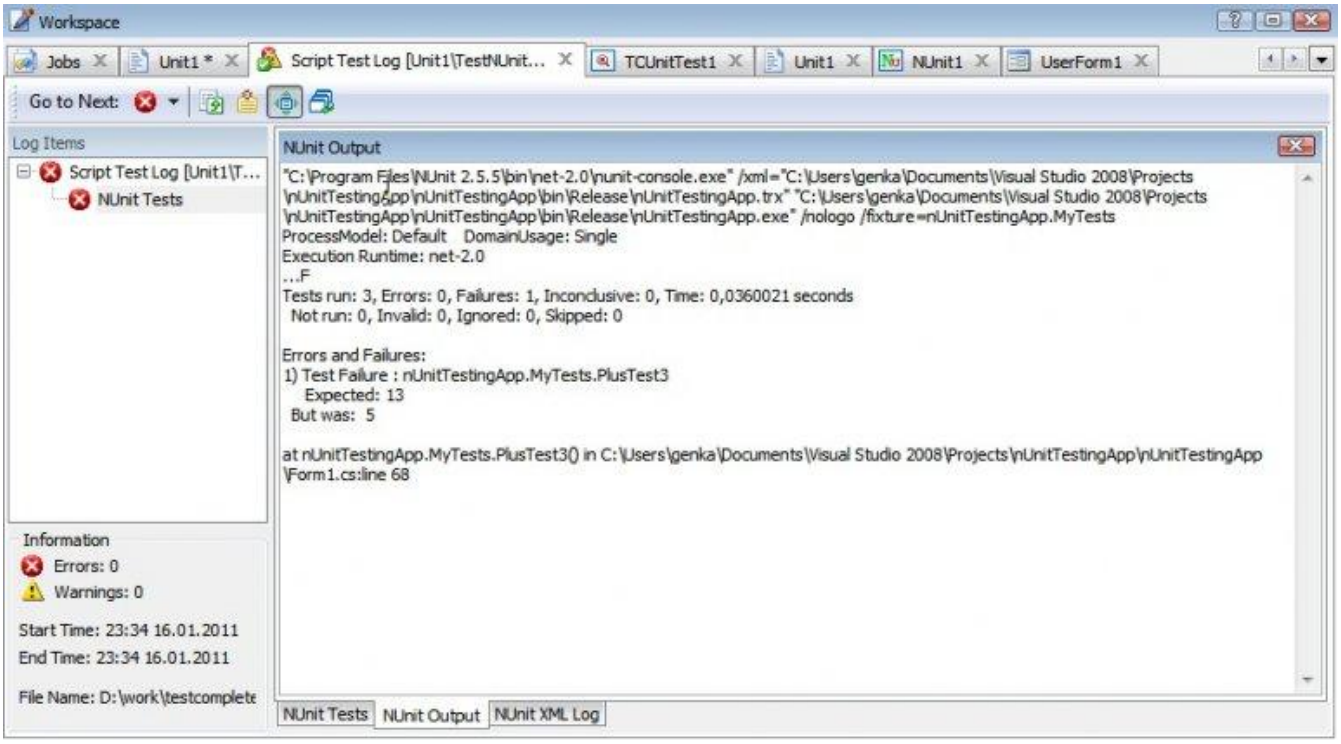
- Assembly file name – имя exe либо dll файла с юнит тестами (в нашем случае это exe-файл приложения)
- Text fixtures – тестовые классы (это необязательное поле, по умолчанию будут использованы все классы с атрибутом TestFixture)



Теперь мы можем запускать модульные тесты прямо из скриптов TestComplete. Например:

```
function TestNUnit()
{
  var ut = UnitTesting1.NUnit1;
}
```

В результате мы получим следующий лог:



Здесь можно прочитать, в каком именно тесте произошла ошибка, какое было ожидаемое и реальное значение и т.п. Эта информация предоставляется приложением NUnit.

Создание юнит тестов TCUnitTest

Кроме использования внешней утилиты, можно дать возможность TestComplete-у получать доступ к юнит тестам непосредственно через приложение.

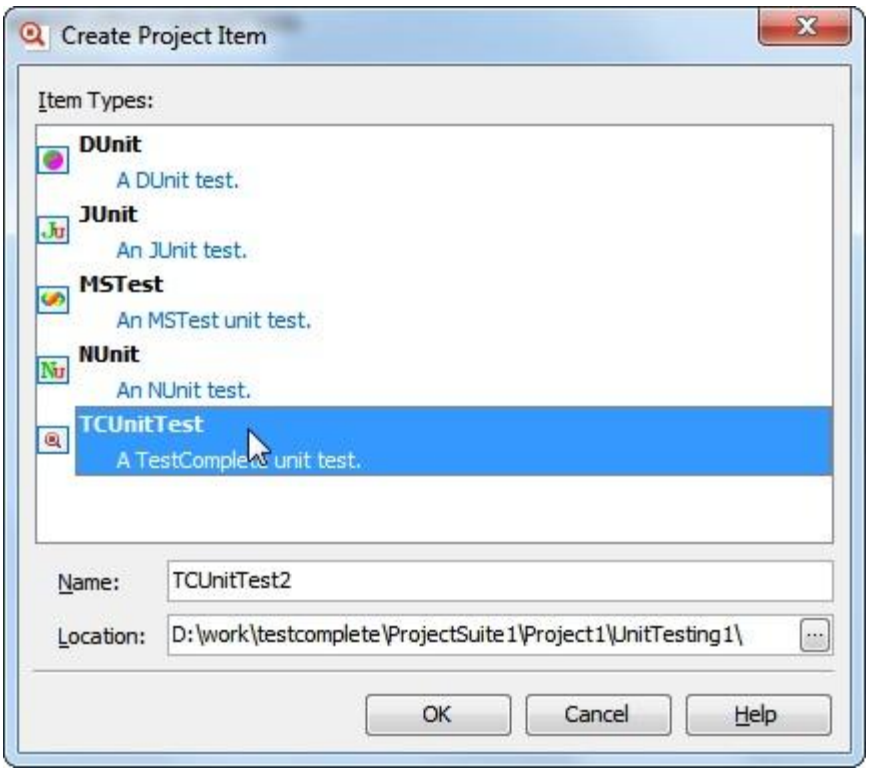
Для этого необходимо в приложении добавить ссылку (Reference) на сборку

<TestComplete>\Bin\Extensions\AutomatedQA.TestComplete.UnitTesting.dll

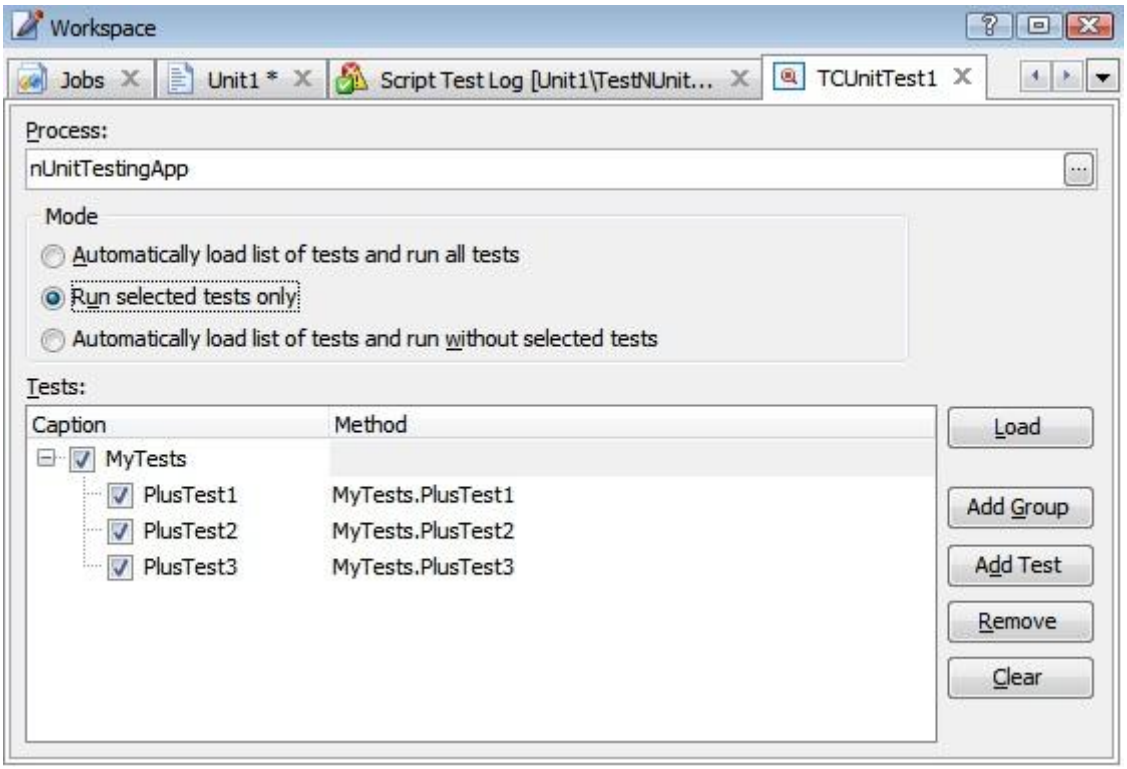
установить свойство этой сборки Copy Local = True, а затем предоставить возможность TestComplete-у получать доступ к тестовым методам. Для этого необходимо воспользоваться методом AddClasses объекта UnitTesting:

```
Type[] typearr = { typeof(MyTests) };

UnitTesting.AddClasses(typearr);
```

Теперь добавим новый юнит тест (правый щелчок на элементе UnitTesting1, Add – New Item – TCUnitTest).



Запустите тестовое приложение и откройте в TestComplete добавленный тест. В первом поле необходимо выбрать процесс приложения (в нашем случае это nUnitTestingApp), а также одну из опций запуска.

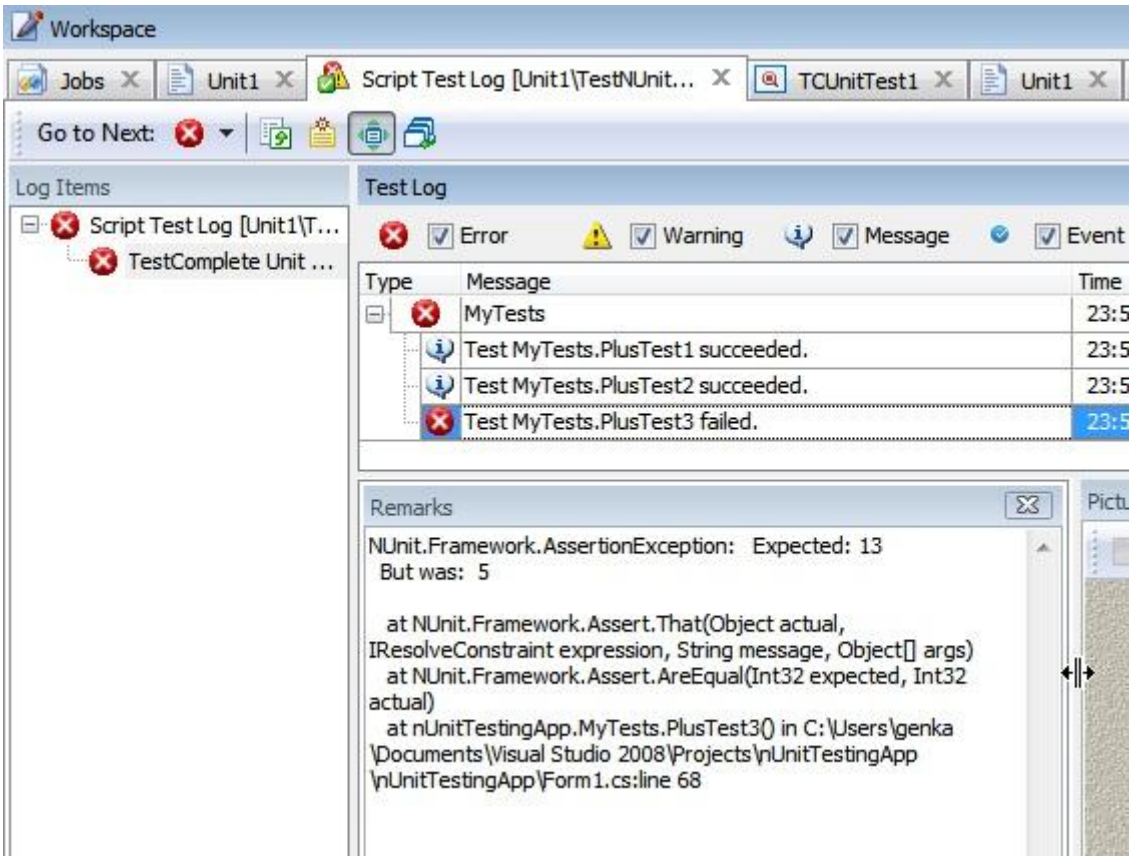


Опции:

- **Automatically load list of tests and run all tests** – запускает все доступные TestComplete-у тесты
- **Run only selected tests** – запускает только выделенные в редакторе тесты

- **Automatically load list of tests and run without selected tests** – запускает все тесты, кроме выделенных в редакторе

Результат запуска этого теста (запуск осуществляется таким же способом, как и в предыдущем случае, с помощью метода Execute):



Как видите, у этого способа есть преимущество перед использованием утилиты командной строки: результаты показаны в привычном для пользователя TestComplete виде. Однако есть и недостаток: в случае использования этого метода необходимо сначала запустить тестовое приложение.

Подробнее о других видах юнит тестов (JUnit, DUnit и т.д.) можно прочитать в справке по TestComplete, раздел Unit Testing Project Item.

18 Полезные объекты TestComplete

Возможности языков, поддерживаемых TestComplete, огромны, однако их не всегда хватает при написании скриптов. Для того, чтобы расширить возможности встроенных языков, а также чтобы предоставить доступ к некоторым специфичным возможностям приложения, в TestComplete были введены несколько очень полезных объектов, которые мы сейчас и рассмотрим.

Мы не будем рассматривать все возможности этих объектов и не будем рассматривать их на примерах, а дадим лишь общие описания объектов и некоторых наиболее часто используемых методов.

Общее правило при работе с TestComplete таково: если вам нужна функция, которой нет в языке программирования, который вы используете, сначала посмотрите, нет ли такой функции в уже готовых объектах TestComplete, и лишь затем приступайте к написанию своей (если в этом есть необходимость).

Объект Sys

Sys – это объект, через который предоставляется доступ ко всем процессам. Кроме того, с помощью объекта Sys можно получить информацию о системе. Например, информация об операционной системе (Sys.OSInfo), доступ к буферу обмена (Sys.Clipboard) и всему экрану (Sys.Desktop), имя компьютера и домен/рабочую группу (Sys.HostName и Sys.DomainName). С помощью объекта Sys.OleObject можно получить доступ к любому COM-объекту.

Объект Runner

Предназначен для управления ходом выполнения скриптов непосредственно из самих скриптов. Полезные методы: **Start** (запускает тесты из текущего проекта); **Stop, Halt** (останавливают выполнение скриптов, метод Halt позволяет при этом запостить в лог сообщение об ошибке); **Pause** (приостанавливает выполнение скрипта и активирует отладчик); **CallMethod** (запуск функции из другого модуля), **CallObjectMethodAsync** (позволяет запустить методы приложения асинхронно).

Особый интерес здесь представляет метод CallObjectMethodAsync, который позволяет не просто обратиться к какому-то методу тестируемого приложения, а запустить его асинхронно, т.е. запустить и продолжить выполнение скрипта, не дожидаясь окончания работы запущенного метода.

Обратите также внимание на то, что в метод CallMethod передается полное имя функции (т.е. имя в виде "имя_модуля.имя_функции", причем это должна быть строка, например "Unit1.MyFunction").

Объект BuiltIn

Большинство методов объекта BuiltIn считаются устаревшими и оставлены только для совместимости со скриптами, написанными в более ранних версиях TestComplete. Новые версии методов можно найти в объектах aqFile, aqConvert, aqObject и прочих.

Вот некоторые методы объекта BuiltIn, на данный момент не считающиеся устаревшими:

MessageDlg, InputBox, ShowMessage, InputQuery (позволяют отобразить на экране небольшие диалоговые окна разных типов для ввода и/или отображения информации)

ParamCount , ParamStr – возвращает количество параметров и сами параметры, переданные в TestComplete в командной строке

SendMail – позволяет отправить e-mail из скриптов

CreateVariantArray, CreateVariantArray2, CreateVariantArray3, VarArrayRedim, VarArrayHighBound, VarArrayLowBound – позволяют работать с Variant-массивами (т.е. массивами такого типа, которые используются в языке VBScript)

Остальные методы будут интересны только пользователям TestComplete версии 6 и ниже, так как в этих версиях программы еще нету объектов aqFile, aqEnvironment и т.п.

Объект Options

Это очень полезный объект, позволяющий во время работы скриптов менять некоторые настройки TestComplete и проекта, такие как используемая модель объектов в веб-приложениях, параметры изображений (формат, качество и т.п.), включение/выключение лога, переменные проекта (project variables) и пр. Пример использования объекта Options мы приводили в главе 4.1 Функциональное тестирование Web-приложений.

Объекты Project и ProjectSuite

Объекты **Project и ProjectSuite** предоставляют доступ к текущему проекту и набору проектов, позволяя определить их параметры (например, путь – свойство Path, имя – свойство FileName, глобальные переменные – свойство Variables и др.).

Объект aqUtils

Предоставляет различные дополнительные возможности, которых нет в других объектах, например издание звукового сигнала с помощью PC Speaker-а (метод **Beep**) и задержка выполнения скрипта на определенное время (**Delay**).

Обратите внимание, что в более ранних версиях метод Delay (который почему-то часто используется) находился в объекте BuiltIn, однако сейчас им не рекомендуется пользоваться (одна из причин – метод BuiltIn.Delay нельзя использовать при написании собственных надстроек, Extensions).

Использовать метод Delay рекомендуется только в самых крайних случаях, когда не получается обойтись методами Find, Wait и т.п. В подавляющем большинстве случаев можно обойтись без метода Delay.

Объект aqEnvironment

Предоставляет доступ для работы с плагинами TestComplete и операционной системой. Один из наиболее интересных методов этого объекта – это метод RebootAndContinue,

позволяющий перезагрузить компьютер и продолжить выполнение скрипта с того самого места.

Объект aqObject

Предназначен для работы с оконными объектами (элементами управления в тестируемых приложениях). С его помощью можно вызвать метод объекта (**CallMethod**), получить список всех свойств и методов объекта (**GetProperties, GetMethods** и т.п.), проверить, поддерживается ли какой-либо метод объектом (**IsSupported**) и многое другое.

Объект aqConvert

Позволяет конвертировать данные одного типа в другой (например, **StrToDate, DateTimeToStr** и т.п.)

Объект aqDateTime

Предоставляет методы для работы с датой и временем

Объект aqString

Содержит большое количество методов для работы со строками

Объекты для работы с файловой системой

Несколько объектов для работы с файлами различных типов, папками и дисками: **aqBinaryFile, aqDriveInfo, aqFile, aqFileInfo, aqFileCertificateInfo, aqFileSystem, aqFileVersionInfo, aqFolderInfo, aqTextFile.**

19 Настройки TestComplete

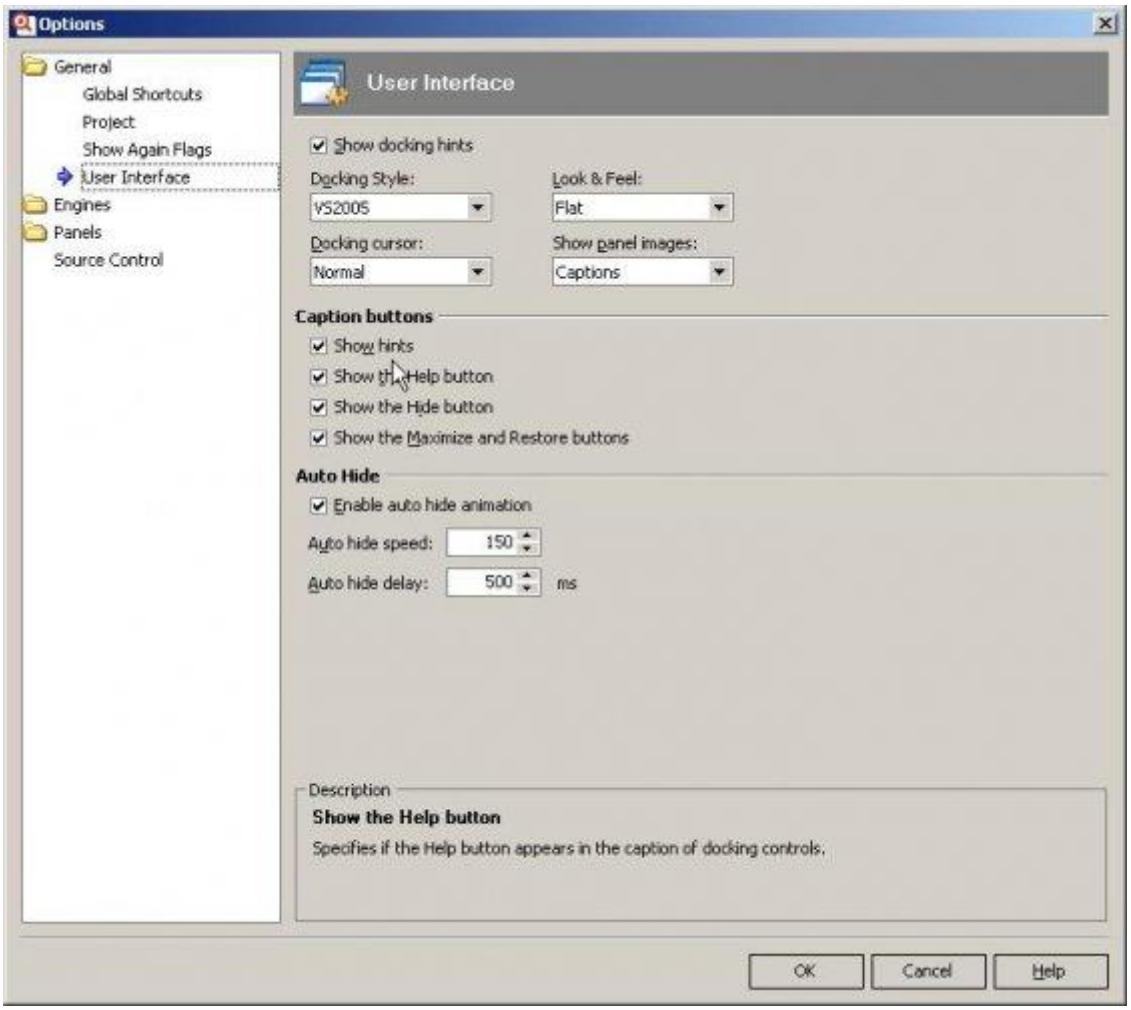
TestComplete обладает огромным количеством настроек (как интерфейса, так и функциональности). С помощью различных настроек можно существенно облегчить работу с TestComplete. Надо лишь помнить, что для большинства возникающих в ходе автоматизации задач в TestComplete кже есть решение, вам надо лишь найти его.

Мы не будем рассматривать все настройки подробно, а остановимся лишь на наиболее интересных и полезных возможностях, которые могут облегчить работу или существенно влияют на работу с приложением.

19.1 Настройка интерфейса TestComplete

Внешний вид TestComplete

Общий вид окна TestComplete можно настроить в меню *Tools – Options – General – User Interface.* Здесь можно выбрать стиль отображения панелей, окон и т.п.

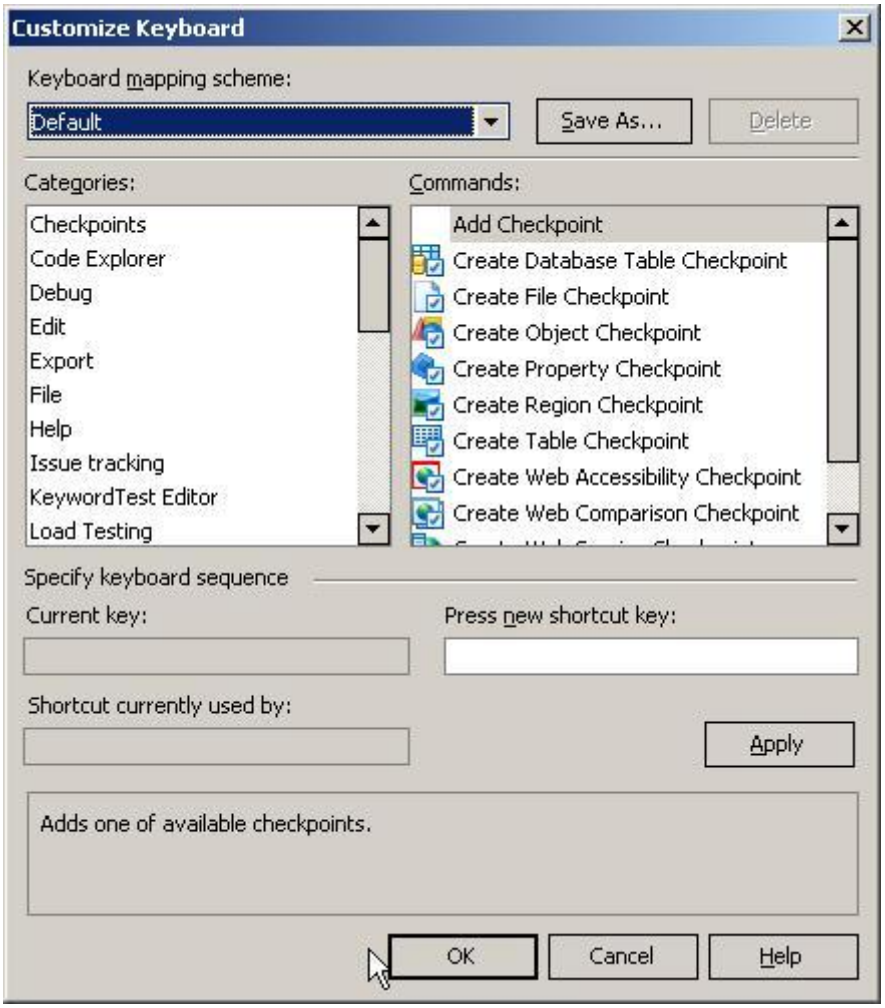


Любая панель в TestComplete может быть либо плавающей (выглядит как маленькое окошко), либо закреплена (выглядит как вкладка). Все панели можно перемещать и располагать как угодно.
Если вы захотите вернуть панелям TestComplete изначальный вид, воспользуйтесь командой меню *View – Desktop – Restore Default Docking.*
Точно так же можно настраивать команды панелей управления. Для этого необходимо

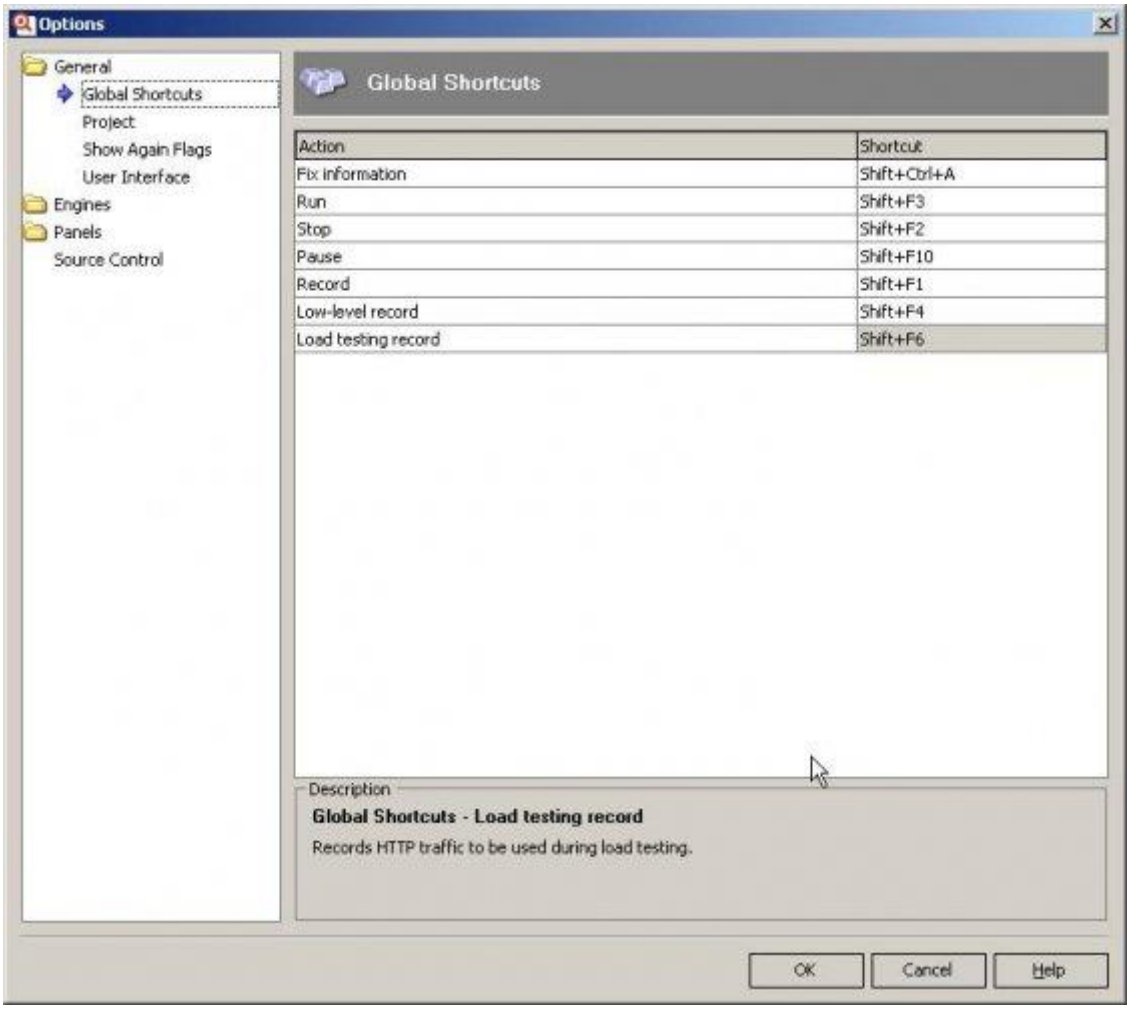
щелкнуть правой кнопкой мыши на панели инструментов и выбрать пункт меню *Customize* (или выбрать пункт меню *Tools – Customize Toolbar*).
Для восстановления первоначального вида панелей инструментов необходимо выбрать пункт меню *View – Toolbars – Restore Default Toolbar.*
Кроме того, через пункты меню *View – Desktop и View – Toolbars* можно сохранить текущий вид панелей и панелей инструментов, а также загрузить сохраненный ранее вид.

Редактор

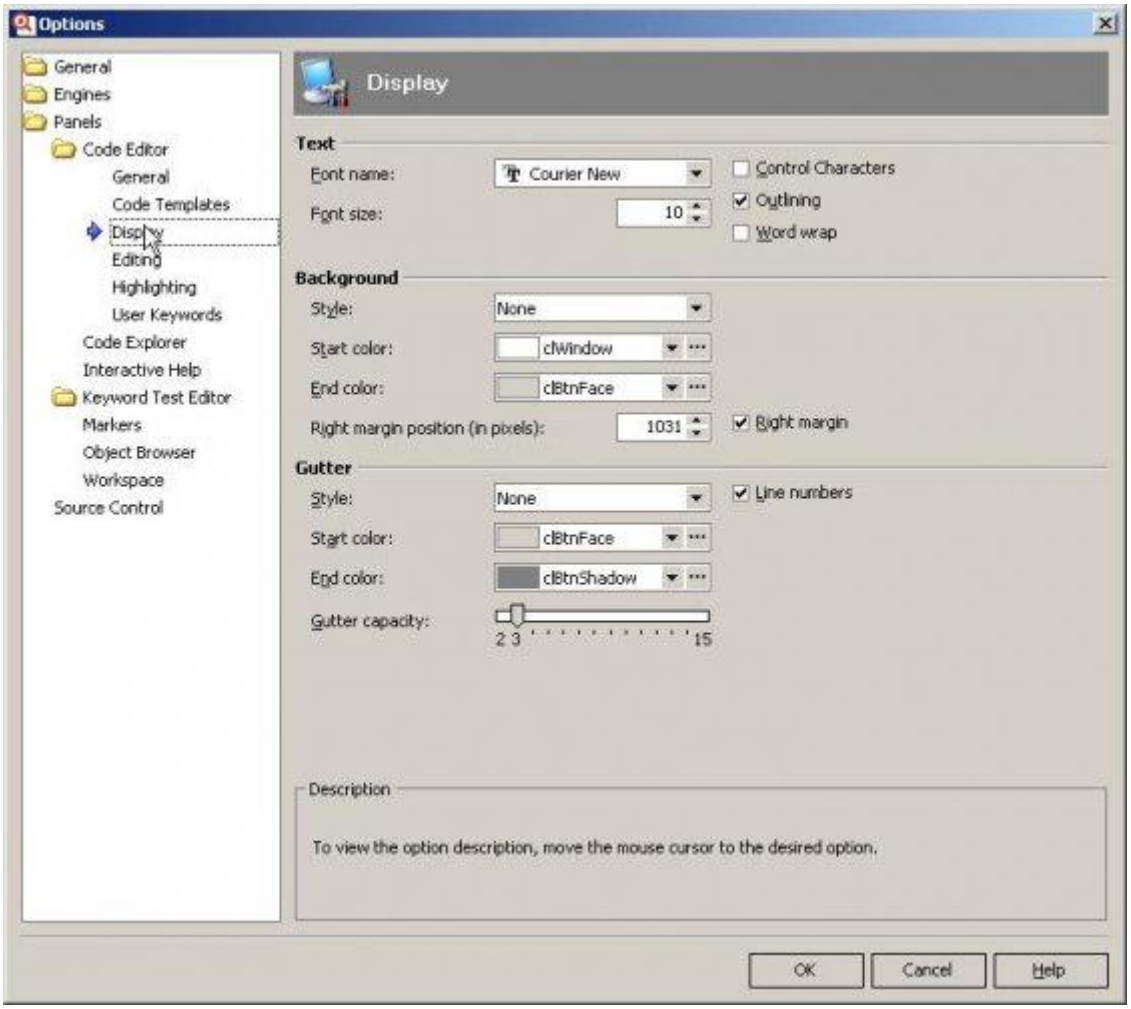
Редактор кода TestComplete также очень гибко настраивается.
В меню *Tools – Customize Keyboard* можно загрузить одну из предопределенных настроек клавиатурных комбинаций (Borland или Visual Studio), или полностью переопределить клавиатурные комбинации для любого действия.



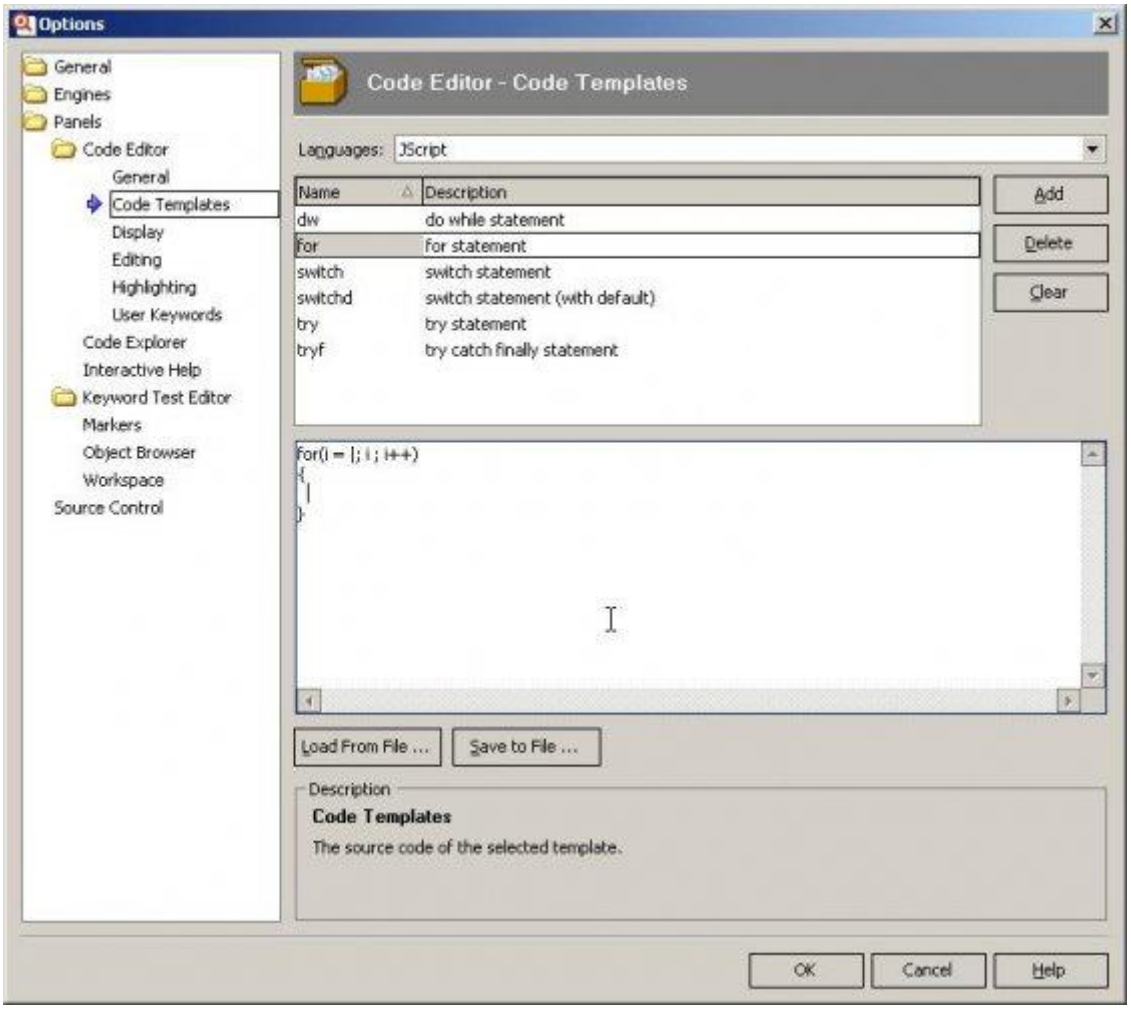
Кроме этих комбинаций клавиш можно также настроить горячие клавиши (т.е. глобальные клавиатурные сочетания, которые будут работать даже тогда, когда окно TestComplete неактивно) для некоторых действий. Сделать это можно в меню *Tools – Option – General – Global Shortcuts.*



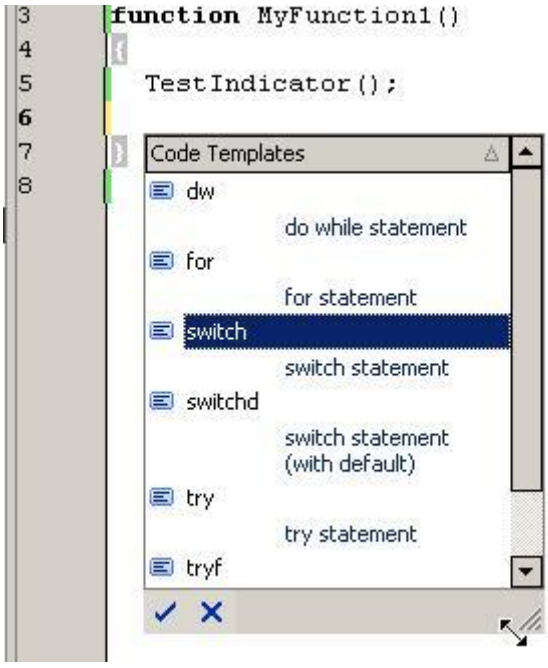
Большое количество настроек редактора можно найти в разделе *Tools – Options – Panels – Code Editor*.



Здесь можно настроить внешний вид редактора и подсветку различных элементов кода (разделы Display, Editing и Highlighting).
В разделе User Keywords можно добавить свои слова, которые будут отображаться в редакторе синим полужирным шрифтом (как, например, слова Sys и BuiltIn).
Если вы пишете много кода и часто пользуетесь определенными конструкциями (например, циклы, условия и т.п.), то вам необязательно каждый раз вводить их все вручную или копировать откуда-то. Для этого есть Шаблоны кода (Code Templates).

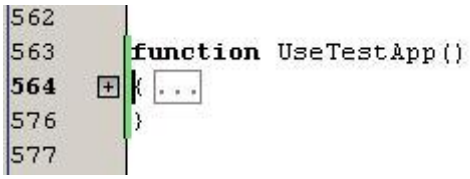


Вам достаточно нажать комбинацию клавиш **Ctrl-J** в редакторе и выбрать необходимый шаблон из списка.



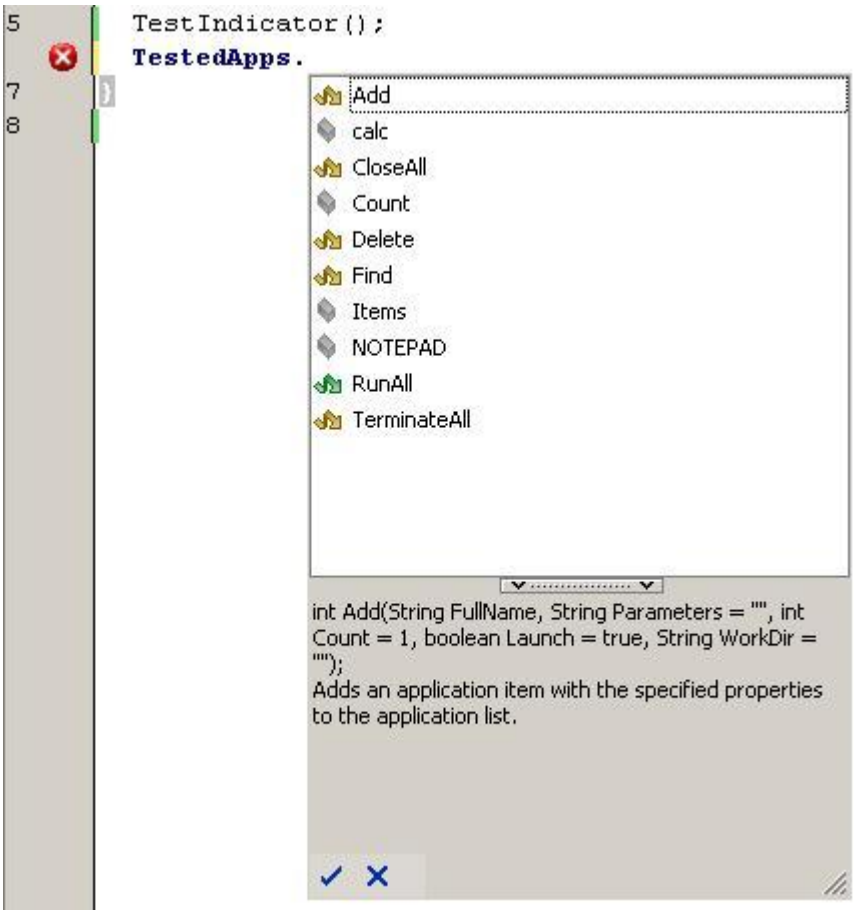
Наборы шаблонов можно сохранять в отдельные файлы и загружать сохраненные ранее шаблоны.

В самом редакторе кода можно сворачивать блоки, которые используются нечасто (например, комментарии к функциям, подробно описывающие работу функции). Для этого необходимо выделить блок кода, который вы хотите спрятать, щелкнуть по нему правой кнопкой мыши и выбрать пункт *Outlining – Hide Selection.* После чего этот блок будет выглядеть в редакторе так:

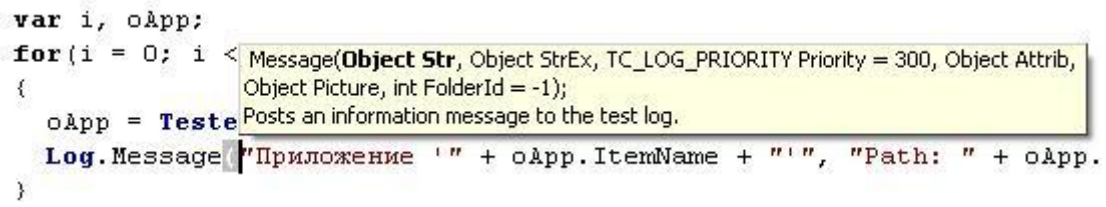


Если вам надо быстро перейти к функции, вызов которой вы видите на экране, просто зажмите клавишу **Ctrl** и щелкните левой кнопкой мыши на имени этой функции

или щелкните правой кнопкой мыши на имени и выберите пункт меню Go to Declaration. Чтобы вызвать появление списка автозаполнения для какого-то объекта, нажмите комбинацию клавиш **Ctrl+пробел.**



Чтобы посмотреть на список параметров какой-либо функции, поставьте курсор сразу за открывающей скобкой и нажмите **Ctrl-Shift+пробел.**



19.2 Настройка параметров TestComplete

Теперь рассмотрим наиболее часто используемые настройки TestComplete, которые влияют не на внешний вид, а на функциональность приложения. Все эти настройки находятся в окне **Options** (меню *Tools - Options*), поэтому дальше речь будет вестись именно о нём. Мы не будем рассматривать все настройки (так как многие из них могут вам никогда и не понадобиться), а рассмотрим лишь те, которые могут облегчить работу с TestComplete или другие настройки, которые часто используются независимо от типа тестируемого приложения.

General – Show Again Flags

В этой группе настроек можно включить или отключить различные оповещения TestComplete (например, проверка обновлений, создавать ли автоматически псевдонимы (Aliases) для маппируемых объектов, показывать ли сообщение об ошибке инициализации MS Script Debugger и т.п.).

Engines – General

Show hidden properties – позволяет отобразить в Object Browser-е скрытые свойства и методы. Эту опцию рекомендуется всегда включать.

Images (кнопка Configure) – здесь вы можете настроить формат файлов, которые будут использоваться при работе с изображениями, в том числе при помещении картинки в лог. Формат BPM является самым точным, однако BMP-файлы занимают много места на диске. Формат JPEG самый экономный с точки зрения занимаемого места на диске, однако его качество гораздо хуже. Для большинства случаев рекомендуется использовать формат PNG, который с одной стороны занимает немного места, а с другой стороны довольно точный.

Object Naming – в этом разделе задаются правила формирования имен окон.

Auto-correct Afx windows и Auto-correct window captions автоматически заменяют на звездочку некоторые части имен окон, которые могут меняться (например, заголовок окна "Untitled - Notepad" не всегда будет содержать слово Untitled в начале, поэтому его лучше заменять на звездочку).

Use short names when possible позволяет в некоторых случаях заменять длинные обращения к элементам управления на более короткие (например, frmMain вместо VCLObject("frmMain")). С одной стороны это более удобно, а с другой стороны

TestComplete-у приходится каждый раз при обращении к такому объекту определять его тип, поэтому в случае использования коротких имен могут быть проблемы при запуске скриптов (например, иногда TestComplete не будет находить нужные объекты на экране).

Engines – HTTP Load Testing

В этом разделе задаются параметры нагрузочного тестирования, подробнее о них можно прочитать в главе 4.2 Нагрузочное тестирование Web-приложений

Engines – Log

Activate after test run – отображать лог после того, как скрипты завершат работу.
Show Log on Pause – отображать лог во время паузы в режиме отладки
Store all logs – хранить ли все логии или лишь определенное их количество.

Engines – Name Mapping

Map objects names automatically – позволяет автоматически создавать имена в Name Mapping-е во время записи скриптов. Если вы не пользуетесь Name Mapping-ом, то эту опцию лучше отключить.

Engines – Recording

Minimize TestComplete – сворачивать TestComplete во время записи скриптов.
Real-time mode – записывать скрипты в "реальном времени", т.е. со всеми паузами между выполнением действий. Включение этой опции бывает не так уж часто.
Do not generate variables – не создавать автоматически переменные, а всегда использовать полный путь к объектам (например, `Sys.Process(...).Window(...).Window(...).Click()` вместо `Notepad.Window(...).Click()`).
Smart variables names – позволяет включить/отключить "умное" именование объектов (например, wndNotepad вместо w1) при записи скриптов. "Не умное" именование использовалось в более ранних версиях TestComplete.

Engines – Stores

Perform the following actions instead of comparing. Эта группа переключателей позволяет обновить данные, хранящиеся в Stores, новыми значениями, вместо того, чтобы осуществлять сравнение. Это может быть полезно в том случае, если сразу во многих местах приложение изменилось и необходимо обновить все или многие объекты, хранящиеся в Stores. При включении этих опций при вызове метода Compare данные в Stores будут заменены на новые. Для этого вам необходимо просто запустить скрипты, которые выполняют проверку, и данные будут обновлены.

Обратите внимание на 2 важных фактора:

1. При обновлении данных в Stores старые данные будут уничтожены безвозвратно
2. Не забудьте вернуть настройки обратно после того, как все данные обновятся, иначе при следующих запусках они будут продолжать обновляться вместо того, чтобы выполнять проверку

Engines – Visualizer

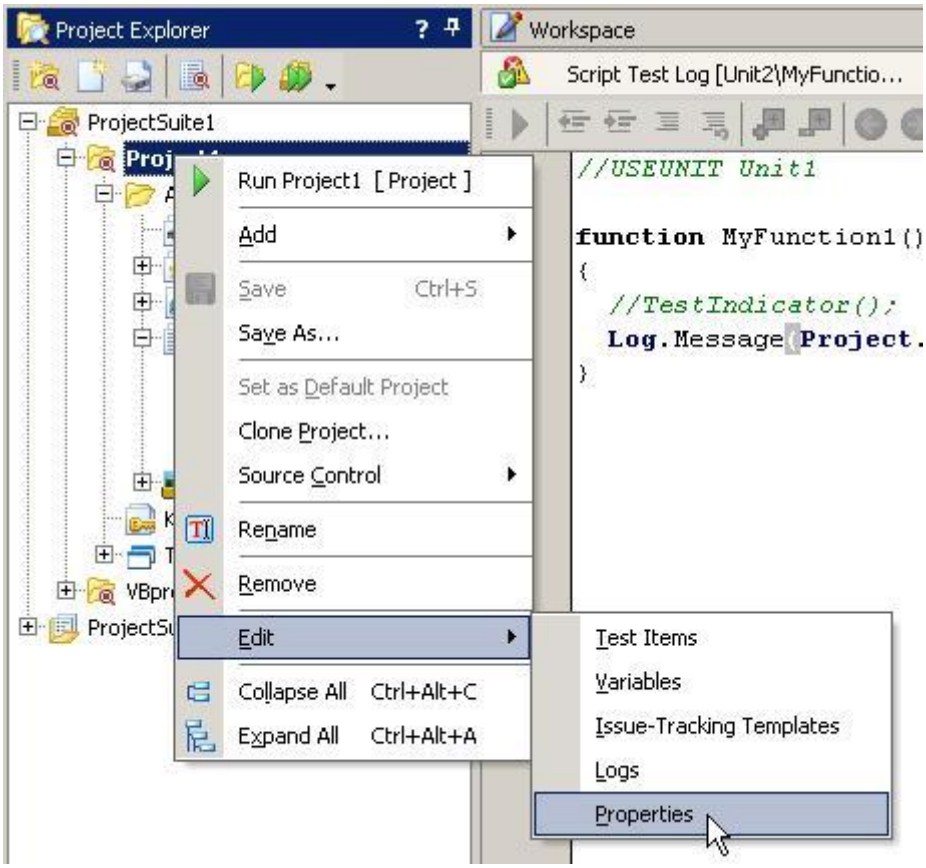
Здесь можно изменить опции Визуализатора (подробнее о нем читайте в главе 11.10 Использование Визуализатора).

Panels – Object Browser

Minimize TestComplete – сворачивает окно TestComplete при открытии окна Object Properties.

19.3 Настройки проекта

Теперь поговорим о наиболее интересных настройках проекта. Чтобы увидеть настройки проекта в TestComplete, необходимо щелкнуть правой кнопкой мыши на имени проекта в дереве слева и выбрать пункт меню *Edit – Properties.*



При этом в правой части окна TestComplete отобразится панель с деревом различных настроек.

General

Object Tree Model – модель дерева объектов (Tree или Flat). В зависимости от того, какую модель вы выберите, по-разному будет представлена иерархия объектов в Object Browser-е. Подробнее оо объектных моделях можно узнать из главы 3.1 Выбор модели объектов
Log location – папка, в которой хранятся логии запусков скриптов
Character encoding – кодировка. используемая в модулях TestComplete. Рекомендуется оставить значение Auto (при этом используется кодировка UTF-8).

Object Mapping

На этой страничке вы можете ассоциировать нестандартные элементы управления со стандартными. Например, если в вашем приложении есть текстовые поля, которые TestComplete определяет как неизвестный объект, можно ассоциировать класс этого объекта с наиболее подходящим текстовым полем и TestComplete будет знать, как с ним работать. Подробнее о процессе создания ассоциаций элементов управления можно прочитать в главе 11.9 Ассоциации объектов (mapping)

Open Applications – General

Use native objects names for TestComplete objects names – если эта опция включена, TestComplete генерирует имена объектов, используя их внутренние имена, т.е. имена, которые используются в самом приложении. Иначе TestComplete будет генерировать имена по имени класса и индексу элемента. Рекомендуется оставить эту опцию включенной.

Open Applications – Debug Agents

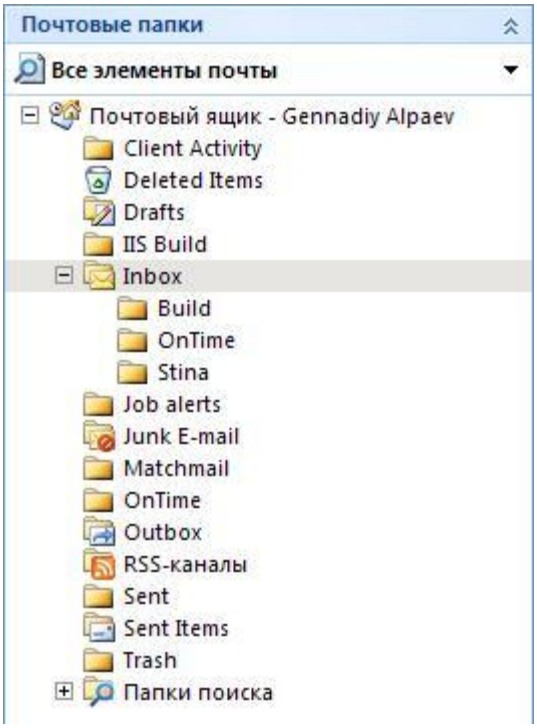
Debug информация приложения с одной стороны дает больше возможностей для доступа к внутренним свойствам и методам приложения, а с другой стороны – замедляет работу TestComplete с приложением. Включайте или отключайте чтение Debug информации в зависимости от того, используете ли вы ее в вашем тестируемом приложении или нет.

Open Applications – MSAA

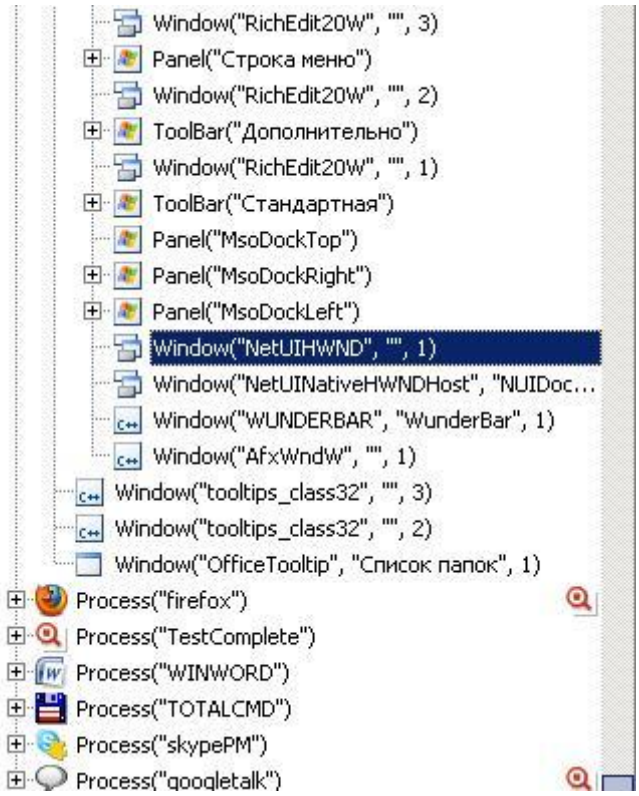
Open Applications – UI Automation

MSAA – это технология, позволяющая сторонним приложениям получать информацию о внутреннем устройстве приложения. Если в вашем приложении используется технология MSAA, это даст вам больше возможностей при работе с ним. В частности технология MSAA используется во всех приложениях Microsoft (например, MS Office). Для того, чтобы включить поддержку MSAA для какого-либо элемента управления, необходимо добавить имя его класса в раздел *Open Applications – MSAA*.

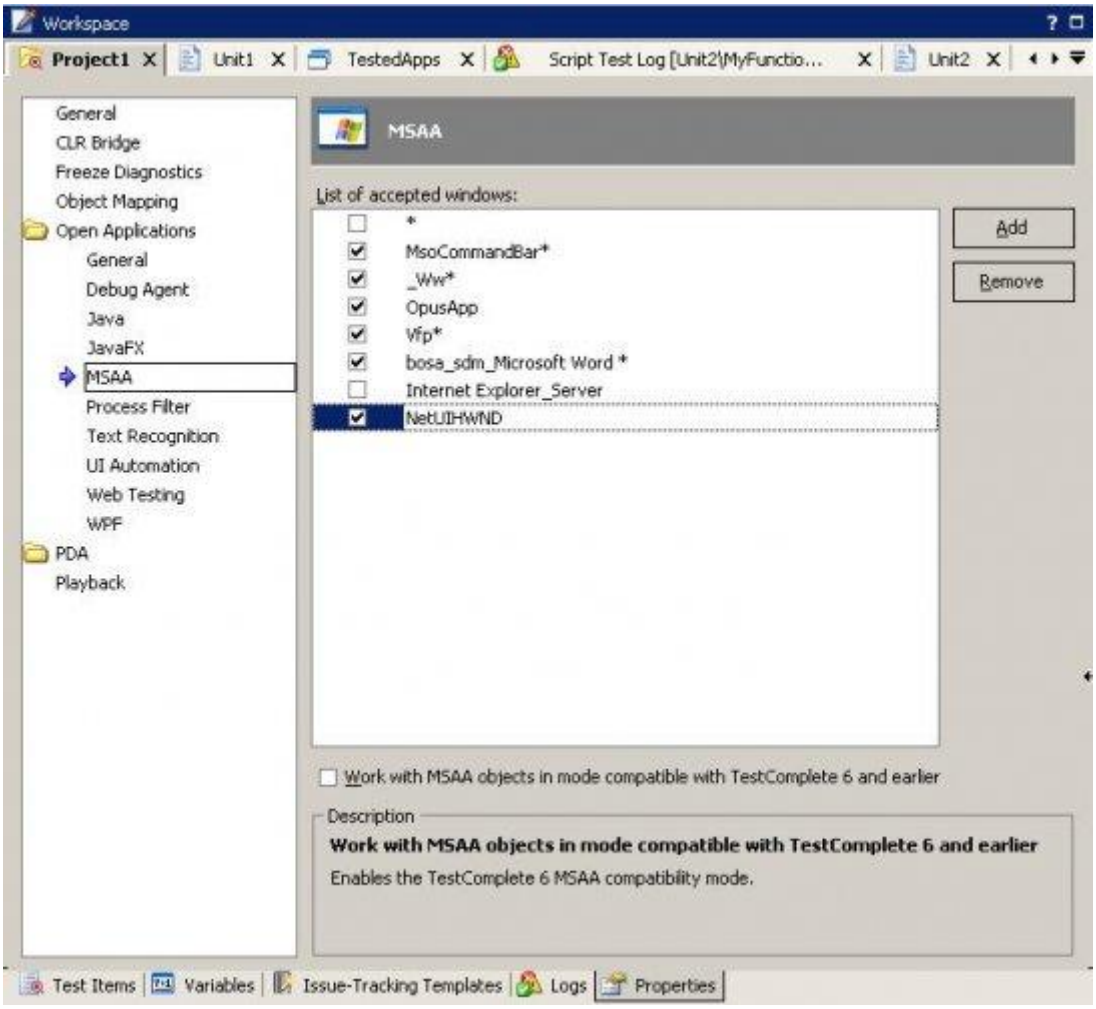
То же самое касается технологии UI Automation, которая используется в приложениях, написанных с использованием .NET 3.0 и выше.
В качестве примера приведем приложение MS Outlook. В левой части окна Outlook есть дерево с папками.



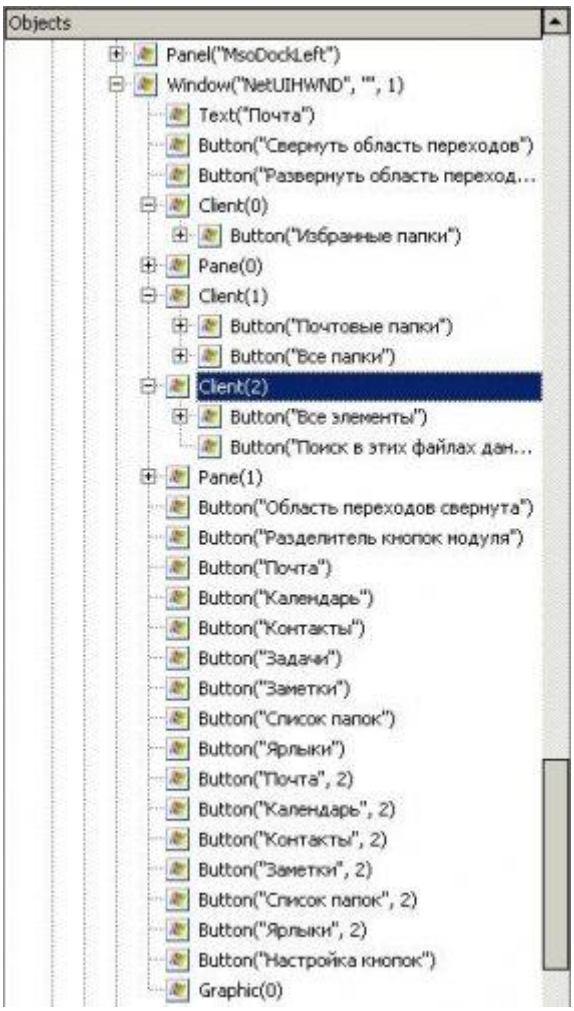
При использовании настроек по умолчанию этот элемент управления будет распознаваться TestComplete-ом так:



Однако если добавить в список MSAA класс NetUIHWND, как показано на рисунке ниже



то мы сразу же получим доступ ко внутренним элементам управления этой панели:



Open Applications – Web Testing

Tree Model – здесь устанавливается модель дерева объектов для веб-приложений. Подробнее о доступных моделях можно прочитать в главе 4.1 Функциональное тестирование Web-приложений
Identification attribute – атрибут, по которому TestComplete будет доступаться к элементам веб-приложений. Name более понятен и удобен, ID более универсален.
Tested host и Record tested host as – позволяют автоматически заменить при записи имя или часть имени сервера другим значением. Это может быть удобно в том случае, если скрипты записываются на одном компьютере, а воспроизводятся на другом, и при этом на разных компьютерах тестируемые веб-страницы находятся в разных местах (например, в разных папках). Тогда можно в поле Tested host ввести значение, которое будет заменяться (например, "http://localhost"), а в поле Record tested host as ввести имя переменной, которое будет подставляться вместо значения в поле Tested host (например, "Project.Variables.MyPath").
Make page object a child of the browser process – позволяет сделать объекты Page непосредственными потомками процесса браузера, что упрощает доступ к страницам, удаляя один "лишний" элемент в иерархии (или даже несколько элементов). Например, при отключенной опции Make page object... доступ к странице будет выглядеть так:

```
Sys.Process("firefox").Window("MozillaUIWindowClass", "Mozilla
Firefox", 1).Window("MozillaWindowClass", "",
4).Page("about:blank")
```

а при включенной опции так:

```
Sys.Process("firefox").Page("about:blank")
```

Ignore dynamic URL parameters – очень часто при работе с веб-приложениями в адресной строке передаются динамические параметры, т.е. такие параметры, которые меняются каждый раз для новой сессии. При записи скриптов TestComplete может заменять значения таких параметров на символ групповой замены (*), чтобы в дальнейшем не пришлось вручную модифицировать скрипты. Для этого достаточно лишь добавить имена параметров в список Dynamic parameters.

Playback

В этом разделе находятся параметры, влияющие на воспроизведение скриптов.

Stop on error, Stop on warning – остановить выполнение скриптов при возникновении ошибки или предупреждения
Minimize TestComplete – сворачивать окно TestComplete во время выполнения скриптов
Disable mouse – отключает мышь на время выполнения скриптов, тем самым сводя к минимуму возможность вмешательства пользователя. Полезно при использовании Low-level процедур
Auto-wait timeout – время ожидания элементов управления. В случае, когда время появление элементов управления сильно варьируется, рекомендуется использовать функции Wait (подробнее смотри главу 3.5 Синхронизация выполнения скриптов)
Delay between events – время задержки между событиями (клик мышки, ввод текста и т.п.). Увеличение этого параметра приводит к замедлению времени выполнения скриптов
Key pressing delay – интервал между нажатиями клавиш при использовании метода Keys. Увеличение этого параметра также приводит к замедлению выполнения скриптов, однако иногда его приходится увеличивать целенаправленно, если при вводе текста вводятся не все символы
Dragging delay – задержка при перетаскивании объектов (если быть точнее, то за сколько миллисекунд курсор мыши передвинется на 20 пикселей). Не рекомендуется ставить здесь слишком маленькое значение, иначе некоторые программы могут "не заметить" перетаскивания
Mouse movement delay – задержка при движении мыши. Если нет необходимости замедлять выполнение скриптов, это значение лучше установить равным нулю
On unexpected window
В этой группе указывается последовательность действий, которые предпримет TestComplete в случае появления непредвиденного окна. **Ignore unexpected window** – игнорировать окно, **Stop execution** – остановить выполнение скриптов, **Click on focused control** – кликнуть на элементе управления в окне, на котором установлен фокус (обычно это кнопка по умолчанию, закрывающая окно), **Press Esc/Enter** – нажать Esc/Enter, **Send WM_CLOSE** – отправить окну сообщение WM_CLOSE (команда закрытия окна).

Кроме того, можно написать свой обработчик непредвиденных окон (см. главу 11.8 Перехват событий). Еще один пример работы с появляющимися окнами можно найти здесь.

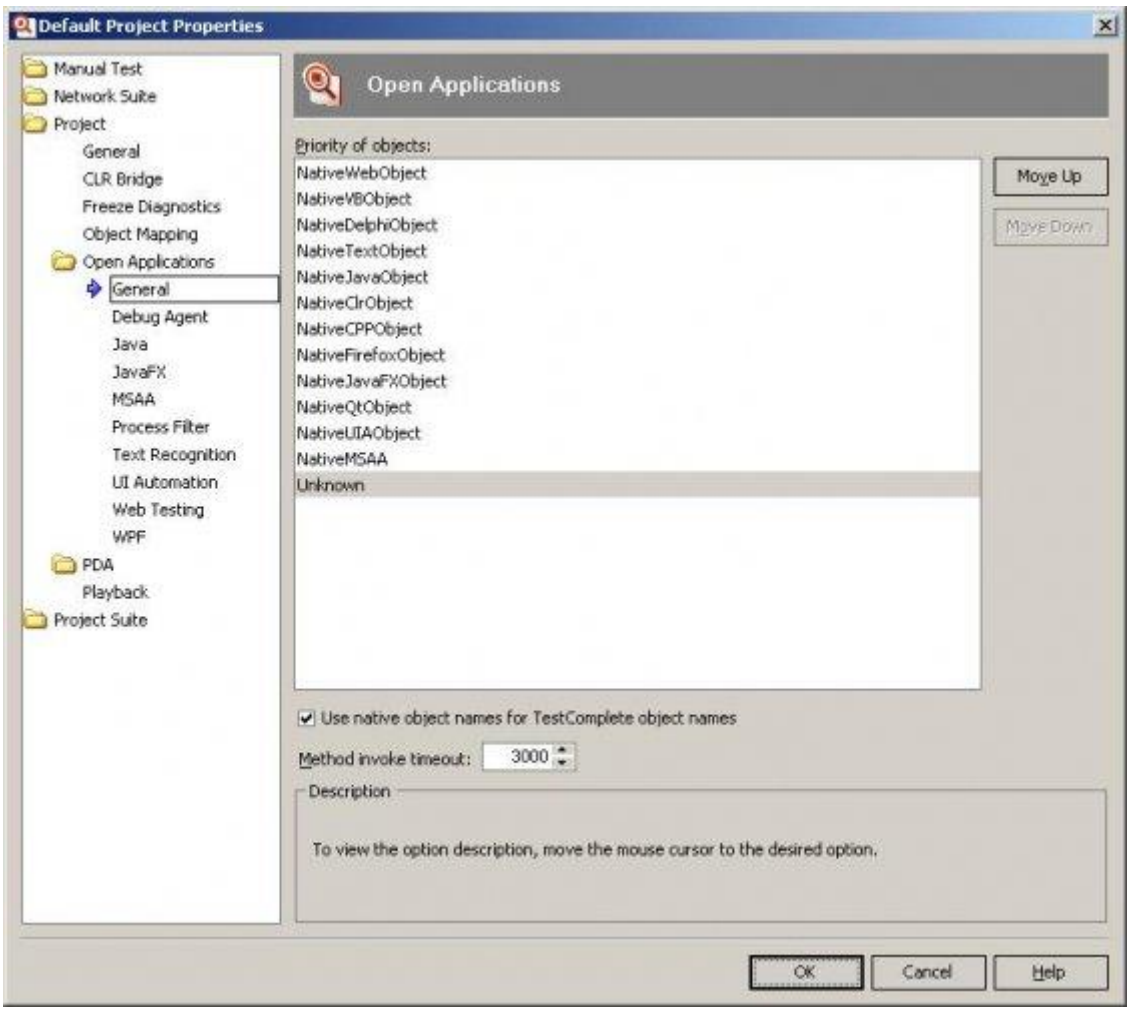
Ignore overlapping window – игнорировать перекрывающие окна
Post image on error – включите эту опцию, если хотите чтобы каждый раз при возникновении ошибки в лог помещался скриншот экрана
Save log every ... minutes – позволяет сохранять лог с определенной периодичностью. Во время работы скриптов TestComplete формирует лог, который хранится в памяти. Рекомендуется время от времени сохранять лог, чтобы в случае критической ошибки

(например, TestComplete выполнит недопустимую операцию и закроется или выключится питание) потерялись не все логии, а лишь их небольшая часть.

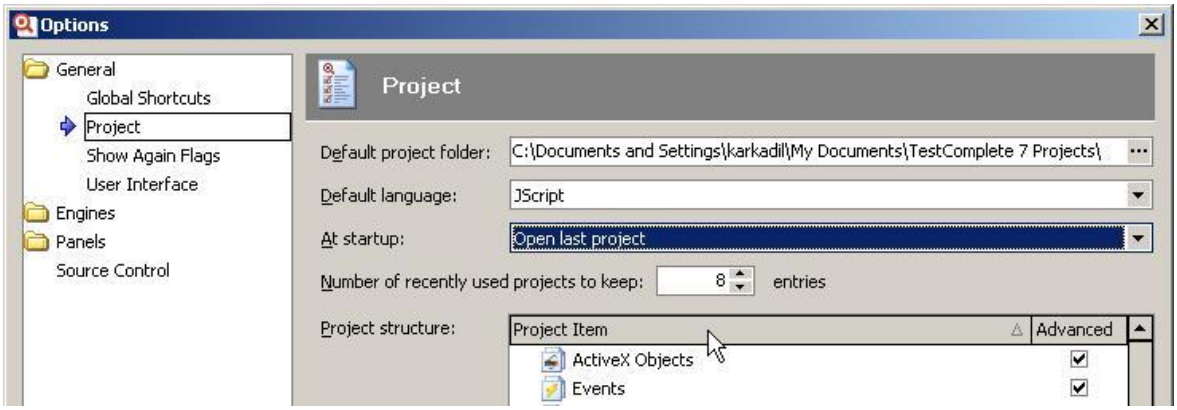
Свойства проекта по умолчанию

В TestComplete есть возможность настроить параметры по умолчанию, которые будут задаваться при создании нового проекта (Object Tree Model, месторасположение лога, ассоциации объектов, и т.п. – практически все параметры, которые меняются на панели параметров проекта).

Для изменения параметров проекта по умолчанию выберите пункт меню *Tools – Default Project Properties* и в открывшемся окне Default Project Properties укажите настройки, которые вы хотите иметь для каждого нового проекта.



Кроме того, в окне Options (*Tools – Options*) в разделе General – Project можно указать язык, который будет использоваться по умолчанию при создании нового проекта (*Default language*), папку для размещения проектов (*Default project folder*) и действие при открытии TestComplete (*At startup*) – открыть последний проект, открыть стартовую страницу или не открывать ничего.


Options
General
Global Shortcuts
Project
Show Again Flags
User Interface
Engines
Panels
Source Control
Project
Default project folder:
C:\Documents and Settings\karkadil\My Documents\TestComplete 7 Projects\
Default language:
JScript
At startup:
Open last project
Number of recently used projects to keep:
8
entries
Project structure:
Project Item
Advanced
ActiveX Objects
Events

20 Тестирование мобильных приложений

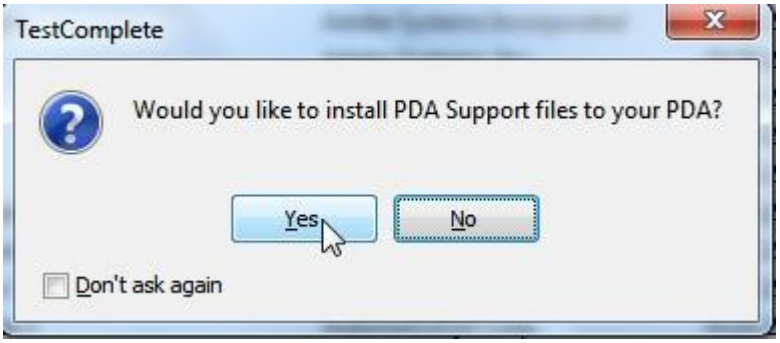
TestComplete позволяет автоматизировать тестирование WinCE приложений (т.е. приложений под Windows Mobile). Если вы устанавливали TestComplete при подключенном устройстве, то все необходимые компоненты были установлены. Если же нет – вам придется установить кое-что дополнительно.

Подготовка к тестированию

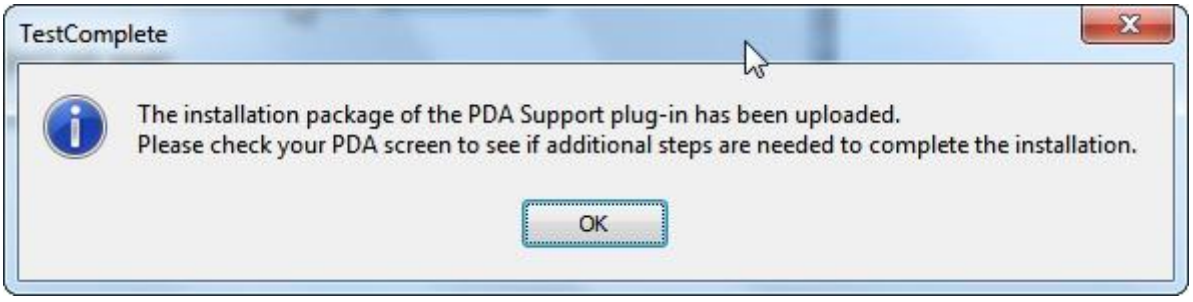
1. **MS Active Synch.** Для систем Windows XP и ниже необходимо установить Miscrosoft Active Synch, после чего подсоединить устройство к компьютеру и дождаться установления соединения. Для Windows Vista и выше всё необходимое ПО установлено по умолчанию.

2. **PDA Support Plugin.** Чтобы установить этот плагин, запустите установку TestComplete заново, и в разделе выбора компонентов приложения включите элемент PDA Support.

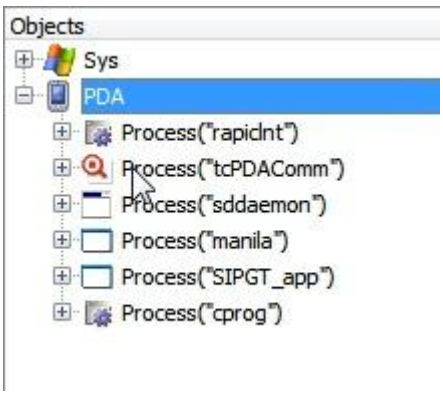
3. **ПО на WinMobile устройство.** После того, как установка завершится, запустите TestComplete и он предложит установить дополнительное ПО на ваше устройство



При этом возможно понадобится сделать дополнительные действия на самом устройстве (разрешить установку, выбрать путь для установки и т.п.)



Если всё прошло успешно, то в Object Browser-е кроме объекта Sys вы увидите новый объект PDA на одном уровне с Sys.



Тестирование

В целом тестирование мобильных приложений в TestComplete не отличается от тестирования любых других приложений. Мы так же жмем на кнопки, считываем текст из элементов управления, выбираем элементы из списков и т.п.

Для управления тестируемыми приложениями в мобильном устройстве в TestComplete есть специальный элемент проекта WinCETestedApps. По аналогии с элементом TestedApps мы можем запускать и закрывать приложения на мобильном устройстве.

Для выполнения низкоуровневых процедур на мобильном устройстве, в TestComplete также есть специальный элемент **WinCE Low-Level Procedures Collection.**

Обратите внимание, что приложения на мобильных устройствах чаще всего выполняются медленнее, чем на компьютере, поэтому стоит чаще использовать синхронизацию (об этом подробно написано в главе 3.5 Синхронизация выполнения скриптов).

Ниже показан пример запуска и простых действий с приложением ZIP на Windows Mobile 6.

```
function TestMobileApp()

{

  WinCETestedApps.ZIP.Run();

  wZip = PDA.Process("IA_Zip").Window("WCE_IA_Zip_Mai", "ZIP", 1).Window("Dialog",
"", 1);

  wZip.Window("Button", "Стоп").Click();

  wZip.Window("Button", "Найти").Click();
```

```
  wZip.Close();
}
```

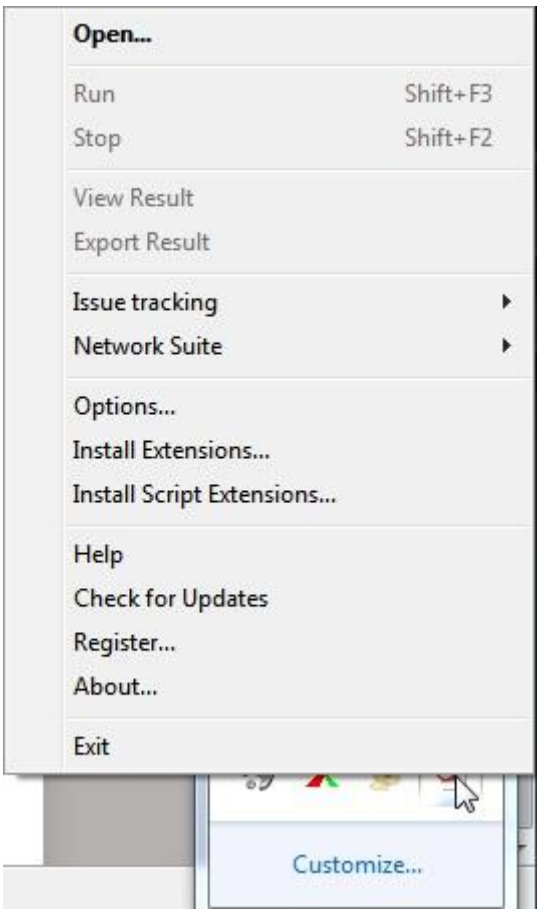
21 Использование TestExecute

TestExecute – это специальная утилита, входящая в поставку TestComplete Enterprise, которую можно использовать для запуска тестов. Фактически это тот же самый TestComplete, только без IDE. TestExecute можно использовать как для самостоятельного запуска тестов, так и для участия машины с установленным TestExecute в распределенном тестировании. Также, если вы работаете на удаленного заказчика, которому хотите показывать результаты работы, вы можете передать ему TestExecute с инструкциями, как запускать в нем тесты.

Скачать TestExecute можно там же, где вы скачивали TestComplete. Кроме того, вы можете дополнительно приобрести сколько угодно копий TestExecute, если вам необходимо, например, осуществлять распределенное тестирование на нескольких компьютерах. Стоимость TestExecute невелика по сравнению со стоимостью TestComplete.

Установка TestExecute не представляет собой ничего сложного. После установки TestExecute можно запустить из папки "C:\Program Files\Automated QA\TestExecute 7\Bin\" или ярлыком на рабочем столе. Как только TestExecute запустится, в системном трее появится его значок.

Щелкнув на нем правой кнопкой мыши мы увидим список доступных команд. Мы рассмотрим лишь некоторые из них.



- **Open** – открыть проект или набор проектов, после чего становится доступным следующий пункт
- **Run** – запустить выполнение скриптов. Для этого необходимо определить Test Items на уровне проекта (подробнее об этом можно прочитать в главе 3.7 Запуск скриптов)
- **Options** – открывает окно настроек, аналогичное окну опций TestComplete, только урезанное (подробнее о настройках можно почитать здесь 19 Настройки TestComplete)
- **Install Extensions/ Install Script Extensions** позволяет подключать/отключать надстройки (подробнее о надстройках написано 10 Создание собственных надстроек)

Кроме того, с помощью TestExecute можно запускать скрипты из командной строки, аналогично тому, как они запускаются в TestComplete. Подробнее об этом можно почитать в главе 11.2 Запуск TestComplete из командной строки

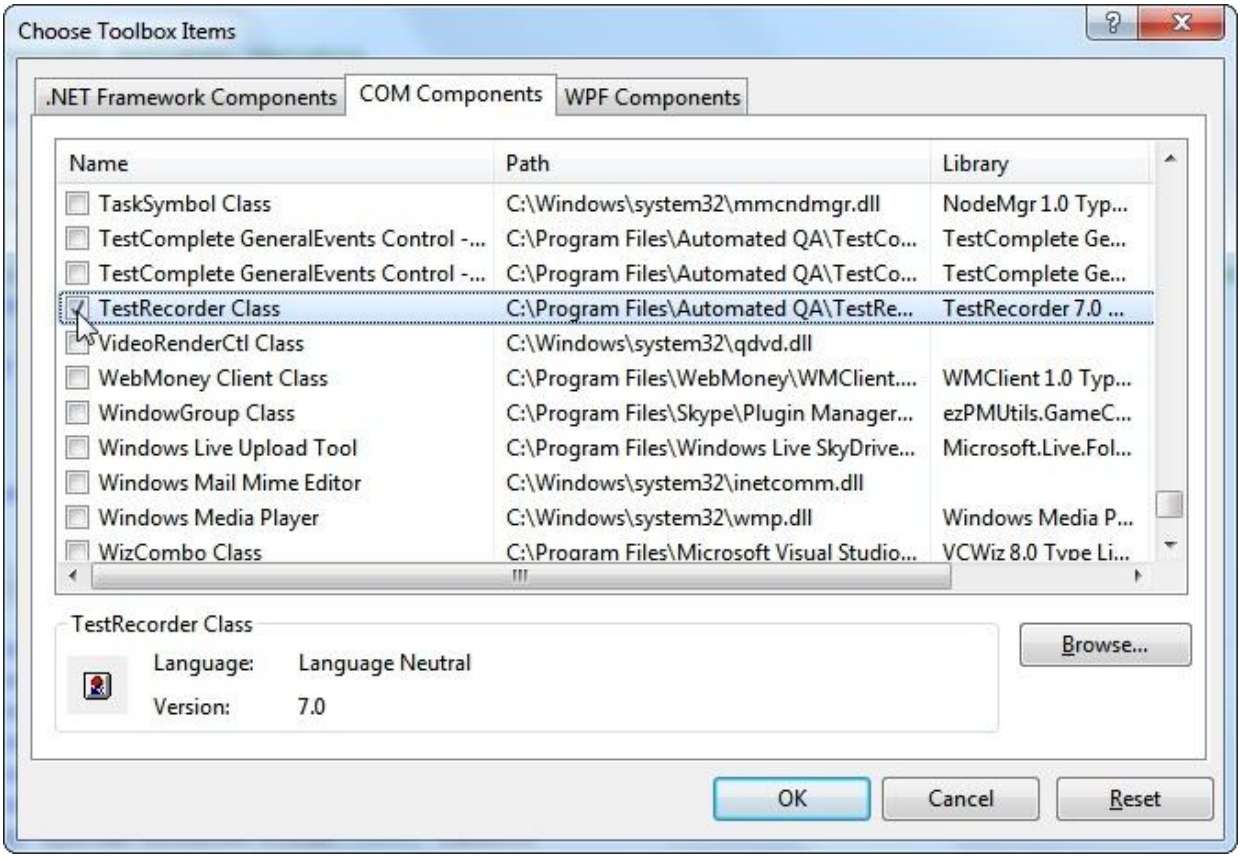
22 Использование TestRecorder

TestRecorder – это набор библиотек для разных типов приложений (.NET, Java и т.п.), которые можно подключить к тестируемому приложению. После этого все действия пользователя с приложением можно сохранить в специальном бинарном формате, а затем в TestComplete импортировать этот бинарный файл и преобразовать его в тестовый скрипт.

Инструмент этот очень удобен тем, что пользователю необязательно запоминать порядок своих действий и для тестирования вполне можно использовать обезьянку :)

Рассмотрим пример использования **TestRecorder** в приложении .NET. Для примера возьмем приложение из прилагаемого архива (nUnitTestingApp.exe).

Прежде всего добавим новый компонент на панель инструментов (меню *Tools – Choose Toolbox Items*, вкладка *COM Components*, компонент *TestRecorder Class*).



Теперь модифицируем метод **Form1_Load** таким образом, чтобы сразу при открытии главной формы приложения начинал работать TestRecorder:

```
private void Form1_Load(object sender, EventArgs e)

    {

      this.axTestRecorder1.Start(true);

    }
```

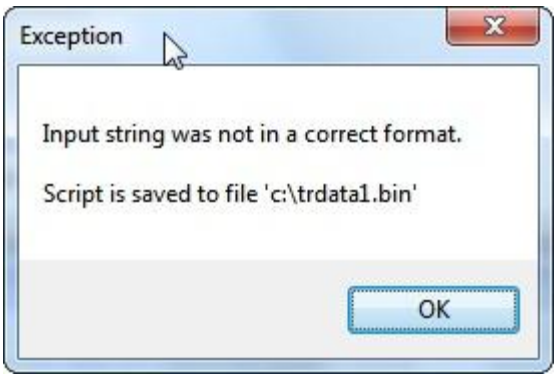
Единственный параметр метода Start указывает, какую модель Object Tree Model необходимо использовать для записи. Если вы попытаетесь вставить в проект TestComplete-а бинарный скрипт с неправильным Object Tree Model, TestComplete выдаст ошибку:

В нашем случае мы везде пользуемся моделью Flat, поэтому передаем параметр true.

Дальше нам необходимо преобразовать метод button1_Click , который вызывается при нажатии на кнопку Plus, таким образом, чтобы он перехватывал исключения:

```
private void button1_Click(object sender, EventArgs e)
    {
      try
      {
        txtResult.Text = (PlusMethod(Convert.ToInt32(this.txtVar1.Text),
          Convert.ToInt32(this.txtVar2.Text))).ToString();
      }
      catch (Exception ex)
      {
        MessageBox.Show(ex.Message + "\n\n" + @"Script is saved to file 'c:\trdata1.bin'",
"Exception");
        this.axTestRecorder1.Stop();
        this.axTestRecorder1.SaveDataToFile(@"c:\trdata1.bin");
      }
    }
```

Теперь мы можем запустить приложение, ввести какие-то цифры в поля ввода, понажимать кнопки Plus и Minus, а затем ввести в одно из полей текст (не число) и нажать Plus. При этом программа попытается конвертировать текст из этого поля в тип Int32, что и вызовет исключение



Теперь мы можем открыть TestComplete и выбрать в нем пункт меню *File – Import – TestRecorder Data – Record Script*, выберем файл **c:\trdata1.bin** и в результате получим следующий вполне работоспособный скрипт:

```
function Test3()

{

  var  nUnitTestingApp_vshost;

  var  form1;

  var  button;

  var  textBox;

  nUnitTestingApp_vshost = Sys.Process("nUnitTestingApp.vshost");

  form1 = nUnitTestingApp_vshost.Form1;

  button = form1.WinFormsObject("button1");

  button.ClickButton();

  textBox = form1.WinFormsObject("txtVar1");

  textBox.Drag(82, 10, 43, 0);

  textBox.wText = "1";

  button.ClickButton();

  textBox = form1.WinFormsObject("txtVar2");

  textBox.Drag(66, 7, 82, 0);

  textBox.wText = "\\\\";

  button.ClickButton();

  nUnitTestingApp_vshost.Window("#32770").Window("Button", "OK").ClickButton();
```

```
}
```

Из него ясно видно, что в поле txtVar2 были введены 2 слеша (“\\\\”) вместо числа, из-за чего и произошло исключение.

Часто задаваемые вопросы (FAQ)

В.: Я пытаюсь проверить, существует ли объект, с помощью свойства Exists, однако в лог мне пишется ошибка «Cannot obtain the window...». Почему?

О.: В TestComplete для того, чтобы проверить, существует ли объект, необходимо воспользоваться методами Wait, которые для каждого класса называются по-своему (например, WaitWindow для класса Window, WaitWinFormsObject для класса WinFormsObject и т.д.). Учтите, что методы Wait возвращают объект, а не логическое значение. Подробнее об использовании методов Wait можно прочитать в главе 3.5 Синхронизация выполнения скриптов.

В.: Во время тестирования мне нужно поставить задержку, однако каждый раз при выполнении задержка может быть разной. Как быть, чтобы каждый раз не использовать очень длинную (максимальную) задержку?

О.: В подобных случаях необходимо привязываться к каким-либо свойствам приложения, которые позволят точно сказать, закончилась ли операция. Например, ждать пока существует окно, или пока элемент управления не станет активным, или пока не появится определенное сообщение и т.п. В каждом конкретном случае это разные события и зависят исключительно от тестируемого приложения. Для отслеживания подобных событий используются методы Wait, о которых можно прочитать в главе 3.5 Синхронизация выполнения скриптов. *И только в самых крайних случаях можно пользоваться функцией aqUtils.Delay(), которая вставляет задержку на определенное время независимо ни от чего.*

В.: Во время работы скриптов TestComplete выводит ошибку «Объект не найден», однако я вижу этот объект в Object Browser. Почему TestComplete его не видит?

О.: Иногда в результате работы скриптов приложение обновляется (целиком или какая-то его часть). В результате скрипты пытаются работать с устаревшими объектами, которые уже не существую, хотя называются они точно так же. Обычно это происходит при использовании переменных, например:

```
var wCalc = Sys.Process("calc").Window("SciCalc");

// какие-то действия, в результате которых объект Window("SciCalc") обновляется

wCalc.Window("Button", "1").Click(); // <-- ОШИБКА! wCalc уже не существует!
```

Чтобы решить эту проблему, необходимо либо заново проинициализировать переменную, либо обновить дерево объектов с помощью метода Refresh(). Метод Refresh() определен для любого объекта. Например, все следующие примеры корректны:

```
Sys.Refresh();

Sys.Process("calc").Refresh();
```

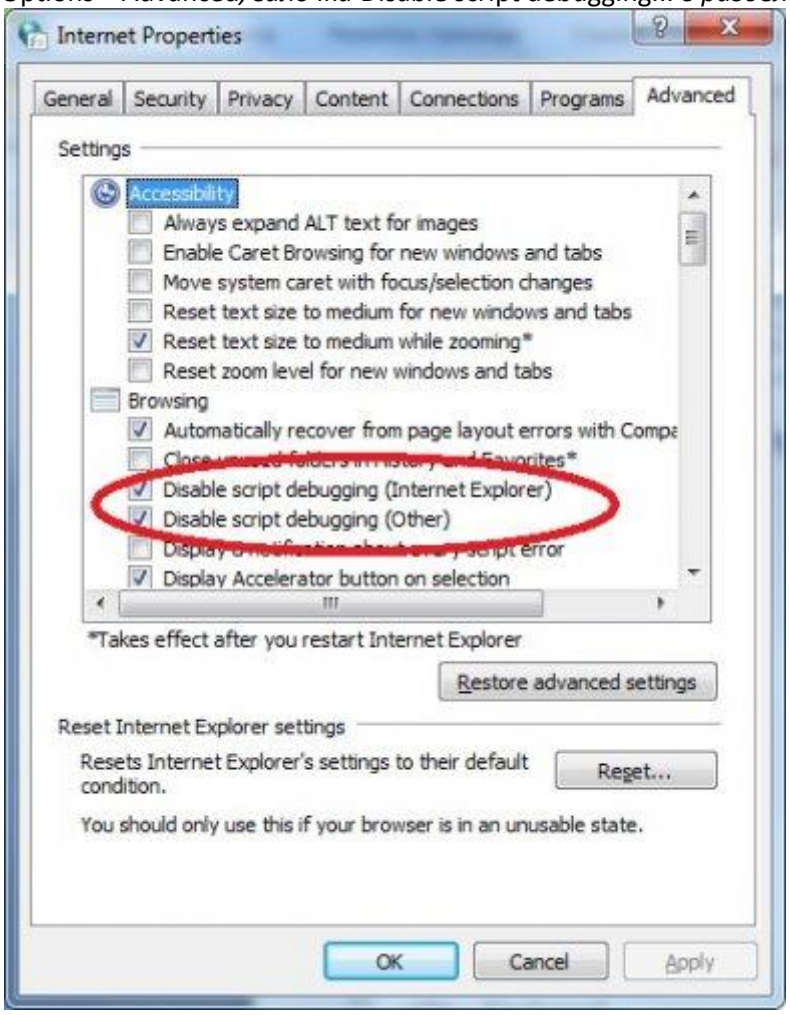
Sys.Process("calc").Window("SciCalc").Refresh();

Sys.Process("calc").Window("SciCalc").Window("Button", "1").Refresh();

В.: Я поставил в скрипте breakpoint, однако TestComplete не останавливает выполнение скриптов в этом месте. Почему?

О.: Убедитесь, что:

1. *Установлен MS Script Debugger*
2. *Включена опция* Debug – Enable Debugging *в TestComplete*
3. *Используется самая последняя версия TestComplete (в некоторых версиях наблюдалась эта проблема, но была быстро устранена в следующих версиях)*
4. *Текущий пользователь системы имеет права Администратора или Отладчика*
5. *TestComplete не запущен в «тихом» режиме (Silent Mode, см. главу 11.2 Запуск TestComplete из командной строки)*
6. *В параметрах интернета включена опция отладки (*Start – Control Panel – Internet Options – Advanced*, галочки* Disable script debugging... *в разделе* Browsing*)*



7. *Если ничего выше не помогло — перезапустите TestComplete и/или компьютер*

В.: TestComplete работает медленно (во время записи или воспроизведения). Что можно сделать?

О.: Есть несколько причин (кроме очевидной: слабый компьютер), которые могут влиять на скорость работы TestComplete во время записи/воспроизведения. Вот несколько советов, которые могут вам помочь:

1. *Отключите Visualizer. Если он включен, то во время записи/воспроизведения создаются скриншоты объектов, что, естественно, замедляет работу*
2. *Отключите неиспользуемые Extension-ы (меню File - Install Extensions). Например, если вы не тестируете .NET приложения, то и плагин ".NET Open Applications Support" вам ни к чему*
3. *Используйте фильтр процессов в Object Browser-е. Начиная с версии 6, TestComplete позволяет отображать в Object Browser-е не все процессы, а только выбранные. Рекомендуется включить отображение только тех процессов, с которыми работают ваши скрипты. Это ускорит обновление дерева объектов в Object Browser-е*
4. *Если вы уверены, что причина медленной работы в тестируемом приложении - попробуйте поменять Object Tree Model и/или Web Tree Model*
5. *В некоторых случаях стандартные методы TestComplete работают медленно (яркий пример - метод Keys в Internet Explorer 7 и выше). Попробуйте другие методы работы с такими элементами управления (например, присваивайие значения текстовым полям напрямую через свойство value вместо использования метода Keys)*
6. *Убедитесь, что задержки при выполнении определенных операций (ввод текста, движение мыши и т.д.) выставлены на минимум (правый щелчок на имени проекта, Edit - Properties - Playback, группа опций "Delqy...")*
7. *Также убедитесь, что методы Wait и Delay используются разумно во всем проекте, так как их частое использование может приводить к существенным задержкам при воспроизведении скриптов*
8. *Отключите Debug Agent, если он вам не нужен*

Более полный список улучшений можно прочитать в статье Automated Test Performance Tips (на английском языке).

«Грязные» трюки :)

Здесь мы храним разные интересности, связанные с TestComplete'ом. Ничего особо грязного тут нету, а некоторые вещи и вовсе бесполезны, но раз уж на них натыкаешься — надо где-то их хранить.

Зарезервированная переменная MSG

Если в TestComplete создать новый пустой модуль и попытаться в нем выполнить строку

```
Log.Message(MSG);
```

то мы получим странное сообщение об ошибке: "Object doesn't support this property or method", хотя в случае попытки использования необъявленных переменных должна возникать ошибка "Microsoft JScript runtime error 'MSG' is undefined".

MSG — это зарезервированное имя структуры, которая содержит элементы Win32-сообщения. Чтобы использовать такую переменную, достаточно ее объявить с обязательным использованием ключевого слова var. Объяснение этому феномену было дано в официальной ньюсгруппе.

Точка в конце строки кода

Если вам необходимо запустить одну строку из редактора, нет необходимости помещать ее в отдельную функцию. Достаточно в конце строки перед точкой с запятой (если таковая имеется) поставить обычную точку. Например, вставьте в редактор следующую строку:

```
Sys.Process("explorer").Terminate();
```

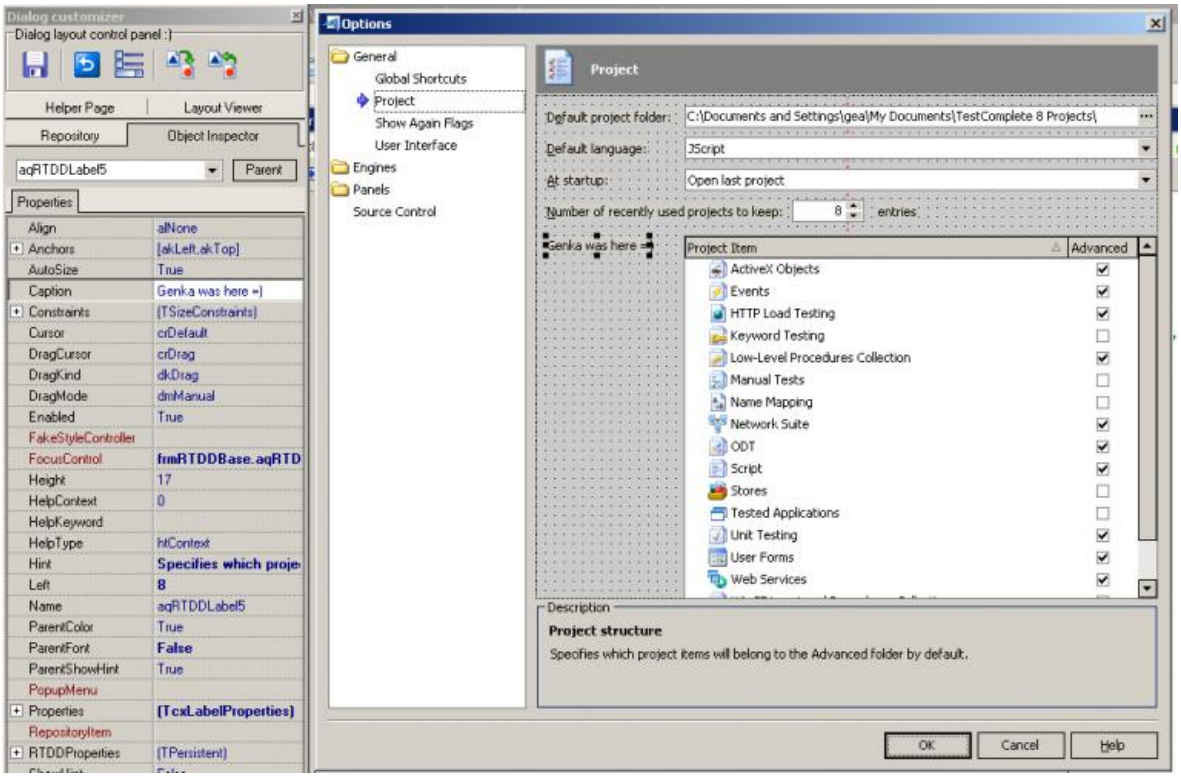
и поставьте точку перед точкой с запятой:

```
Sys.Process("explorer").Terminate().;
```

В результате у вас завершится процесс explorer и пропадет панель задач :)

Окно Dialog Customizer

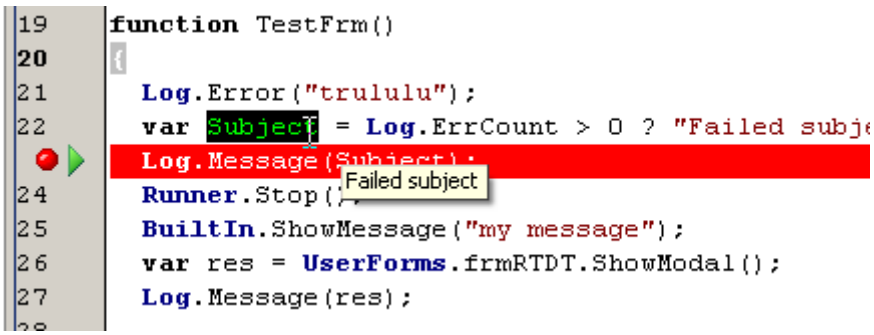
Откорйте окно Options (меню Tools -> Options) и нажмите комбинацию клавиш Ctrl-Alt-Shift-c. В результате на экране появится панелька Dialog Customizer и возможность редактировать окно Options.



Только будьте осторожны, а то там можно такое понаделывать, что обратно потом не вернёте :)

Вычисление значений выражений в режиме отладки

Если поставить брекпоинт на какой-то строке, то во время работы скрипта TestComplete приостанавливает выполнение скрипта в этом месте. Если во время этой паузы навести курсор мыши на переменную (или выделить какое-то выражение и навести курсор мыши на выделенный текст), TestComplete выдаст значение переменной/выражения во всплывающей подсказке.



Так как TestComplete старается вычислять значения выражений, на которые мы наводим курсор, это может приводить к странным последствиям. Например:

- если навести курсор мыши на строку `Runner.Stop();`, то выполнение скрипта прекратится и TestComplete выйдет из режима Debug
- если навести курсор мыши на строку `BuiltIn.ShowMessage("...")`, TestComplete покажет соответствующее сообщение
- если в панели Watch List ввести выражение `Sys.HighlightObject(...)` и вместо многоточия вставить имя любого существующего объекта, TestComplete подсветит этот объект на экране (но при этом сам TestComplete зависает намертво)