**Лабораторная работа №3**

**“Тестирование пользовательского интерфейса. CodedUi тесты.”**

**Цель работы:** Научиться формировать требования к пользовательскому интерфейсу и проводить тестирование графического интерфейса.

**Продолжительность работы** - 4 часа.

**Содержание**

## 1. Тестирование графического интерфейса….........................................................................1 2. Требования к пользовательскому интерфейсу ………………………...............................2 3. NUnitForms…………..……………........................................................................................4 4. Coded UI…………………..…………………........................................................................6

## 5. Порядок выполнения лабораторной работы......................................................................13

**Тестирование графического интерфейса**

Графический интерфейс пользователя (Graphical user interface, GUI) –разновидность пользовательского интерфейса, в котором элементы интерфейса (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

Задачей тестирования графического интерфейса пользователя является обнаружение ошибок следующего характера:

* Ошибки в функциональности посредством интерфейса
* Необработанные исключения при взаимодействии с интерфейсом
* Потеря или искажение данных, передаваемых через элементы интерфейса
* Ошибки в интерфейсе (несоответствие проектной документации, отсутствие элементов интерфейса)

Функциональное *тестирование пользовательского интерфейса* состоит из пяти фаз:

* анализ требований к пользовательскому интерфейсу;
* разработка тест-требований и тест-планов для проверки пользовательского интерфейса;
* выполнение тестовых примеров и сбор информации о выполнении тестов;
* определение полноты покрытия пользовательского интерфейса требованиями;
* составление отчетов о проблемах в случае несовпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Так, тест-планы для проверки пользовательского интерфейса, как правило, представляют собой сценарии, описывающие действия пользователя при работе с системой. Сценарии могут быть записаны либо на естественном языке, либо на формальном языке какой-либо системы автоматизации пользовательского интерфейса. Выполнение тестов при этом производится либо оператором в ручном режиме, либо системой, которая эмулирует поведение оператора.

При сборе информации о выполнении тестовых примеров обычно применяются технологии анализа выводимых на экран форм и их элементов (в случае графического интерфейса) или выводимого на экран текста (в случае текстового), а не проверка значений тех или иных переменных, устанавливаемых программной системой.

Под полнотой покрытия пользовательского интерфейса понимается то, что в результате выполнения всех тестовых примеров каждый интерфейсный элемент был использован хотя бы один раз во всех доступных режимах.

Отчеты о проблемах в пользовательском интерфейсе могут включать в себя как описания несоответствий требований и реального поведения системы, так и описания проблем в требованиях к пользовательскому интерфейсу. Основной источник проблем в этих требованиях - их тестонепригодность, вызванная расплывчатостью формулировок и неконкретностью.

**Требования к пользовательскому интерфейсу**

Требования к пользовательскому интерфейсу могут быть разбиты на две группы:

* требования к внешнему виду пользовательского интерфейса и формам взаимодействия с пользователем;
* требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса.

Другими словами, первая группа требований описывает взаимодействие подсистемы интерфейса с пользователем, а вторая - с внутренней логикой системы.

**Требования к размещению элементов управления на экранных формах**.

Данные требования могут определять общие принципы размещения элементов пользовательского интерфейса или требования к размещению конкретных элементов. Например, общие требования по размещению элементов на графической экранной форме могут выглядеть следующим образом: *Каждое окно приложения должно быть разбито на три части: строка меню, рабочая область и статусная строка. Строка меню должна быть горизонтальной и прижатой к верхней части окна, статусная строка должна быть горизонтальной и прижатой к нижней части окна, рабочая область должна находиться между строкой меню и статусной строкой и занимать всю оставшуюся площадь окна*. При тестировании данного требования достаточно определить, что в каждом окне системы действительно присутствуют три части, которые расположены и прижаты согласно требованиям даже при изменении размеров окна, его сворачивании/разворачивании, перемещении по экрану, при перекрытии его другими окнами. Примером требований по размещению конкретного элемента может служить следующее: *Кнопка "Начать передачу" должна находиться непосредственно под строкой меню в левой части рабочей области окна*. При тестировании такого требования также необходимо определить, сохраняется ли расположение элемента при изменении размера окна, а также при использовании элемента (в данном случае - при нажатии).

**Требования к содержанию и оформлению выводимых сообщений**

Требования к содержанию и оформлению выводимых сообщений определяют текст сообщений, выводимых системой, его шрифтовое и цветовое оформление. Также часто в таких требованиях определяется, в каких случаях выводится то или иное сообщение.

Так, например, для тестирования требования *Сообщение "Невозможно открыть файл" должно выводиться в статусную строку прижатым к левому краю, красным цветом, полужирным шрифтом в случае недоступности открываемого файла по чтению*.

необходимо проверить, что при возникновении указанной ситуации сообщение действительно выводится согласно требованиям. Однако в случае тестирования требования вида *Сообщения об ошибках должны выводиться в статусную строку прижатыми к левому краю красным цветом полужирным шрифтом*. необходимо проверять форматы всех возможных сообщений об ошибках программы во всех возможных ошибочных ситуациях. Таким образом, очевидно, что при тестировании пользовательского интерфейса не всегда можно однозначно определить количество тестовых примеров, которые понадобятся для тестирования требования. Эта проблема вызвана тем, что требования к пользовательскому интерфейсу зачастую кажутся слишком очевидными для их точной формулировки. Эта неконкретность требований и вызывает большое количество тестов для каждого требования.

**Требования к форматам ввода**

Данная группа требований определяет, в каком виде информация поступает от пользователя в систему. При этом кроме собственно требований, определяющих корректный формат, к этой группе относятся требования, определяющие реакцию системы на некорректный ввод. Для проверки таких требований необходимо проверять как корректный ввод, так и некорректный. Желательно при этом разбивать различные варианты ввода на классы эквивалентности (как минимум на два - корректные и некорректные).

**Требования к реакции системы на ввод пользователя**

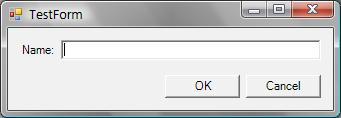
Данный тип требований определяет связь внутренней логики системы и интерфейсных элементов. Например, *При нажатии кнопки "Сброс" значение таймера синхронизации передачи должно сбрасываться в 0*. Для проверки такого требования в тестовом примере должно быть сымитировано нажатие на кнопку "Сброс", после чего должна проводиться проверка значения таймера. Однако некоторые требования определяют в качестве реакции системы не то, как меняется ее внутреннее состояние, а реакцию пользовательского интерфейса. Например, в требовании *При нажатии кнопки "Отложенный сброс" должно выводиться окно "Ввод значения времени для отложенного сброса"*. в качестве реакции на использование одного интерфейсного элемента определяется появление другого интерфейсного элемента. Такие требования проверяются при помощи имитации ввода пользователя и анализа появляющихся интерфейсных элементов.

**Требования к времени отклика на команды пользователя**

В качестве отдельного типа требований можно выделить требования к времени отклика системы на различные пользовательские операции. Это связано с тем, что подсознательно пользователь воспринимает операции продолжительностью более 1 секунды как длительные. Если в этот момент система не сообщает пользователю о том, что она выполняет какую-либо операцию, пользователь начнет считать, что система зависла или работает в неверном режиме. В связи с этим либо каждое предельное время отклика должно быть указано в требованиях и пользовательской документации, либо во время длительных операций должны выводиться информационные сообщения (например, индикатор прогресса). Значения предельного времени и равномерность увеличения значений индикатора прогресса должны проверяться соответствующими тестами.

## NUnitForms

NUnitForms является расширением фреймворка NUnit и позволяет делать автоматическое тестирование Windows Forms приложений. Тесты, в данном случае, могут открывать формы и работать с элементами управления, имитируя работу пользователя. Общие сведения, дистрибутивы и документация доступны здесь:  <http://sourceforge.net/projects/nunitforms>. Не вдаваясь в подробности структуры NUnitForms, рассмотрим пример unit-теста для диалогового окна с одним текстовым полем, кнопками «OK» и «Cancel»:



В данном случае функционал очень простой: если никакие данные не введены, выводится сообщение «Поле Name не может быть пустым» и форма не закрывается; если же в поле введены данные, то форма закрывается для последующей обработки введенного значения без каких-либо сообщений. Здесь очевидны два тестовых сценария: один — для проверки случая, когда данные не введены, второй — когда в поле формы введены данные.

Посмотрим, как будет выглядеть код unit-теста для вышеописанного функционала в рамках NUnitForms:

using System.Windows.Forms;  
using NUnit.Extensions.Forms;  
using NUnit.Framework;  
  
namespace InrecoLan.NUnitForms.Tests  
{  
 [TestFixture]  
 public class NUnitTest : NUnitFormTest  
 {  
 TestForm form;  
 public override void Setup()  
 {  
 base.Setup();  
 form = new TestForm();  
 form.Show();  
 }  
  
 [Test]  
 public void TestNoData()  
 {  
 ExpectModal("Message", new ModalFormActivated(TestFormNoDataHandler));  
 var nameTextbox = new TextBoxTester("txtName");  
 nameTextbox["Text"] = string.Empty;  
 Assert.AreEqual(string.Empty, nameTextbox.Text);  
 var okButton = new ButtonTester("btnOK");  
 okButton.Click();  
 Assert.IsFalse(form.DialogResult == DialogResult.OK);  
 }  
  
 [Test]  
 public void TestData()  
 {  
 var nameTextbox = new TextBoxTester("txtName");  
 nameTextbox["Text"] = "abcdefg";  
 Assert.AreEqual("abcdefg", nameTextbox.Text);  
 var okButton = new ButtonTester("btnOK");  
 okButton.Click();  
 Assert.IsTrue(form.DialogResult == DialogResult.OK);  
 }  
  
 public void TestFormNoDataHandler()  
 {  
 var messageBoxTester = new MessageBoxTester("Message");  
 if (messageBoxTester != null)  
 {  
 messageBoxTester.ClickOk();  
 }  
 }  
 }  
}

**Примечания/пояснения:**

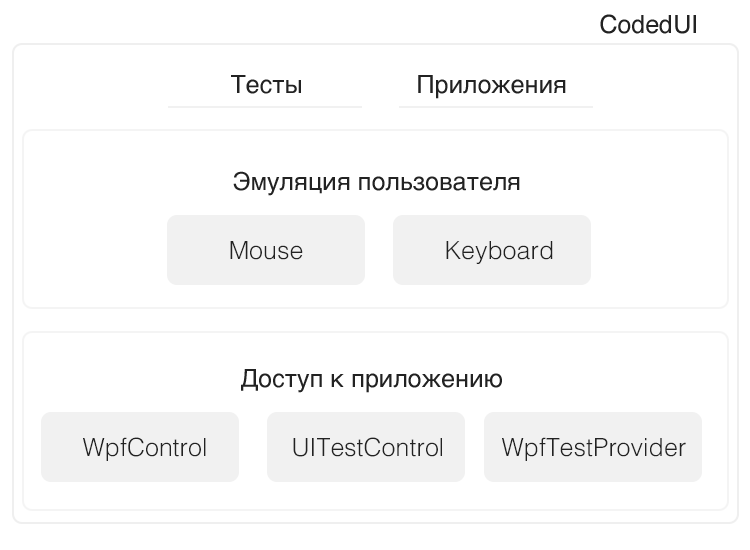
|  |
| --- |
| * обращение к элементам управления осуществляется по именам, к формам — по заголовку; * метод Setup() вызывается каждый раз перед запуском очередного теста (метод с атрибутом [Test]); * если в процессе выполнения тестового сценария предполагается вызов модальных форм (MessageBox, другие формы, которые вызываются с помощью метода ShowDialog()), то такие вызовы должны обрабатываться особым образом, с применением метода ExceptModal(), параметрами являются заголовок окна и обработчик, в котором прописывается последовательность действий для вызываемого диалогового окна; * как видно из листинга, тест для GUI почти не отличается от обычного, многим знакомого, unit-теста в контексте фреймворка NUnit, за исключением следующего: класс теста наследуется от NUnitFormTest, для доступа к элементам управления применяются специальные классы ButtonTester, TextBoxTester, MessageBoxTester. |

**Coded UI**

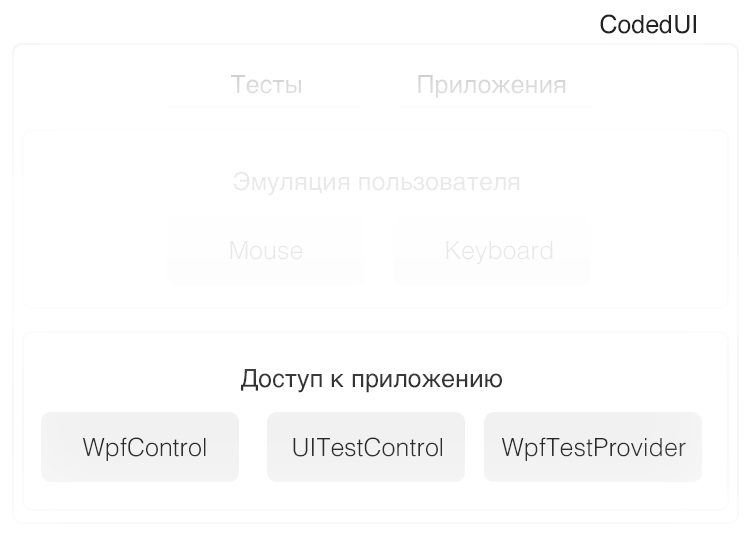
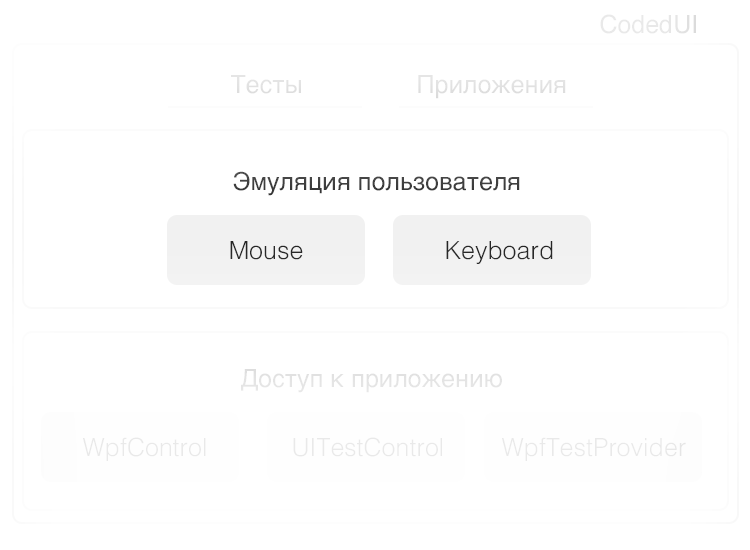
Модульные тесты обычно работают, вызывая методы в интерфейсе тестируемого кода. Однако, при разработке пользовательского интерфейса, полноценное тестирование должно включать в себя нажатие кнопок и проверку появления соответствующих окон с правильным содержанием. Программируемые тесты пользовательского интерфейса (CUIT) – это автоматизированные тесты, проверяющие действия в пользовательском интерфейсе. Детальнее в теме MSDN [Testing the User Interface with Automated Coded UI Tests](http://msdn.microsoft.com/ru-ru/library/dd286726.aspx).

Отличительной особенностью Coded UI является Рекордер, позволяющий создавать тесты даже без навыков программирования. Код тестов может писаться с помощью рекордера и самостоятельно, используя классы, предоставляемые пространством имён Microsoft.VisualStudio.TestTools.UITesting (их использует и сам CodedUI при генерации кода). Это позволяет получать тесты с понятным, компактным кодом, но не избавило от проблем поиска элементов и их доступности.

Представить себе CodedUI в работе можно как инфраструктуру, состоящую из классов фреймворка, тестируемого приложения и самих тестов. Подробнее компоненты изображены на схеме:



Для того, чтобы элемент приложения мог о себе что-то сказать, он должен реализовывать технологию Microsoft UI Automation. В таком случае он позволяет программно обращаться к нему, используя библиотеки из пространства имен System.Windows.Automation. Они применяются, например, в уже упомянутом на диаграмме классе WpfTestProvide.

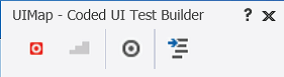
  
Первому нашедшему зачет по лабораторной работе №3  


**Как создавать и использовать CUIT**. **Создание CUIT.**

Для того, чтобы создать программируемый тест пользовательского интерфейса, необходимо создать проект CUIT. В диалоговом окне **New Project**, опция создания CUIT находится под **Visual C#\Test**. Если проект уже создан, в него нужно добавить новый программируемый тест пользовательского интерфейса.

Для записи теста, в диалоговом окне **Generate Code** нужно выбрать **Record Actions**. Окно Visual Studio минимизируется и в нижней правой части экрана появится рабочее окно для записи CUIT.

Просто нажимаем Record и запускаем приложением, которое необходимо протестировать.

[](http://devopshub.net/wp-content/uploads/2013/10/1.png)

**Запись CUIT**

Выполняем все действия, которые хотим протестировать (позже можно будет их отредактировать).

Можно также использовать кнопку **Target** для создания проверок состояния элементов пользовательского интерфейса.

Кнопка **Generate Code** преобразует выполненную последовательность действий в код модульного теста. Сгенерированный код можно менять по своему усмотрению. Например, удалить нечаянно выполненное действие.

**Выполнение CUIT**

CUIT выполняются аналогично остальным модульным тестам. При работе с репозиторием, UI тесты следует загружать вместе с остальными тестами, и они выполнятся как часть тестов проверки сборки.

**Подсказка**: Пока выполняются CUIT, не стоит прикасаться к мышке и клавиатуре.

**Изменение и добавление проверок**

Итак, у нас уже есть последовательность действий, преобразованная в исполняемый код. При выполнении этого кода, записанные действия воспроизводятся.

Но не хватает собственно проверок. Код, тестирующий состояния элементов UI, нужно добавлять самостоятельно. Используя кнопку Target можно создать прокси объект, который будет представлять выбранный элемент UI. После этого пишется код, использующий публичные методы таких прокси объектов для того, чтобы тестировать состояния элементов.

**Расширение обычной процедуры для использования множества значений**

Код можно изменять для того, чтобы записанная процедура повторялась с разными входящими значениями.

Самый простой вариант – это добавить в код цикл и наборы значений.

А можно связать тест с отдельной таблицей значений, которая может быть файлом электронных таблиц, файлом XML или базой данных. Например, в Excel рисуется таблица, в которой каждый ряд – это набор данных для каждого повторения цикла. В каждой колонке предоставляются значения для определенной переменной. Первый ряд – это шапка, в которой определены имена переменных:

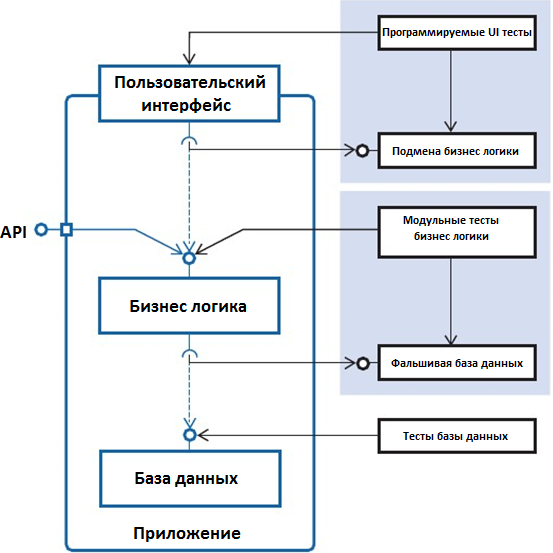
**Вкус                                                                     Размер**  
Овсянка                                                               Маленькая  
Селедка                                                               Большая

В свойствах теста создается новая связь (Data Connection String). Мастер создания связей позволяет выбрать источник данных. Внутри кода можно писать выражения типа

C#  
var flavor = TestContext.DataRow[“Вкус”].ToString();

**Изоляция**

Как и при работе с любым модульным тестом, тестируемые компоненты или уровни можно изолировать – в данном случае пользовательский интерфейс – предоставляя фальшивый бизнес уровень. Подмена должна просто вести журнал вызовов и иметь возможность менять свои состояния так, чтобы с помощью проверок можно было убедиться, что пользовательский интерфейс совершил правильные вызовы и правильно отобразил состояние.



Хорошо изолированные модульные тесты

**Как быть с подходом «сначала тесты»?**

В такой ситуации может показаться, что нецелесообразно писать тесты до того, как написан код. Ведь нужно создать интерфейс пользователя перед тем, как записывать действия для CUIT.

В какой-то мере это справедливо, особенно если пользовательский интерфейс динамически реагирует на состояния бизнес логики. Тем не менее, часто можно записать некоторые действия над кнопками, которые особо ничего не делают во время записи, и добавить проверки, которые заработают после того, как будет подключена бизнес логика.

**CUIT – это модульные или системные тесты?**

Программируемые UI тесты являются очень эффективным способом быстро писать тесты. Если честно, то CUIT разрабатывались под две задачи: тестировать пользовательский интерфейс в изоляции (с подмененной бизнес логикой); и системное тестирование всего приложения (этой теме будут посвящены будущие заметки).

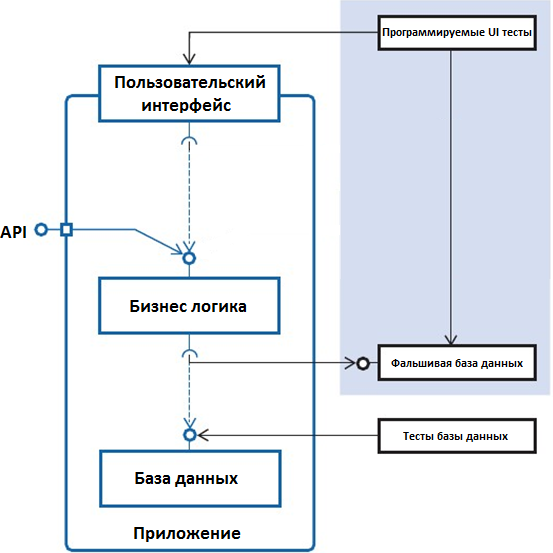
Но CUIT настолько быстрый способ создания тестов, что так и хочется расширить сферу их применения. Предположим, что создается маленькое десктопное приложение, которое пользуется базой данных или сетью. И бизнес логика определяется пользовательским интерфейсом. Очевидно, что быстрый способ написания тестов бизнес логики – это записать CUIT для всех основных функций, подменяя внешние источники (web или базу данных). Казалось бы, это более оптимальное использование времени, чем писать код для тестов бизнес логики.

Если прислушаться, то можно услышать громкое возмущение приверженцев современной методологии. Ведь тестирование бизнес логики нажатием кнопок в графическом интерфейсе намертво связывает UI и бизнес логику. А это ставит крест на лучших практиках разработки, которые требуют разделять интерфейс и логику. Причина банальна – при изменении пользовательского интерфейса перестают работать тесты бизнес логики.

Но это еще не все. Поскольку CUIT можно создать только после того, как программа уже запускается, следование такому подходу не позволит применить стратегию «сначала тесты» (которая позволяет сосредоточиться на идеях и обсуждении того, что именно должен делать код).

По этим причинам, не рекомендуется использовать CUIT как замену правильным модульным тестам бизнес логики. Лучше думать о бизнес логике как определяемой API (зависящей от других компонент кода), а об UI просто как об одном из способов вызывать действия API. Ну а для написания API, хорошо начинать с написания примеров последовательностей вызовов, которые станут частью тестовых методов.

Если есть уверенность в том, что приложение будет жить недолго, оно мало и незначительно, тогда CUIT могут быть отличным способом быстро набросать несколько тестов.

[](http://devopshub.net/wp-content/uploads/2013/10/14.png)

**CUIT для тестирования приложения**

**Дизайн программируемых UI тестов**

При запуске теста, движок CUIT должен найти каждый элемент управления, который используют записанные действия. Это достигается с помощью прохода по дереву представления, используя имена элементов UI. Если пользовательский интерфейс был изменен, тесты могут не работать из-за того, что не смогут найти элементы. Хотя у движка есть некоторая эвристика для поиска изменившихся элементов, можно увеличить ее шансы сработать:

* В HTML у каждого элемента должен быть идентификатор.
* В Windows технологии презентаций надо использовать Accessibility.
* При разработке своих элементов управления, можно определить расширение CUIT для того, чтобы помочь программе записи распознавать действия (подробнее в теме MSDN [Enable Coded UI Testing of Your Custom Controls](http://msdn.microsoft.com/ru-ru/library/hh552522.aspx" \t "_blank)).Например, при использовании элемента выбора файла, записывается не последовательность кликов, а какой файл был выбран.

**Поддержание работоспособности программируемых UI тестов**

Недостатком CUITs является то, что они должны быть воспроизводимы даже при значительных изменениях пользовательского интерфейса. Можно минимизировать необходимые усилия по обеспечению работоспособности CUIT:

* Создавать отдельные записи, тем самым разделяя методы, для разных форм или страниц, а также для групп из небольшого количества действий.
* В случае изменений определять затронутые методы и перезаписывать только их тесты.
* Использовать редактор CUIT для обновления кода. Код можно изменять и напрямую, но при использовании редактора результаты получаются более надежными. Больше информации в теме MSDN [How to: Edit a Coded UI Test Using the Coded UI Test Editor](http://msdn.microsoft.com/ru-ru/library/gg269469.aspx).

**Порядок выполнения лабораторной работы**

1. Создать графический интерфейс калькулятора в соответствии с документацией (см. комментарии в проекте к лабораторной работе №2).
2. С помощью тестов пользовательского интерфейса и библиотеки для создания mock создать набор тестов, которые проверяют, что при нажатии на кнопку соответствующей операции вызывается заданный метод прослойки (например, при вызове метода прослойкой DisplayError() отображается диалоговое окно).

Инструкции по работе с тестами:

<https://ashamray.wordpress.com/tag/coded-ui/>  
 <https://habrahabr.ru/post/97012/>  
 <https://msdn.microsoft.com/en-us/library/dd286726.aspx#VerifyingCodeUsingCUITCreate>

**Ссылки:**

1. <http://delta-course.org/docs/Delta2014S-T2-L7.pdf>
2. <http://www.intuit.ru/studies/courses/1040/209/lecture/5418>
3. <http://blog.qapl.ru/4-ways-to-automate-qt/>
4. <https://inrecolan.ru/blog/ilya-koshevoy/415-nunitforms>
5. <http://sonyks2007.blogspot.ru/2014/07/coded-ui-tests.html>
6. <https://habrahabr.ru/company/2gis/blog/220337/>
7. <http://devopshub.net/programmiruemyie-testyi-polzovatelskogo-interfeysa/>