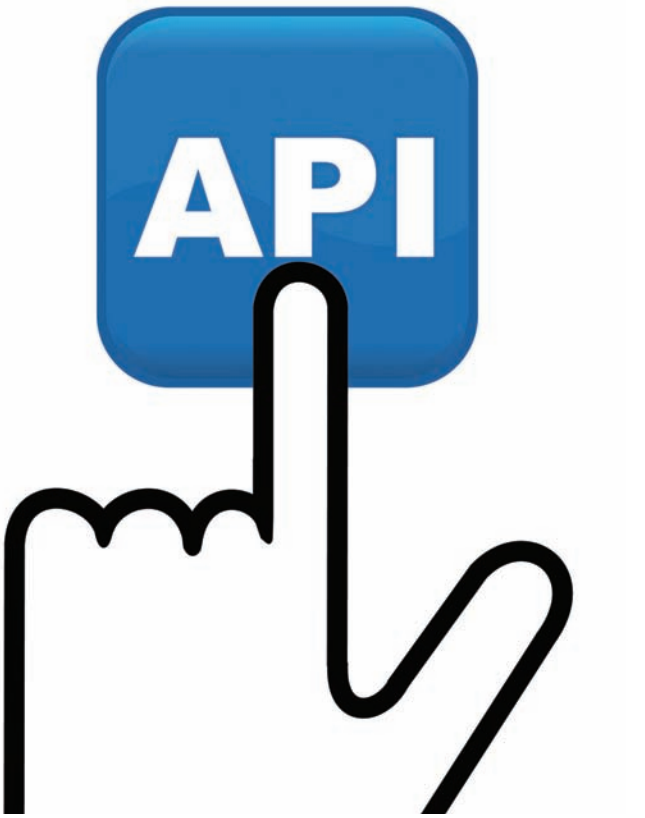

УНИВЕРСАЛЬНЫЙ АВТОТЕСТ, ИЛИ КАК МЫ АВТОМАТИЗИРОВАЛИ РУЧНЫЕ ТЕСТЫ API В nanoCAD

Как можно вручную протестировать API? И напротив, если есть API, чем плохи модульные тесты? При разработке API nanoCAD мы столкнулись с тем, что не весь API можно протестировать при помощи модульных тестов: часть API неразрывно связана с пользовательским интерфейсом и интерактивным взаимодействием с пользователем.

В этой статье мы расскажем о том, как мы тестировали API вручную, через какие стадии автоматизации прошли, и какой подход позволил нам создать надежные и легко поддерживаемые автотесты[1] (рис. 1).

Рис. 1

Задачи тестирования nanoCAD можно разделить на две основные группы. Первая – тестирование nanoCAD как инструмента конструктора, где главным вопросом является: "Может ли конструктор начертить то, что требуется?". Вторая группа – тесты API. Здесь мы ищем ответ на другой вопрос: "Могут ли работать те приложения, которые должны?".

С ручными тестами программы, в которой работает пользователь, всё понятно. А что же такое ручной тест API? И почему бы не использовать для тестирования API обычные модульные тесты?

Действительно, для тестирования значительной части API модульные тесты подходят. Но есть ряд функций, вызов которых приводит к запросу действия пользователя. Например, приложение, вызвав функцию GetPoint() или GetAngle(), может попросить пользователя ввести точку или угол. Чтобы проверить, как работает подобный функционал, мы должны создать тестовое приложение, вызывающее функции API со всеми основными вариантами параметров, но для автоматизации таких тестов нам понадобится каким-либо образом автоматизировать действия пользователя.

Схема ручных тестов API имеет вид, показанный на рис. 2.

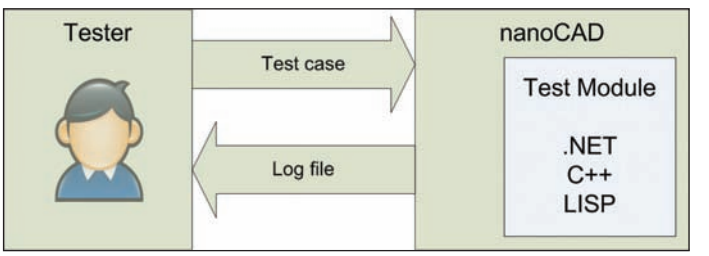

Рис. 2

Для проверки каждой функции API была написана отдельная команда. Тестировщик запускает команды вручную, далее он выбирает точки на экране, вводит координаты и т.п. согласно тест-кейсу.

[1] *По мотивам доклада на конференции SQA Days-12.*

```
Команда: GetAngleTest
Должен быть загружен документ UnitTest.dwg
Укажите угол мышью, первая точка: 0,0,0
Вторая точка: 100,100,0
Проверка на отмену. Нажмите Esc: <Esc>
Проверка на ввод 0. Введите 0: 0
Проверка на запрещение ввода 0. Введите 0: 0
Значение должно быть ненулевым.
Проверка на запрещение ввода 0. Введите 0: 1
Проверка на запрещение пустого ввода. Нажмите Enter или пробел:
<Enter>
Проверка на запрещение пустого ввода. Нажмите Enter или пробел:
<Space>
Проверка на запрещение пустого ввода. Нажмите Enter или пробел:
1
Проверка на пустой ввод. Нажмите Enter или пробел: <Enter>
Проверка свободного ввода. Введите не число: #sqadays12
Проверка на значение по умолчанию. Нажмите Enter <135>: <Space>
Проверка ввода с ключевыми словами. Введите число или [Пи/Два-
пи/]: Пи
Проверка ввода по умолчанию с ключевыми словами.
Нажмите Enter или пробел <135> или [Пи/Два-пи/]: <Enter>
```

По завершении тест-кейса анализируется лог-файл.

Приступая к автоматизации, мы начали с нажатия "красной кнопки" и записали скрипт. Чуда не случилось: все точки, как и ожидалось, были записаны в экранных координатах.

```
Window.MouseClick(100, 200); // Экранные координаты
```

Что делать? Экранная система координат не подходит, поскольку положение элементов чертежа на экране может меняться, а надежный автотест не должен зависеть от:

- размеров и взаиморасположения окна приложения и его панелей управления;
- отображения элементов управления в разных версиях ОС, от стилей оформления ОС;
- многомониторных конфигураций,
- и многого-многого другого...

Ответ напрашивался сам собой: нужно хранить точки в системе координат чертежа (рис. 3).

Но как в ходе теста преобразовывать координаты чертежа в экранные координаты?

Традиционным способом решения этой задачи является создание адаптера для системы автоматизированного тестирования, который загружается в nanoCAD и осуществляет пересчет координат. Однако создание такого адаптера привязывает к конкретной системе автоматизированного тестирования – этого мы решили по возможности избегать и взамен использовать существующий программный интерфейс.

Мы дополнили COM-модель преобразованиями из экрана в чертеж и обратно:

```
nanoCAD.Utility.CoordFromPixelToWorld()
nanoCAD.Utility.CoordFromWorldToPixel()
```

и заменили тестировщика в схеме теста на систему автоматизированного тестирования (рис. 4).

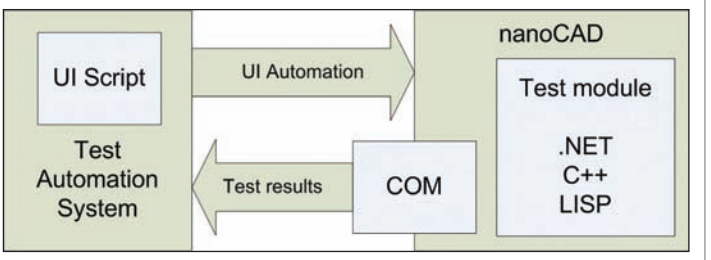

Рис. 4

Теперь наш автоматизированный тест выглядит следующим образом:

```
x_drawing = 3.14;   // При проигрывании: координаты чертежа
y_drawing = 159265;  // преобразуются в экранные координаты
FromWorldToPixel(x_drawing, y_drawing, x_screen, y_screen);

Window.MouseClick(x_screen, y_screen); // При записи было 100, 200,
                                       // а сейчас может уже и нет.
```

Одним из недостатков этого подхода является невозможность автоматически записать такой скрипт – при записи экранные координаты должны динамически преобразовываться в координаты чертежа. При наличии полноценного адаптера эту задачу удалось бы решить.

Главным же недостатком является то, что логика теста разделена на две части. Сам тест загружен в nanoCAD, а скрипт автоматизации – в систему автоматизированного тестирования. Такие тесты тяжело поддерживать и отлаживать: любое изменение логики теста требует синхронного внесения изменений в два различных модуля.

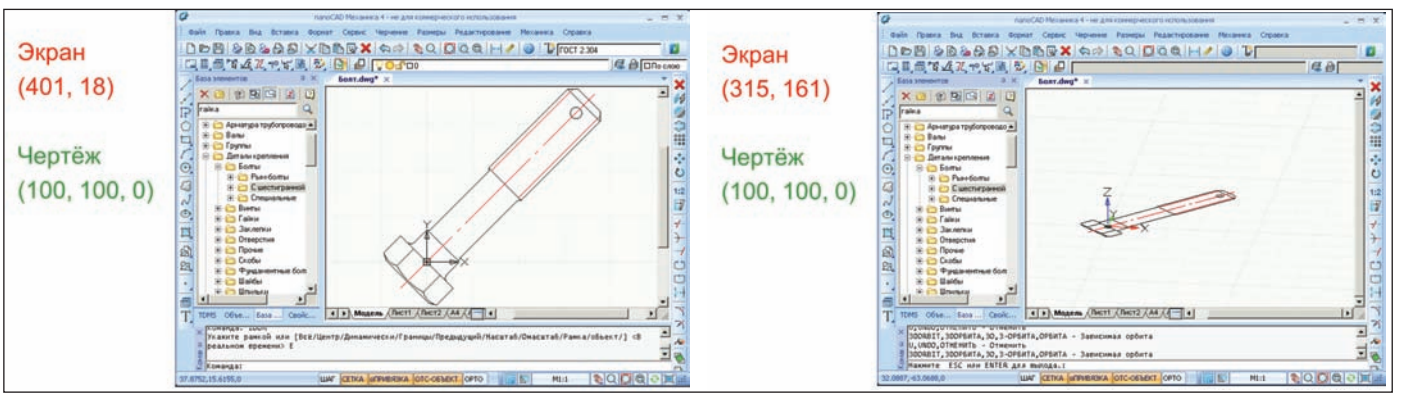

Рис. 3

Мы решили устранить этот недостаток и добиться того, чтобы вся логика теста была в одном модуле. Были рассмотрены два варианта:

1. Логика теста расположена в системе автоматизированного тестирования; в nanoCAD загружен универсальный тест, исполняющий действия, переданные из ведущего автотеста (рис. 5).

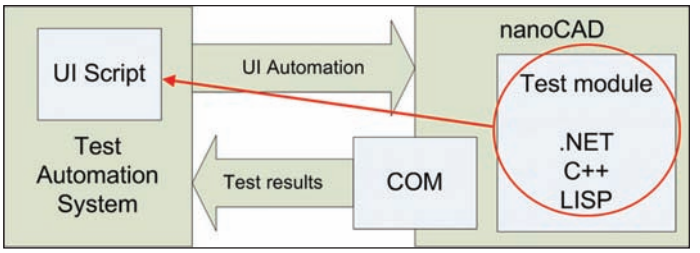

Рис. 5

2. Логика теста расположена в nanoCAD, в систему автоматизированного тестирования загружен универсальный автотест, исполняющий действия, переданные из ведущего теста (рис. 6)

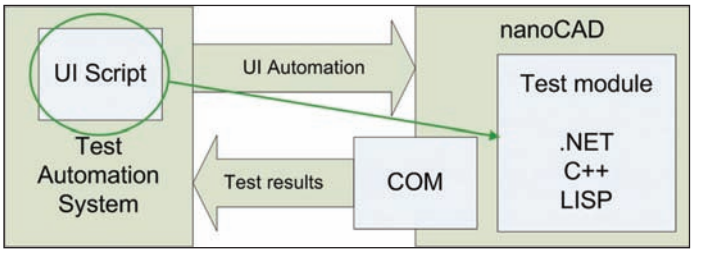

Рис. 6

Первый вариант не подходит, так как лишает нас возможности ручного прогона теста. Мы выбрали второй вариант; библиотеку, которую использует каждый универсальный автотест, мы назвали универсальным проигрывателем (рис. 7).

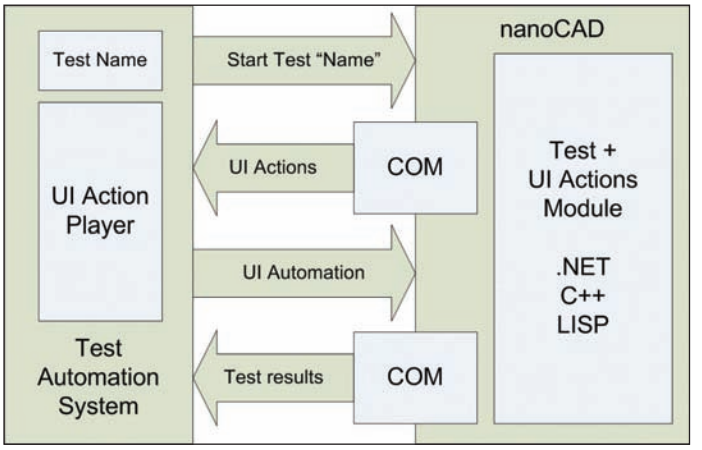

Рис. 7

Несмотря на то что схема теста кажется сложнее предыдущих, поддерживать тесты стало гораздо проще. Теперь сценарий автотеста содержит лишь название теста и код запуска универсального проигрывателя. Универсальный проигрыватель запрашивает у модуля тестов список действий и последовательно их выполняет.

Как это всё выглядит на практике? Вернемся к примеру – тесту функции GetAngle(), который упоминался в начале статьи.

```
Команда: GetAngleTest
...
Укажите угол мышью, первая точка: 0,0,0
Вторая точка: 100,100,0
...
```

Команду GetAngleTest, загруженную в nanoCAD, можно запустить как вручную, так и автоматически.

```
[CommandMethod("GetAngleTest")]
public void GetAngleTest()
{
 // Подготовка действий UI к последующему проигрыванию в универсальном проигрывателе
 testRunner.AddAction(new ActionClickDocument(0, 0, 0));
 testRunner.AddAction(new ActionClickDocument(100, 100, 0));
 testRunner.SendActions();

 // Тест EditorInput.Editor.GetAngle()
 PromptAngleOptions opts = new PromptAngleOptions("Укажите угол мышью");
 PromptDoubleResult pr = this.ed.GetAngle(opts); // Этот вызов мы и тестируем
 thisAssert.IsTrue((pr.Status == PromptStatus.OK) && (pr.Value > 0), "Указание угла мышью. " + pr.ToString());
}
```

Автоматизированный тест RunGetAngleTest лишь исполняет те действия, которые ему прислали из команды GetAngleTest.

```
[TestMethod]
public void RunGetAngleTest()
{
 // Запуск команды 'GETANGLETEST'
  Keyboard.SendKeys(uICommandLineEdit, "GETANGLETEST{Enter}", ModifierKeys.None);

 // Проигрывание действий UI
 this.ProcessUIActions(context);
}
```

В настоящее время универсальный проигрыватель написан только для Visual Studio Coded UI Tests. Но теоретически для переноса автотестов в другую систему достаточно переписать универсальный проигрыватель – мы использовали только базовый функционал, который есть в каждой системе автоматизированного тестирования.

Илья Слободин,
руководитель проекта
"Клуб разработчиков nanoCAD"
E-mail: islobodin@nanocad.ru