Геннадий Алпаев

Автоматизация тестирования от «А» до «Ы»

- *Функциональное и регрессионное тестирование*
- *Обзор популярных инструментов*
- *Лучшие практики и подходы*
- *Пример создания проекта с нуля*

С примерами на TestComplete

Кратко и по существу

Понятным языком

2012

Оглавление

Введение

Вообще-то сначала я хотел назвать эту книгу «Автоматизация тестирования от ААА до ЫЫЫ», имея ввиду «ААА! У НАС СОВСЕМ НЕТ АВТОМАТИЗАЦИИ!» и «ЫЫЫ, ОНО РАБОТАЕТ!», но потом решил немного упростить название. Название «От А до Ы» говорит само за себя: я постарался раскрыть все темы, связанные с автоматизацией, однако всего рассказать невозможно, именно поэтому книга не называется «От А до Я».

Автоматизация тестирования – сравнительно молодая отрасль разработки программного обеспечения (ПО). Каждый день появляются новые решения новых проблем, предлагаются новые подходы, разрабатываются новые инструменты, однако есть и уже устоявшиеся наработки, доказавшие свою пользу за время существования. О них и пойдет речь в этом учебнике.

Я постарался написать эту книгу как можно более простым языком, избегая формальных определений, формул и вычислений везде, где это возможно. Также я старался следовать принципу «краткость – сестра таланта» и не использовать туманных рассуждений и прочих излишеств, от которых в голове читателя остаётся непонятная мешанина слов и никакого понимания.

В главах, посвящённых практическим вопросам автоматизации, я использовал TestComplete по нескольким причинам:

- я знаком с ним лучше всего;
- он наиболее популярный *коммерческий* инструмент в русскоязычной среде;
- в отличие от бесплатных инструментов содержит все компоненты, необходимые для обучения.

Скачать его можно по ссылке:

TestComplete – http://smartbear.com/products/qa-tools/automated-testing/

Начиная со следующей главы я веду речь от множественного числа (используя местоимение «мы» вместо «я»). Это вовсе не из-за завышенного самомнения; просто за время работы в области автоматизации я работал со многими замечательными людьми, а потому выражаю обычно не своё личное мнение, а мнение и опыт всех этих людей, каждый из которых был хорош по-своему и у которых я многому научился.

Структура учебника

Этот учебник разбит на несколько частей:

- В первой главе мы в общих чертах рассматриваем процесс автоматизации, не углубляясь в подробности. Эта глава будет полезна тем, кто не знает, с чего приступать к автоматизации тестирования на проекте.
- Во второй главе мы подробно рассматриваем различные подходы к автоматизации, которые существуют и используются в современных инструментах автоматизации. Эта глава будет полезна как новичкам, так и тем, у кого есть небольшой опыт автоматизации, чтобы структурировать свои знания.
- В учебнике мы рассматриваем в примерах работу с десктоп-приложениями, однако всё сказанное одинаково применимо и к веб-приложениям, и к мобильным. В третьей главе мы кратко рассматриваем особенности работы с такими приложениями по сравнению с десктопными.
- В четвертой части мы затрагиваем отдельные вопросы, которых не касались в предыдущих главах, а также приводим примеры хороших и плохих подходов и решений в автоматизации тестирования. Эти

разделы будут полезны всем, в том числе опытным автоматизаторам (что-то повторить, что-то вспомнить).

- Пятая глава полностью посвящена созданию smoke-теста с нуля, подробно рассматриваются вопросы разбиения проекта на отдельные части, используются сразу несколько подходов, рассмотренных во 2-й главе.

- Шестая глава посвящена вопросу выбора инструментов для автоматизации, а также приведен список наиболее известных инструментов, которые есть на рынке.

Чего нет в этой книге?

Эта книга даёт детальное описание различных аспектов автоматизации тестирования ПО. Однако при этом предполагается, что читатель обладает базовыми знаниями тестирования, программирования и процесса разработки. Вот некоторые вопросы, которые не будут здесь рассматриваться подробно:

- **Это не учебник по программированию!** В процессе автоматизации мы часто программируем, поэтому читатель должен обладать хотя бы общим пониманием программирования (переменные, циклы, условия, функции, объектно-ориентированное программирование – вот базовый набор понятий, с которыми вы должны быть знакомы).

- **Это не учебник по тестированию!** Так как автоматизируем мы именно процесс тестирования, то знание этой области обязательно (тесткейс, дефект). Если вы слабо разбираетесь в этой области, можете прочитать любую книгу по тестированию (книга Сэма Канера *«Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений»* является, пожалуй, классикой жанра, однако помимо неё написано огромное количество книг по данной тематике. Также здесь мы не касаемся таких вопросов, как планирование и организация процесса тестирования в целом, управление командой тестирования и прочими теоретическими размышлениями. Мы концентрируемся в основном на практических вопросах автоматизации.

- **Это не учебник по инструментам!** В этой книге мы используем TestComplete, кроме того в последней главе мы рассказываем о других инструментах, которые в данный момент есть на рынке. Однако это не учебник по инструментам автоматизации.

В остальном, надеемся, нам удалось создать уникальный учебник, покрывающий практически все аспекты автоматизации тестирования.

Приятного чтения!

1. С чего начать?

В этой главе мы обратимся к вопросам, с которых обычно начинается автоматизация тестирования. Выбор инструмента и его изучение, создание команды, организация кода – все эти вопросы очень важны, так как от них зависит, будет ли автоматизация тестирования в вашей компании успешной или нет. Кроме того, на данном этапе вам, возможно, также покажутся интересными глава 5 и главы 1-2 из 3-ей части учебника.

Так как эта книга посвящена автоматизированному тестированию, мы практически не затрагиваем вопросы организации тестирования. Например, в этой главе предполагается, что вы точно знаете, что именно вы будете автоматизировать (т.е. у вас есть usecase'ы или testcase'ы, план тестирования и т.п.).

1.1 Команда, или «кто должен заниматься автоматизацией?»

Зачастую автоматизация тестирования в компании внедряется следующим образом. Команда тестировщиков не справляется с объемами работ и принимается решение о внедрении автоматизации для облегчения (или даже замены!) ручного тестирования. Эта задача возлагается на кого-то из тестировщиков, который подбирает инструмент, записывает скрипты с помощью средств записи, а через неделю выясняется, что скрипты работают неправильно, так как в приложение были внесены изменения. Скрипты записываются заново (что не было запланировано), потом эта ситуация повторяется несколько раз, на перезаписывание не хватает времени, и в итоге команда решает, что автоматизация слишком накладна и не решает задач, которые хотели решить с её помощью.

Этот подход может быть неправильным по нескольким причинам. Во-первых, тестировщики не всегда знакомы с программированием, а автоматизация – это написание кода, и чем код лучше, тем проще его поддерживать. Во-вторых, инструмент мог быть выбран неудачно для данного проекта. В-третьих, начальные ожидания от автоматизации могли быть попросту завышены. Таких причин может быть очень много.

Вот несколько советов по тому, какими навыками должны обладать автотестеры:

1. **Умение программировать** – обязательно! Автоматизация – это написание кода, а это значит, что придется сталкиваться как с процедурным, так и с объектно-ориентированным программированием, заниматься рефакторингом, разбирать чужой код и многое другое.

2. **Понимание процесса тестирования.** Вы должны понимать, что именно вы автоматизируете, и делать это нужно правильно и эффективно. Например, нет смысла автоматизировать функциональность, которой пользуется 1% пользователей, когда не автоматизированы другие части приложения, которыми пользуются 90% пользователей.

3. **Общее понимание разработки ПО** и различных особенностей операционной системы. Занимаясь автоматизацией, вам вполне возможно придется сталкиваться с такими задачами, как работа с базами данных, вызовы системных функций, взаимодействие с другими приложениями и т.п. Естественно, для того, чтобы воспользоваться этими возможностями, необходимо иметь хотя бы общее представление о том, как это всё работает.

В целом можно сказать, что порог вхождения (т.е. набор навыков, необходимых для работы) в автоматизации тестирования выше, чем в ручном тестировании, но ниже, чем в программировании.

В случае, когда автоматизацией тестирования занимается несколько человек, лучше чтобы в команде были 1-2 автоматизатора или программиста уровня Senior для осуществления общего контроля, принятия решений, помощи новичкам и т.п. Кроме того, количество молодых специалистов не должно превышать количество опытных автоматизаторов, чтобы последним не приходилось тратить всё время на обучение и консультации.

С другой стороны, не стоит привлекать программистов к автоматизации тестирования:

- Во-первых, не всем программистам это нравится и в результате они попросту переходят работать в другие компании.
- Во-вторых, программисты обычно пишут автотесты на уровне кода (unit тесты) и нет смысла загружать их дополнительной работой.
- В-третьих, взгляд на программу с точки зрения программиста отличается от точки зрения тестировщика, а потому тесты могут оказаться менее качественными, чем тесты, написанные тестировщиком.

На первых порах также имеет смысл пригласить стороннего консультанта, который поможет с обучением, выбором инструмента, подходами и т.п. Почему-то этот подход не очень распространён в странах бывшего СССР, наши компании предпочитают иметь в штате постоянного сотрудника. Это не всегда является оптимальным выбором, так как для перекупки специалиста такого уровня потребуются довольно большие финансовые затраты, а как консультант такой человек может оказаться незаменимым.

1.2 Когда начинать автоматизацию?

Автоматизацию тестирования лучше начинать как можно раньше, чтобы с ростом приложения постоянно росло количество автотестов.

Однако автоматизировать стоит лишь стабильную функциональность, которая была протестирована вручную и одобрена заказчиком, когда известно, что кардинально эта часть приложения меняться в ближайшем будущем не будет. Иначе можно столкнуться с несколькими проблемами:

1. Заказчик окажется недоволен и часть приложения будет сильно переделана. В этом случае придётся полностью переделывать и автоматические скрипты, тестирующие эту часть приложения.
2. Если функциональность не была протестирована вручную, в автоматических скриптах мы рискуем делать проверки неправильного поведения программы (например, проверка появления неверного значения при расчетах), что в принципе недопустимо.
3. В случае, если некоторая функциональность сделана не полностью, мы можем написать неполные скрипты, пропустив часть проверок. При этом либо придется время от времени возвращаться к уже написанному коду (что неудобно, так как постоянно появляются новые и новые задачи и придётся переключаться между старыми и новыми тестами), либо мы можем просто забыть, что часть проверок в этом месте приложения у нас не сделана, что чревато появлением ошибок, которые не будут отловлены.

Конечно, существует и другой подход. Можно разрабатывать тестовые скрипты одновременно или даже до того, как будет написан код приложения. Однако при этом программистам приходится более формально подходить к написанию кода (использовать строгие соглашения об именовании элементов управления и переменных) и в целом такой подход требует больше времени, чем написание тестов для уже готового функционала.

1.3 Выбор инструмента

Более подробно о выборе инструмента мы поговорим в Части 3 этой книги, сейчас же дадим общие указания и направление.

Прежде всего необходимо понять, что ни один инструмент автоматизации тестирования не удовлетворит ваших нужд на 100%. Это происходит потому, что проектов существует огромное количество и у каждого есть какие-то свои особенности. Производители ПО для автоматизации просто не смогут реализовать все пожелания всех пользователей, да еще и именно в том виде, в каком хочет каждый из них. Поэтому мы говорим о наиболее подходящем инструменте для вашей ситуации.

При выборе инструмента необходимо прежде всего поискать в интернете или спросить на подходящих форумах те, которые, как вам кажется, могут вам подойти. Каждый такой инструмент вы устанавливаете и пробуете с ним работать в течение какого-то времени (неделя, месяц — в зависимости от того, сколько инструментов вы подобрали и сколько времени у вас есть на изучение и выбор).

Вот некоторые параметры, которые стоит учитывать при выборе инструмента:

1. **Цена.** Не всегда лучший инструмент стоит дорого, а бесплатный — вовсе не означает «худший». Сейчас есть большое количество инструментов, и среди бесплатных встречаются в том числе лидеры (например, Selenium – лидер среди средств тестирования веб-приложений).

2. **Поддерживаемые технологии.** Если в вашей компании несколько отделов, в каждом из которых разрабатываются приложения под разные платформы (например, .NET, Java и PHP), и при этом вы хотите разрабатывать общий фреймворк для всех этих проектов, то инструмент должен поддерживать все технологии. Другой подход — использование в каждом проекте отдельного инструмента, со своими библиотеками и подходами.

3. **Удобство использования.** Об этом можно судить по отзывам других пользователей и по своему опыту.

4. **Актуальность инструмента.** Если программа, которую вы выбрали, давно не обновлялась, вполне возможно, что её разработка остановлена (либо будет остановлена в будущем). Это можно узнать у производителей или по отзывам на различных независимых форумах. Нет смысла начинать использовать инструмент, если через полгода его поддержка будет остановлена (особенно если инструмент коммерческий и вы не имеете доступа к его исходным кодам).

5. **Скорость изучения и понимания.** Насколько хорошо организована в программе справочная система, есть ли курс «Для начинающих», является ли он интуитивно понятным и т.п. Если для понимания элементарных вещей вам приходится изучать исходный код и нет никакой справочной системы — в будущем вам придётся туго.

Это далеко не полный список того, на что стоит обращать внимание. Вы можете дополнить его в соответствии с особенностями вашего проекта и подбирать инструмент в соответствии с составленным списком.

1.4 Изучение инструмента (пилотный проект)

Как только вы выбрали один или несколько инструментов, которые кажутся вам подходящими, не торопитесь сразу же пускать его «в дело» и стартовать написание рабочих скриптов. Прежде всего необходимо ознакомиться с инструментом и попробовать его в деле (в конце концов для этого и существуют оценочные версии (evaluation version).

Для этих целей создают так называемый «пилотный» (пробный) проект. Его цель — создать простой проект, который будет тестировать небольшой кусок приложения. После этого у вас сложится определенное впечатление о продукте, который вы собираетесь использовать.

Обратите внимание на несколько важных моментов:

1. **Всегда есть вероятность того, что в результате написания пилотного проекта вы поймете, что данный инструмент вам не подходит.** С одной стороны — не стоит принимать скоропалительных выводов через 2 часа использования и отказываться от инструмента. С другой — не стоит игнорировать проблемы, с которыми вы сталкиваетесь в пилотном проекте, так как эти проблемы в дальнейшем будут возникать во время работы. В этом нет ничего плохого, для этого пилотный проект и предназначен. Лучше обнаружить это раньше, чем через полгода после покупки инструмента.

2. **Код из пилотного проекта необязательно затем использовать в реальном проекте.** Вполне вероятно, что после окончания пилотного проекта у вас будет совершенно другое видение того, как нужно подходить к автоматизации с использованием этого инструмента. Это вполне нормально. Весьма вероятно, что по окончании опробования вы просто удалите пилотный проект и реальный проект начнёте разрабатывать «с нуля». Задача на этом этапе — попробовать и принять решение об использовании.

3. **Выбор того, что автоматизировать в пилотном проекте.** Так как на пилотный проект отводится сравнительно немного времени, необходимо тщательно подобрать сценарии, которые вы хотите автоматизировать. Они не должны быть ни сильно простыми, ни сильно сложными, чтобы с одной стороны не пропустить какие-то важные особенности, а с другой стороны — не зависнуть сильно на трудных задачах. Например, хорошо подойдет для автоматизации часть smoke-теста. Если в тестируемом приложении встречаются сложные элементы управления (таблицы, комбинированные списки, деревья и т.п.), стоит поработать с каждым из них хотя бы поверхностно, чтобы определить, может ли инструмент автоматизации работать с такими элементами вообще. Если не умеет — следует выяснить, есть ли возможность самому написать код для работы с такими элементами (например, в случае кастомных .NET элементов инструмент может предоставлять доступ к внутренним свойствам и методам элементов, с помощью чего можно написать свои функции для работы с ними).

4. **Не стоит сильно затягивать пилотный проект.** Также не стоит углубляться в решение какой-то одной задачи, отставляя в сторону остальные, или начинать создавать многофункциональный фреймворк. Все эти задачи — не для пилотного проекта.

В результате двух последних активностей: выбор инструмента и создание пилотного проекта (или нескольких проектов), вы должны принять решение о том, какой инструмент будете использовать и начинать его серьёзное изучение.

На этапе изучения инструмента часто используется метод Record & Play, если таковой имеется в приложении для автоматизации, когда включается запись, выполняются действия над тестируемым приложением, после чего мы получаем готовый код скрипта. По мере более глубоко изучения инструмента необходимость в использовании записи постепенно отпадает. Подробнее об этом мы поговорим в главе *2.1 Запись и воспроизведение.*

1.5 Создание фреймворка

Фреймворк — это набор функций и/или классов, которые облегчают создание тестовых скриптов и позволяют легко осуществлять действия, которые трудно сделать стандартными средствами инструмента. Фреймворк может включать в себя:

- Систему логирования (если выбранный инструмент не имеет таковой или же её возможности не удовлетворяют вашим нуждам)

- Функции для выполнения проверок (например, для сравнения сложных элементов типа массивов, объектов, таблиц и т.п.)

- Функции для работы с данными (базы данных, таблицы и т.п.)
- Функции для манипулирования с различными типами данных (например, преобразование одного объекта в другой), не предусмотренные возможностями языка программирования
- Функции для преобразования и выполнения ключевых слов (keywords) в код на языке программирования
- Функции, отвечающие за запуск тестов (в определенной последовательности или какого-то набора тестов)
- И многое-многое другое

По сути к фреймворку относится всё, что не имеет прямого отношения к тестам или другому коду, работающему с приложением.

Обычно разработка фреймворка не является отдельной задачей. В самом начале можно написать какие-то функции, которые нам точно понадобятся (например, класс лога или функции для работы с базами данных), однако это вовсе не значит, что на этом можно остановиться. Обычно основная работа по созданию фреймворка приходится на начало проекта, однако по мере написания тестов требования к фреймворку могут меняться, соответственно приходится менять и сам фреймворк.

Фреймворк может быть как очень простым (несколько функций или небольшой класс, позволяющие выводить лог в текстовый файл), который можно написать за один день, так и очень сложным (например, генерирующий отчеты в разных форматах, сохраняющий их в базу данных, позволяющий просматривать отчеты с различной степенью детализации и т.д.), на разработку которого можно потратить месяц и больше. Сложность фреймворка зависит от требований на вашем проекте или в вашей компании и от того, насколько выбранный инструмент соответствует этим требованиям.

После того, как создан фреймворк, можно приступать к написанию тестов.

1.6 Создание тестовых скриптов

Прежде всего необходимо решить, что автоматизировать в первую очередь, а что оставить на потом. Обычно имеет смысл прежде всего автоматизировать наиболее критичные для приложения и простые тесты (они обычно включены в smoke тест), а затем переходить к более экзотичным и сложным задачам. Это имеет смысл по двум причинам:

1. Автоматизация наиболее критичных тестов позволит сэкономить время и силы при проведении регрессионного тестирования.
2. Более сложные задачи могут потребовать лучшего знания инструмента автоматизации. Эти знания будут приобретены в процессе написания более простых начальных тестовых скриптов.

Собственно написание тестов состоит из двух частей:

1. Создание общего кода (common code) приложения
2. Написание тестов

Под общим кодом имеются ввиду любые действия, которые приходится выполнять больше одного раза. Например, процесс логина (открытие странички или запуск приложения, ввод имени пользователя и пароля,

проверка успешного входа в систему) – это очень хороший пример действий, которые стоит вынести в отдельную функцию или метод.

В зависимости от сложности приложения, таких примеров может быть много. Более того, в больших системах общий код, в свою очередь, также может разделяться на подуровни. Например, создание активного пользователя может состоять из нескольких этапов (создание учетной записи, первый вход пользователя в приложение и заполнение им дополнительных полей, не созданных на первом этапе). В таком случае мы можем разделить код на 2 уровня. На первом уровне будут функции низкого уровня (создание пользователя; вход в приложение; заполнение информации), а на втором уровне – одна функция, которая будет вызывать 3 предыдущие, в том числе осуществляя дополнительные действия между ними (например, переход на страницу входа после создания учетной записи).

Такой подход называется «функциональной декомпозицией», более подробно он описан в главе *2.2 Декомпозиция* этого раздела.

При написании тестов и общего кода стоит сразу же придерживаться определенных правил именования и написания кода (coding standards). Можно взять существующие правила, написать свои или же дополнить чьи-то готовые правила своими, специфичными для вашего проекта. Подробнее об этом мы поговорим в главе *5.4 Полезные советы.*

1.7 Запуск тестов

Перед тем, как добавлять написанный тест в набор выполняемых тестов, необходимо убедиться в том, что он работает, причем работает правильно.

- **Работает.** Чтобы убедиться, что тест работает, необходимо несколько раз запустить его и убедиться в том, что он отрабатывает до конца. Бывает так, что тест работает сам по себе, но не работает в связке с другими тестами (например, предыдущий тест оставляет открытым тестируемое приложение, а новый тест пытается открыть его снова). Поэтому лучше запустить несколько раз тест в одиночном режиме, а затем несколько раз в связке с другими тестами, чтобы убедиться, что новый тест «вписывается» в существующий набор тестовых скриптов.

- **Работает правильно.** В тестах мы выполняем различные проверки, для этого мы их и пишем. Поэтому новый тест необходимо также проверить на корректность работы. Например, если где-то в тесте мы нажимаем кнопку и проверяем, что появилось сообщение, можно поставить задержку на несколько секунд сразу после нажатия кнопки и закрыть появившееся окно вручную При этом тест должен выдать ошибку. Другой пример - проверки (verifications). Если мы выполняем в тесте проверку того, что в текстовом поле находится определённый текст, можно в скрипте изменить проверяемый текст и убедиться, что скрипт выдаёт в логе ошибку.

После выполнения всех проверок не забывайте убирать временные изменения, которые добавляли для проверки правильной работоспособности скрипта.

Теперь можно перейти к запускам тестов. Как и когда их запускать? Ответ на этот вопрос сильно зависит от количества автоматических тестов и того, как у вас в проекте построен процесс разработки. Вот примеры того, как можно выполнять запуски.

- **Запуск всех тестов вручную.** Одной командой запускается весь набор тестов. Обычно это обязательно делать перед релизом, чтобы полностью проверить работоспособность тестируемого приложения.

- **Запуск тестов частями вручную.** Можно разбить все тесты на группы по функциональности и запускать только те группы, чью функциональность необходимо проверить в данный момент.

- **Smoke test.** Запуск поверхностного теста, который проверяет общую работоспособность приложения. Обычно эти тесты проходят быстро и проверяют, что основные модули приложения доступны пользователю, но не выполняют каких-либо специальных проверок функциональности. Такой тест можно запускать после каждого билда автоматически (например, сделав его частью плана continuous integration).

- **Автоматический запуск всех тестов.** Можно на отдельном компьютере настроить автоматический запуск тестов (с помощью встроенных средств инструмента либо с помощью возможностей операционной системы) в определенное время. Здесь огромное пространство для фантазии: можно запускать все тесты по ночам, можно запускать тесты непрерывно (т.е. после прохождения всех тестов скачивать с репозитория последнюю версия приложения и тестов и запускать их снова), можно каждую ночь запускать набор основных тестов, а по выходным – всего набора скриптов и т.д.

Кроме всего вышеперечисленного, некоторые инструменты автоматизации позволяют делать распределенный запуск на нескольких компьютерах сразу. Например, вы можете запускать один и тот же набор тестов на разных версиях операционной системы или же распределять тесты между разными компьютерами. Даже если инструмент не обладает подобными встроенными возможностями, такую схему запусков можно настроить самим (например, создав несколько файлов запуска, каждый из которых запускается на отдельном компьютере, а набор тестов для каждого компьютера определять самостоятельно). В случае распределенных запусков, логии скриптов могут складываться куда-то каждым компьютером, или же один компьютер может выполнять роль сервера, собирая логии со всех рабочих машин в сети.

Если компьютер один, а тестов много, можно установить на него виртуальные машины (с помощью программ типа Virtual Box, VMware Workstation или MS Virtual PC) и запускать тесты там.

1.8 Просмотр и анализ результатов

После того, как скрипты отработали, нам необходимо знать результаты пройденных тестов. Для просмотра этих результатов используются логи – отчеты, в которых отображается информация по всем тестам.

Логи должны содержать всю информацию о пройденных шагах: какие опции выбирались, какие данные вводились, какие результаты были получены и т.п., однако самое главное, что должно быть в логе – ошибки. Ошибки должны быть легко обнаруживаемы и к ним должна прилагаться вся необходимая информация для воспроизведения (шаги, данные и т.д.).

Ошибка в логе скрипта – это не всегда ошибка приложения. Довольно часто ошибки в логе свидетельствуют о каких-то недоработках в самих скриптах, поэтому будьте осторожны с такими вещами, как автоматическая регистрация ошибок в багтрекере (по крайней мере нет смысла сразу же отправлять такие ошибки программистам). Бывает так, что достаточно просто перезапустить скрипт с ошибкой и он отработает.

Кроме того, логи должны быть настраиваемыми. Например, тестировщику может быть нужна информация, которая будет совершенно излишней для менеджера, заказчику информации может понадобиться еще меньше, чем менеджер, а программист, наоборот, потребует подробностей, которые совершенно не нужны тестировщику.

Хранить логи лучше всего там, где к ним могут получить доступ все желающие (например, на файловом сервере). В конце работы скриптов логи также могут быть отправлены по почте всем заинтересованным лицам.

В главе *5.2 Логи автоматических скриптов* мы более подробно поговорим о генерации логов скриптов.

1.9 Заключение

На этом мы заканчиваем вступительную часть и переходим к рассмотрению практических подходов, которые используются в автоматизации тестирования. Для продолжения ознакомления с теоретической частью автоматизации, рекомендуем обратиться к главе 5. Другие вопросы, в которой рассмотрены различные хорошие и плохие практики, а также к части 3 этой книги, где рассматриваются вопросы выбора и использования инструментов автоматизации тестирования.

2. Техники и подходы в автоматизации

Мы приступаем ко второй главе, в которой остановимся на практических аспектах автоматизации функционального тестирования. Если до этого мы говорили о том, что нужно делать, то теперь поговорим о том, как это можно сделать.

Для примеров мы выбрали инструмент компании SmartBear TestComplete, так как он является наиболее популярным инструментом в русскоязычной среде. TestComplete позволяет использовать несколько языков для создания скриптов, среди них – JScript, который мы и будем использовать в наших примерах[1].

Кроме того, вам понадобится понимание того, как TestComplete работает с окнами и другими экранными объектами. Если вы знакомы с TestComplete, то можете смело переходить к следующей главе (*2.1 Запись и воспроизведение*).

В TestComplete все объекты системы (будь то процессы, окна или элементы управления) представлены в виде дерева, которое можно увидеть на специальной панели Object Browser (рис. 2.1).

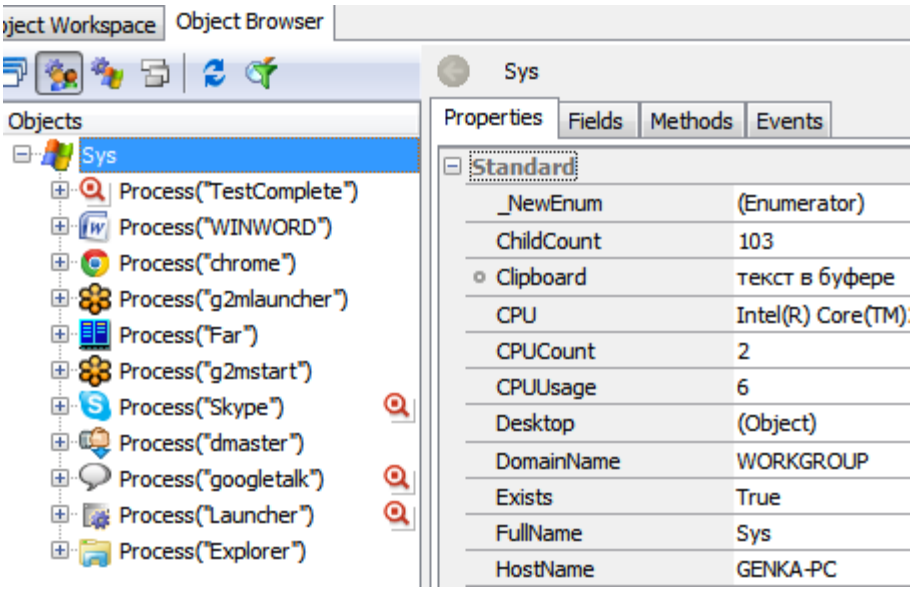


Рис. 2.1. Object Browser в TestComplete

Главным объектом системы является объект `Sys`, используя его свойства можно, например, получить доступ к буферу обмена (свойство `Clipboard`) или другим элементам системы.

Дочерними элементами объекта `Sys` являются процессы (например, TestComplete или WINWORD). У процессов также есть различные свойства (например, Path, который содержит полный путь к запускаемому файлу) и другие дочерние объекты – окна (Window). Некоторые окна являются невидимыми и нам никогда не придется с ними работать, но тем не менее в системе они существуют и TestComplete их отображает в дереве объектов. Соответственно, у каждого окна есть свои дочерние элементы – это, собственно, элементы управления, с которыми работает пользователь (кнопки, списки, текстовые поля и т.д.). С точки зрения системы они также являются «окнами», потому для доступа к ним используется метод Window(как и для окон).

[1] Если вы хотите выполнять примеры, рассмотренные в этом учебнике, вы можете скачать пробную 30-тидневную версию TestComplete на сайте http://smartbear.com/

У каждого объекта (будь то объект `Sys`, процесс или элемент управления) также есть методы (Methods), которые мы можем использовать для этих объектов, например: `Click()` – щелкнуть левой кнопкой мыши по объекту, `Refresh()` – обновить информацию об элементе, `Keys()` – ввести в элемент какой-то текст.

В качестве примера приведем код для нажатия на кнопку «плюс» в окне Калькулятора:

```
Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus").Window("Button", "+").Click()
```

В этом примере:

- **Sys** – главный объект – система
- **Process("CalcPlus")** – обращение к процессу Калькулятора по имени (CalcPlus)
- **Window("SciCalc", "Calculator Plus")** – обращение к главному окну Калькулятора (SciCalc – класс окна, Calculator Plus – его заголовок
- **Window("Button", "+")** – обращение к кнопке «плюс» (Button – класс окна, в нашем случае это кнопка, + - текст на кнопке)
- **Click()** – обращение к методу, который осуществляет клик левой кнопкой мыши

Приведем еще несколько примеров работы с объектами в TestComplete.

`Sys.Process("CalcPlus").Exists` – проверка существования процесса

`Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus", 1).Close()` – закрыть окно Калькулятора

`Sys.Process("CalcPlus").Terminate()` – уничтожить процесс (аналогично выполнению команды *Завершить процесс* в окне *Диспетчер задач*).

Теперь, когда вы имеете представление о том, как TestComplete работает с объектами, можно приступить к изучению различных подходов к автоматизации[2].

2.1 Запись и воспроизведение (Record & Playback)

Запись – это самый простой и одновременно самый ненадежный способ создания тестовых скриптов. Многие программы для автоматизации тестирования позволяют просто включить запись, выполнить действия над тестируемым приложением и в результате получить готовый скрипт.

Создавать рабочие скрипты таким образом не рекомендуется по нескольким причинам:

1. Генерация неоптимального кода. Средства записи не умеют генерировать условия и циклы, без которых можно создать лишь простейшие скрипты.
2. Переменные, которые генерируются в результате записи, зачастую имеют непонятные или громоздкие имена, что усложняет поддержку кода в будущем.
3. Код скриптов, написанный вручную, обычно лучше структурирован и более читабельный.

[2] На самом деле в TestComplete всё устроено несколько сложнее, однако для наших целей такого простого описания будет достаточно. Если вы хотите изучить TestComplete более подробно, вы можете, например, обратиться к бесплатному онлайн-руководству на сайте http://tctutorial.ru/

4. Средства записи позволяют записать лишь действия пользователя с приложением, однако зачастую в скриптах приходится выполнять и другие действия (например, вычисления), которые записать невозможно.

5. Если в приложении используются нестандартные элементы управления (например, ActiveX), средство записи не сможет отследить обращение к их внутренним свойствам и методам. В таких случаях обычно записываются «поверхностные» действия (такие как щелчки мыши по определенным координатам или ввод текста). В результате малейшее изменение в тестируемом приложении (например, изменение размера элемента управления) повлияет на работоспособность скриптов.

Есть, однако, несколько случаев, в которых применение записи оправдано:

1. Изучение инструмента. Во время изучения инструмента запись – самый простой и быстрый способ изучения инструмента: как работать с окнами, элементами управления и т.п.

2. Запись скрипта с целью его дальнейшей модификации вручную. Если запись в инструменте автоматизации достаточно продвинутая, в некоторых случаях бывает проще и быстрее записать некоторые действия, а затем внести небольшие изменения.

3. В некоторых инструментах при записи скрипта одновременно формируется «карта» окна (или страницы) приложения (например, класс или другая структура, содержащие все необходимые элементы). Если запись – самый простой способ создания такой структуры, то имеет смысл воспользоваться ею хотя бы один раз, чтобы создать эту структуру и дальше работать с ней, создавая тестовые скрипты вручную.

К достоинствам записи также можно было бы отнести возможность создания автоматических скриптов людьми, которые совершенно незнакомы с программированием, однако в этой книге мы не рассматриваем этот вариант (см. главу *1.1 Команда, или «кто должен заниматься автоматизацией?»*).

В TestComplete можно записать 2 вида тестов: скриптовые и keyword-driven. О втором подходе мы поговорим далее в этом разделе, а сейчас рассмотрим создание обычного скрипта.

Предположим, нам необходимо в окне Калькулятора нажать последовательно кнопки 1, 2, 3, 4, 5 и 6. Включим в TestComplete запись (меню Test – Record – Record Script), запустим Калькулятор (например, из приложения FAR Manager), нажмем все необходимые кнопки, остановим запись и посмотрим, что мы получили в итоге:

```
function Test1()
{
  var  wndConsoleWindowClass;
  var  wndSciCalc;
  wndConsoleWindowClass = Sys.Process("Far").Window("ConsoleWindowClass", "*");
  wndConsoleWindowClass.Keys("[Enter]");
  wndSciCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wndSciCalc.Window("Button", "1").ClickButton();
  wndSciCalc.Window("Button", "2").ClickButton();
  wndSciCalc.Window("Button", "3").ClickButton();
  wndSciCalc.Window("Button", "4").ClickButton();
  wndSciCalc.Window("Button", "5").ClickButton();
  wndSciCalc.Window("Button", "6").ClickButton();
}
```

Совершенно очевидно, что это – не самый лучший пример кода, а ведь мы всего лишь запустили приложение и нажали 6 кнопок.

Первое, что бросается в глаза, – это запуск Калькулятора. Во время записи в FAR'е была открыта нужная нам папка, поэтому для запуска приложения нам достаточно было нажать [Enter]. Однако при запуске скриптов даже сам FAR может быть не запущен, не говоря уж о том, какая папка окажется в нем текущей. Совершенно очевидно, что запуск приложения нужно делать иначе. Для этого мы воспользуемся специальным объектом TestComplete, который называется **`TestedApps`**.

Далее, нажатие на 6 кнопок подряд лучше было бы организовать в цикле. Тогда в случае, если позже нам придется изменить наш тест, чтобы он нажимал кнопки от 1 до 9, нам придется внести всего одно изменение, вместо того, чтобы копировать последнюю строку 3 раза, а затем в трёх строках менять цифры.

Вот как мог бы выглядеть код, написанный вручную:

```
function Test2()
{
  TestedApps.CalcPlus.Run();
  var CalcWnd = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  for(var i = 1; i <= 6; i++)
  {
    CalcWnd.Window("Button", i).ClickButton();
  }
}
```

Обратите внимание, что кроме общего упрощения кода и улучшения его читабельности, мы также изменили имя переменной окна Калькулятора на более удобное (CalcWnd вместо wndSciCalc).

Однако мы можем пойти дальше и написать функцию, которая будет принимать в качестве параметра строку с цифрами (например, "123456") и нажимать в Калькуляторе соответствующие цифры. Вот пример такой функции:

```
function EnterNumbers(nums)
{
  var CalcWnd = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  for(var i = 0; i < nums.length; i++)
  {
    CalcWnd.Window("Button", nums.substr(i, 1)).ClickButton();
  }
}
```

А вот как теперь будет выглядеть сам тест:

```
function Test3()
{
  TestedApps.CalcPlus.Run();
  EnterNumbers("123456");
}
```

Мы не просто сокращаем объем кода, но также даем себе возможность в дальнейшем вводить необходимые числа очень простым способом в любой момент. В случае с записанным скриптом это невозможно.

Мы рекомендуем всегда писать тесты вручную и стараться не пользоваться средствами записи, кроме случаев, оговоренных выше.

2.2 Декомпозиция (module-based)

Функциональная декомпозиция – это разделение кода скриптов на отдельные функции/методы, выполняющие определенные действия, которые затем вызываются в тестовых скриптах. Попросту говоря, не стоит вываливать все действия прямо в тест, некоторые действия лучше выделить в отдельные функции. Во-первых, это предотвратит в дальнейшем дублирование кода, во-вторых, такие тесты гораздо удобнее читать и модифицировать.

Например, в отдельные функции можно выделить:

- **Навигация.** Любые действия, связанные с навигацией по приложению, скорее всего будут выполняться довольно часто. Даже для простого действия (например, открытие окна поиска) можно написать отдельную функцию. Согласитесь, что строка вида

  ```
  openSearch()
  ```

  выглядит лучше, чем просто обращение к пункту меню

  ```
  Sys.Process("notepad").Window("Notepad", "*").MainMenu.Click("Edit|Find...");
  ```

  Более того, в функции openSearch() мы можем предусмотреть проверку того, что окно Search действительно открылось.

- **Проверки.** Так как смысл тестирования – выполнение проверок, скорее всего у вас будет много проверок в тестах. Например, если вы хотите проверить данные во всех элементах окна, стоит сначала написать универсальную функцию, которая будет принимать в качестве параметров само окно и массив данных, которые мы ожидаем увидеть, и поместить эту функцию во фреймворк, а затем для каждого окна написать отдельную функцию (или метод), которая будет в свою очередь вызывать универсальную функцию проверки с нужными параметрами.

- **Ввод данных.** Если в приложении вам часто приходится работать с данными (заполнять формы или окна необходимыми данными), это тоже лучше организовать отдельными функциями по такому же принципу, как и проверки.

В декомпозицию также входит размещение функций/методов в соответствующих отдельных модулях. Например, можно все высокоуровневые функции проверок поместить в один модуль, а функции заполнения – в другой. Большие приложения состоят как бы из отдельных частей, поэтому имеет смысл выносить в отдельный модуль всё, что связано с конкретной частью функциональности приложения (например, в случае с простым приложением Блокнот мы можем создать отдельные модули File, Edit, Format, View и Help, таким образом разбив наше приложение в соответствии с пунктами меню; другой пример – группировка по функциональности, когда в модуле Search находятся функции для работы с поиском и заменой, в модуле Print – всё, что необходимо для работы с пунктами меню Page Setup и Print, и т.д.).

Здесь нет никаких особых правил, кроме одного: структура проекта должна быть ясной и интуитивно понятной, чтобы даже новичок мог разобраться, где что находится. В разных компаниях могут применяться различные подходы к декомпозиции.

Как уже было сказано выше, мы занимаемся функциональной декомпозицией для того, чтобы упростить себе работу (не дублировать код и писать удобные тесты). В идеале код собственно теста не должен содержать никаких «низкоуровневых» действий (прямых обращений к меню и другим элементам управления, обращений к свойствам и методам окон и т.п.), а только лишь вызовы вспомогательных функций. Однако на практике это не всегда бывает достижимо. Например, если в каком-то тесте нам необходимо нажать на кнопку в окне, причем делается это один единственный раз и больше никогда и нигде не используется – нет

смысла писать для этого действия отдельную функцию. Как и в любом деле, к вопросу создания вспомогательного кода надо подходить без фанатизма.

В целом в функциональной декомпозиции можно выделить 3 уровня:

- **Фреймворк.** Самый «высокоуровневый» код, который может использоваться не только для любых частей приложения, но и даже в нескольких проектах (если инструмент позволяет это делать).
- **Код приложения[3].** На этом уровне хранится код, который используется только для конкретного приложения (проверки, навигация и т.п.).
- **Код тестов.** Собственно тесты, которые мы запускаем.

На каждом уровне может существовать несколько дополнительных подуровней. Например, код фреймворка может быть разделен на код работы с логом, код работы с базами данных, код обработки исключений и т.д.; код приложения может делиться на подуровни, как было описано выше в примере с Блокнотом; тесты также могут относиться к разным группам (например, их можно разбить на группы, каждая из которых будет соответствовать определенной части приложения; другой подход – выделить smoke test и тесты для полного тестирования).

Три простых общих правила при работе с функциями на разных уровнях:

1. Избегайте прямых вызовов функций самого высокого уровня из функций самого низкого (т.е. не стоит в тестах вызывать напрямую функции фреймворка). В некоторых случаях это бывает необходимо или неизбежно, но в целом старайтесь избегать этой практики.
2. Не вызывайте функции более низкого уровня из функций более высокого (то есть, например, не вызывайте функции кода приложения из функций фреймворка или тестовые функции из фреймворка или кода приложения). Здесь не может быть исключений, так как при отступлении от этого правила нарушается целостность проекта (например, если где-то вы вызываете из фреймворка функцию, которая находится на уровне кода приложения, то вы не сможете использовать этот фреймворк в другом проекте).
3. На уровне тестов должны находится самодостаточные функции, которые не должны вызывать другие функции этого же уровня. Это не касается функций фреймворка и кода приложения.

Иногда у вас может возникать желание нарушить третье правило и спроектировать тесты таким образом, чтобы один из них зависел от другого, а значит их нужно либо запускать в определенной последовательности, либо вызывать один тест из другого. Это плохая практика, каждый тест должен иметь возможность запускаться отдельно от остальных, подготавливая в случае необходимости тестируемое приложение, данные и т.п. Этому есть простое объяснение: если у вас есть 3 теста, которые зависят от четвертого, то в случае, если 4й тест прошел с ошибкой, остальные три также будут содержать ошибки, хотя на самом деле с ними всё в порядке.

Пожалуй, наиболее ярким примером такого подхода является процесс тестирования инсталляции приложения. Кажется, что имеет смысл написать три теста (install, modify, uninstall), которые будут запускаться именно в таком порядке, таким образом «помогая» друг другу. В такой ситуации лучше написать вспомогательную функцию на уровне кода приложения, которая будет проверять, установлено ли

[3] Здесь и далее под «кодом приложения» мы подразумеваем код скриптов, который предназначен для работы с тестируемым приложением, а не «исходный код тестируемого приложения».

приложение, и если нет – то устанавливать его наиболее простым способом (т.е. избегая многочисленных проверок, которые мы бы сделали в обычном тесте, например с помощью silent install). Затем эта функция вызывается в начале каждого теста, которому необходимо установленное приложение (в нашем примере это modify и uninstall). Если приложение уже установлено, то мы просто переходим к следующему шагу теста, если нет – быстро устанавливаем его и дальше продолжаем выполнение теста.

В качестве небольшого примера создания кода приложения и тестов, рассмотрим Калькулятор. Предположим, нам необходимо выполнять сложение двух чисел и проверять полученный результат. Для нашего примера мы ограничимся только целыми числами.

Здесь можно сразу выделить несколько необходимых нам методов: ввод числа, нажатие на кнопки «+» и «=», проверка результата и очистка поля. В тесте мы создадим массивы складываемых чисел и ожидаемых результатов, которые затем будем прогонять в цикле. Опять же для простоты мы предполагаем, что на момент запуска теста Калькулятор уже открыт.

Вот как выглядят функции кода приложения.

```
function enterNumber(num)
{
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  for(var i = 0; i < num.length; i++)
  {
    wCalc.Window("Button", num.substr(i, 1)).Click();
  }
}

function verifyResult(expectedResult)
{
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  var actualResult = wCalc.Window("Edit", "").wText.split(",")[0]
  if(expectedResult != actualResult)
  {
    Log.Error("Incorrect result", "Expected: " + expectedResult +
              "\nActual: " + actualResult);
  }
}

function clickPlus()
{
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wCalc.Window("Button", "+").Click();
}

function clickEqual()
{
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wCalc.Window("Button", "=").Click();
}

function clearResult()
```

```
{
  Log.Message("Clearing result pressing [Esc] key");
  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wCalc.Keys("[Esc]");
}
```

А вот как выглядит тест.

```
function TestPlus()
{
  var nums1 = ["5", "13", "124"];
  var nums2 = ["42", "3", "24"];
  var results = ["47", "20", "148"];

  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wCalc.Activate();

  for(var i = 0; i < nums1.length; i++)
  {
    clearResult();
    Log.Message("Adding " + nums1[i] + " and " + nums2[i]);
    enterNumber(nums1[i]);
    clickPlus();
    enterNumber(nums2[i]);
    clickEqual();
    verifyResult(results[i]);
  }
}
```

Мы специально ввели один неправильный результат (20 вместо 16), чтобы продемонстрировать, как будет выглядеть ошибка в логе.

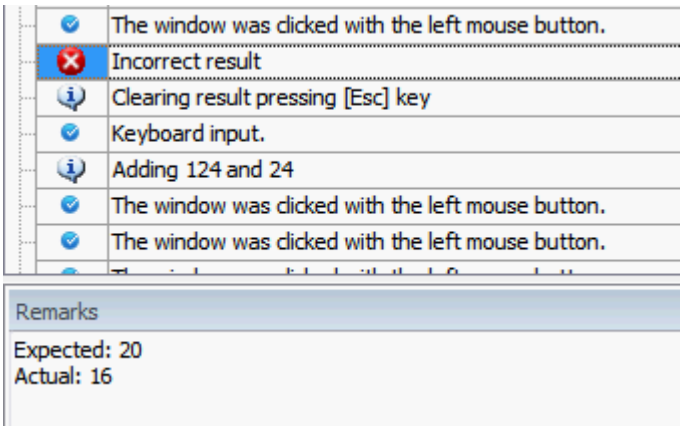


Рис.2.2. Лог выполненного теста

В некоторых компаниях практикуется подход раздельного написания кода приложения и кода тестов, т.е. сначала составляется список функций, которые могут пригодится, затем они пишутся и отлаживаются, и только потом тестировщики приступают к написанию тестов, используя готовые функции. Исходя из нашего

опыта, это не очень удачный подход, так как на этапе проектирования не всегда можно правильно оценить, какие функции мы будем использовать в дальнейшем, а какие – нет, а также как правильно их организовать.

Лучше создавать код приложения параллельно с созданием тестов, когда вы видите конкретную необходимость в той или иной функции. При этом можно сразу создавать и другие похожие функции, если вы видите, что скоро в них возникнет необходимость.

Например, в нашем примере мы создали две функции: `clickPlus()` и `clickEqual()`. Если бы это был реальный проект, то сразу можно было бы создать аналогичные функции для других действий (например, `clickMinus`, `clickDivide` и т.п.), или даже сразу объединить их в одну функцию `clickAction()` с параметром.

2.3 Тесты, управляемые данными (data-driven)

Data-driven (DDT) – это очень мощный и широко используемый подход в автоматизации тестирования, суть которого заключается в отделении данных от кода, помещая все данные в отдельное хранилище и считывая их по мере надобности. На сегодняшний день практически все инструменты, предназначенные для автоматизации тестирования, поддерживают data-driven подход.

Data-driven подход позволяет не только хранить данные удобным для разработки и поддержки способом, но также может быть полезен нетехническим специалистам, отвечающим за качество продукта, так как они всегда могут проконтролировать данные, которые используются в тестировании, и внести изменения в случае необходимости.

Данные хранятся в таблицах в формате, похожем на формат базы данных (т.е. таблица с именованными колонками и строками, каждая строка представляет собой одну единицу данных). В качестве контейнера обычно используются файлы Excel или CSV-файлы. Электронные таблицы удобны тем, что в одном файле можно хранить сразу много таблиц. С другой стороны, CSV-файл является обычным текстовым файлом, который можно просмотреть и отредактировать даже не имея под рукой Excel или другой программы, позволяющей редактировать эти файлы.

Вот примеры данных, которые обычно заносятся в таблицы:

- Входные и выходные данные для различных проверок.
- Данные, по умолчанию хранящиеся в базе данных приложения и использующиеся в тестировании (например, может не быть возможности создавать какие-то сущности непосредственно в приложении, они могут присутствовать в приложении сразу после установки). Подобные данные имеет смысл вносить в таблицы DDT в том случае, если нет доступа к базе данных тестируемого приложения.
- Данные об элементах управления (например, в случае автоматической проверки дизайна и интерфейса).

Отличие данных, используемых в DDT подходе от данных, хранящихся в базе данных в том, что в тестовых скриптах мы только считываем данные из таблиц DDT, но не модифицируем их. Конечно, данные можно хранить в базе данных (таким образом имея к ним полный доступ) или же вносить изменения другими путями (например, подключаясь к таблице через ADO), однако в большинстве случаев данные заносятся в хранилище вручную, а в скриптах только считываются. Именно поэтому в большинстве инструментов, поддерживающих DDT, вы не найдете функций для модификации данных.

В предыдущей главе мы рассмотрели пример с тремя массивами чисел (числа из первого массива num1 складывались с числами второго массива num2, а ожидаемые результаты хранились в массиве results). Давайте переделаем этот пример таким образом, чтобы данные хранились в отдельном файле.

Для начала откроем Excel и создадим в нем страничку Plus с нужными данными.

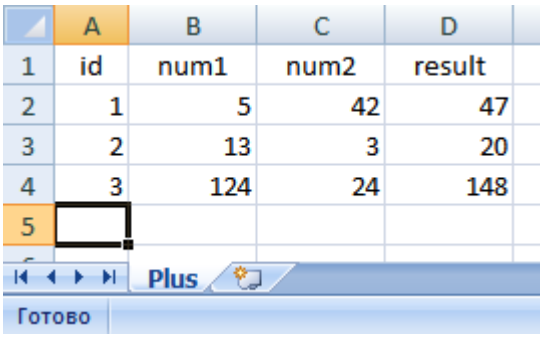


Рис. 2.3. Данные в Excel файле

Первая строка таблицы содержит имена колонок, все остальные строки - данные, соответствующие этим колонкам.

Обратите внимание на колонку id – это ключевое поле (как в базе данных), содержащее порядковый номер строки. Наличие этой колонки не является обязательным, однако дальше мы покажем, как наличие подобной колонки может упростить доступ к данным.

Теперь сохраним файл в папку с проектом и напишем скрипт, который будет считывать цифры из созданной таблицы и вводить их в Калькулятор.

```
//USEUNIT Func_Decomposition


function TestCalcDDT()
{
  var excel = DDT.ExcelDriver(Project.Path + "ddt_data.xls", "Plus");

  var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  wCalc.Activate();

  while(!excel.EOF())
  {
    clearResult();
    Log.Message("Adding " + excel.Value("num1") + " and " +
excel.Value("num2"));
    enterNumber(excel.Value("num1").toString());
    clickPlus();
    enterNumber(excel.Value("num2").toString());
    clickEqual();
    verifyResult(excel.Value("result").toString());
    excel.Next();
  }

  DDT.CloseDriver(excel.Name);
}
```

Как видите, пример из предыдущей главы практически не изменился, однако теперь все данные можно хранить в Excel файле. В дальнейшем, скорее всего, нам бы понадобилось добавлять аналогичные тесты для операций «минус», «умножить» и «разделить». Используя Excel, мы просто добавим новые листы в файл и внесем туда все необходимые данные, а при обращении к драйверу (`DDT.ExcelDriver`) вторым параметром передадим соответствующее имя.

Обратите внимание на использование метода `toString()` при считывании данных из файла. Формат данных в ячейках Excel может быть разным и в случае необходимости считанные данные необходимо преобразовывать. Например, в нашем случае в ячейках хранятся целые числа и если не преобразовывать их в строки, функция `enterNumber()` будет интерпретировать число как пустую строку и числа вводится не будут. Другой способ – всегда хранить в таблицах только строки. При этом вам, возможно, придется выполнять в скриптах обратное преобразование (из строки в другие типы), однако по крайней мере вы будете всегда уверены в том, что при считывании из таблицы получаете строку.

Теперь перейдём к использованию колонки id. Если нам необходимо пройтись по всем записям из таблицы, то она нам не нужна. Однако бывают случаи, когда нам нужны только одна конкретная строка из таблицы (например, если мы используем таблицу для хранения данных, которые жёстко забиты в нашем приложении). Так как TestComplete (как, собственно, и многие другие инструменты) не даёт нам встроенной возможности обратиться к конкретной строке по ключевому полю, мы можем написать собственную функцию, которая это делает.

```
function GetRowById(fileName, sheetName, id)
{
  var row = new Object();
  var excel = DDT.ExcelDriver(fileName, sheetName);

  while(!excel.EOF())
  {
    if(excel.Value("id") == id)
    {
      for(var i = 0; i < excel.ColumnCount; i++)
      {
        colName = excel.ColumnName(i);
        row[colName] = excel.Value(colName);
      }
      DDT.CloseDriver(excel.Name);
      return row;
    }
    excel.Next();
  }

  DDT.CloseDriver(excel.Name);
  return null;
}
```

После этого мы можем легко получить доступ ко всем данным строки примерно таким образом:

```
function TestId()
{
  var row = GetRowById(Project.Path + "ddt_data.xls", "Plus", 2);
  Log.Message(row["result"])
```

```
}
```

Этот пример кода выведет нам в лог число 20, которое хранится в строке 2, колонка result.

Не забывайте закрывать драйвер после использования (метод `DDT.CloseDriver`), а также предусматривайте его закрытие в случае возникновения исключений, так как слишком большое количество открытых драйверов может в итоге привести к переполнению и вы не сможете получить доступ к данным.

2.4 Тесты, управляемые ключевыми словами (keyword-driven)

Keyword тесты – это подход, позволяющий (по крайней мере теоретически) упростить написание тестов за счет того, что тестировщикам не приходится писать код скриптов, а достаточно скомпоновать тесты из имеющихся ключевых слов (keyword'ов) с параметрами, как при использовании конструктора.

На данный момент существует два основных вида keyword тестов. Они не имеют каких-либо названий, поэтому мы их назовём «простой» и «сложный».

«Простые» keyword тесты

«Простые» keyword тесты можно создавать, например, с помощью встроенных средств инструментов TestComplete и Quick Test Pro. Суть их заключается в том, что для каждого элементарного действия, которое мы можем сделать (щелчок мышью, ввод текса, выбор пункта меню) в конструкторе предусмотрено ключевое слово и набор параметров, которые необходимо им передавать (например, в случае ввода текста параметром служит вводимый текст). Фактически каждое ключевое слово в данном случае – это аналог какого-то метода (например, команда Post Screenshot из набора keyword'ов TestComplete – это аналог метода Log.Picture).

Несмотря на своё предназначение (упрощение создания тестов), у «простых» keyword тестов есть ряд серьёзных недостатков:

- Создание keyword теста – процесс довольно трудный (если речь идет не об автоматической записи, конечно), так как для каждого действия необходимо делать большое количество кликов мышью (выбрать подходящее действие, ввести кучу параметров и т.п.).
- Редактирование такого теста – задача также не из простых (по той же причине). Создание циклов, условий и других подобных конструкций в keyword тестах сложнее, чем написание аналогичного кода скрипта.
- Ограниченное число команд. Не все действия можно реализовать с помощью keyword'ов, поэтому в какой-то момент придется либо всё-таки прибегнуть к программированию, либо отказаться от автоматизации каких-то задач.
- Даже при использовании интуитивно понятного редактора keyword тестов человек, не имеющий опыта в программировании, вряд ли сможет создать более-менее сложные и надёжные тесты, поэтому без опытного автоматизатора всё равно не обойтись

Чтобы посмотреть, как выглядит keyword тест в TestComplete, достаточно выбрать пункт меню *Test – Record – Record Keyword Test*, выполнить какие-то действия в тестируемом приложении, после чего остановить запись (аналогично записи обычного скриптового теста). Например, на рисунке 2.4 показан пример теста, который выполняет в Калькуляторе действие «2+3» и проверяет полученный результат.

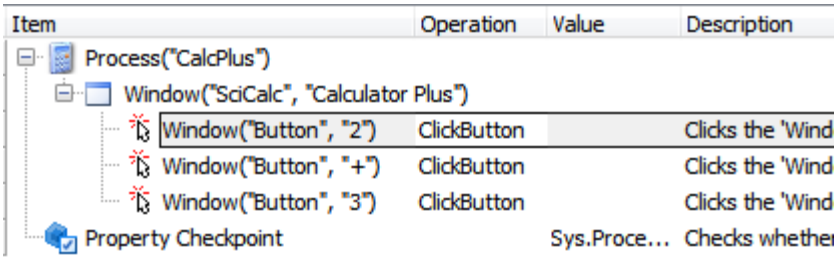


Рис. 2.4. Keyword тест в TestComplete

Если мы теперь, например, захотим выполнить это действие 3 раза, нам придется добавить в нужном месте элемент For Loop, создать для него переменную, задать начальное и конечное значения, а также при необходимости шаг цикла (Рис. 2.5).

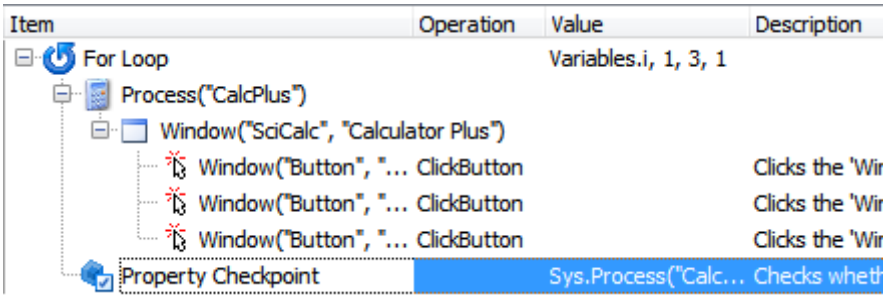


Рис. 2.5. Keyword тест с циклом

Те же самые действия, написанные вручную в обычном скрипте, займут меньше времени и в дальнейшем их проще читать и модифицировать.

«Сложные» keyword тесты

Сложный подход заключается в том, что для создания keyword тестов программист сам определяет, какие именно ключевые слова ему нужны, какие параметры они будут принимать и какие действия выполнять.

Например, одно ключевое слово может служить для вызова нескольких функций, каждая из которых принимает разное количество параметров и в результате которой выполняется большое количество действий. Сами тесты при этом могут храниться в базе данных или в data-driven таблицах.

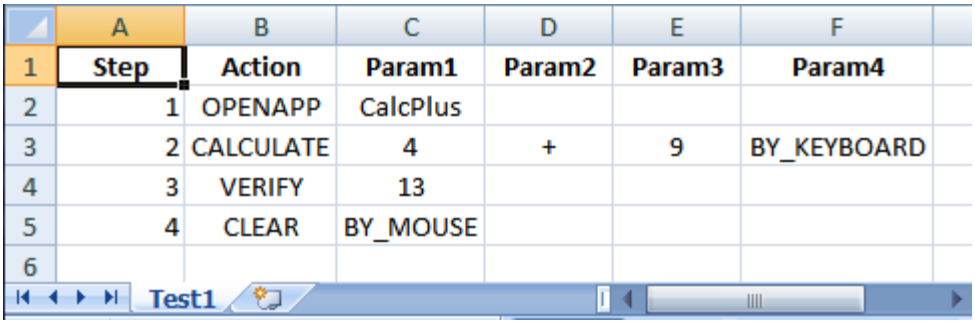
Собственно, «сложные» keyword тесты представляю собой ничто иное как более продвинутый вариант data-driven подхода. При этом часть кода может даже храниться в таблицах и вызываться из скриптов динамически (например, в языке JScript с помощью функции eval можно выполнить любой JavaScript код, переданный в виде строкового параметра), однако отладка таких скриптов сильно затрудняется.

Создание таких тестов состоит из двух частей:

- Создание собственно тестов с помощью ключевых слов. Тесты обычно хранятся в таблице (Excel, база данных и т.п.).
- Создание «драйвера» – набора функций или класса, которые будут выполнять определенные действия в зависимости от выбранного ключевого слова.

Создание драйвера – это самая важная и сложная часть подобных тестов, так как драйвер должен обрабатывать огромное количество ключевых слов и параметров, выполняя соответствующие действия.

В качестве примера создадим простой тесткейс, который будет вычислять в Калькуляторе сумму двух чисел. Вы сами можете решать, что создавать сначала: тесты или драйвер. Мы решили начать с теста. На рисунке 2.6 показано, как он выглядит в Excel файле:



	A	B	C	D	E	F
1	**Step**	**Action**	**Param1**	**Param2**	**Param3**	**Param4**
2	1	OPENAPP	CalcPlus			
3	2	CALCULATE	4	+	9	BY_KEYBOARD
4	3	VERIFY	13			
5	4	CLEAR	BY_MOUSE			
6						

Test1

Рис. 2.6. Пример keyword тесткейса

Имя листа в данном случае (Test1) соответствует имени тесткейса. Соответственно на одном листе у нас будет только один тест.

Первая колонка (Step) – это номер шага. Вторая (Action) – это собственно действие, которое мы хотим выполнить в этом шаге (шаг 1 – открыть приложение; шаг 2 – посчитать; шаг 3 – проверить результат; шаг 4 – очистить результат).

Дальше идут колонки параметров. Так как параметры будут разными в зависимости от действия, колонки названы просто Param1, Param2 и т.д. Например, для действия OPENAPP (открыть приложение) нам нужно передать всего один параметр: имя приложения. Для действия CALCULATE (посчитать) нам нужно передать много параметров (два числа и действие), а также способ вода данных (BY_KEYBOARD означает, что будет эмулироваться ввод данных с клавиатуры, BY_MOUSE – нажатие на кнопки с помощью мыши) и т.д. Если в дальнейшем количество параметров для какого-то действия увеличится, мы просто добавим новую колонку.

Теперь перейдём к написанию драйвера. В нашем случае мы решили не усложнять код и остановились на обычной функции. Кроме того, опять же в целях упрощения задачи, мы не создаем никаких дополнительных функций (например, при открытии приложения можно было бы проверять, не открыто ли оно уже, и если да – то предварительно закрывать его и т.п.), хотя в реальном проекте без этого не обойтись.

```
function DoAction(action)
{
  switch(action)
  {
    case "OPENAPP":
      Log.Message("Running application " + arguments[1]);
      TestedApps.Items(arguments[1]).Run();
      break;

    case "CALCULATE":
      Log.Message("Calculating");
      var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
      if(arguments[4] == "BY_KEYBOARD")
      {
        for(var i = 1; i <= 3; i++)
        {
          wCalc.Keys(arguments[i]);
```

```
      }
      wCalc.Keys("=");
    }
    else  // BY_MOUSE
    {
      for(var i = 1; i <= 3; i++)
      {
        wCalc.Window("Button", arguments[i]).Click();
      }
      wCalc.Window("Button", "=").Click();
    }
    break;

  case "VERIFY":
    Log.Message("Verifying result");
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
    var result = wCalc.Window("Edit", "").wText.split(",")[0];
    if(result != arguments[1])
    {
      Log.Error("Wrong result", "Exp: " + arguments[1] + "\nAct: " + result);
    }
    break;

  case "CLEAR":
    Log.Message("Clearing result");
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
    if(arguments[4] == "BY_KEYBOARD")
    {
      wCalc.Keys("[Esc]");
    }
    else  // BY_MOUSE
    {
        wCalc.Window("Button", "C").Click();
    }
    break;

  default:
    Log.Error("Unknown action: " + action);
    break;
  }
}
```

И теперь нам осталось лишь написать небольшую функцию, которая будет собирать всё это добро вместе и выполнять тест. Вот пример такой функции:

```
function TestRunner()
{
  var test = DDT.ExcelDriver(Project.Path + "keyword_data.xls", "Test1");
  while(!test.EOF())
  {
```

```
    DoAction(test.Value("Action"), test.Value("Param1"), test.Value("Param2"),      test.Value("Param3"),
test.Value("Param4"));

    test.Next();

  }

  DDT.CloseDriver(test.Name);

}
```

Это – очень сильно упрощенный пример keyword-driven подхода. Мы не ставили целью создать полноценный фреймворк, а лишь объяснили основной принцип действия тестов, управляемых ключевыми словами. Если вы захотите воспользоваться этим подходом, вам придется написать гораздо больше дополнительного кода, который будет запускать все ваши тесты. Пожалуй, здесь нельзя выделить плохие или хорошие подходы, вы сами должны решать, насколько сложным будет ваш фреймворк.

Например, можно хранить тесты в одном или в нескольких файлах; можно хранить по одному или по несколько тестов на одной странице Excel; можно задавать для каждого теста приоритет или связывать его с конкретной функциональностью приложения, чтобы иметь возможность запускать тесты выборочно; можно создавать составные ключевые слова, у которых в качестве параметров будут указаны другие ключевые слова и т.д.

Keyword-driven методология – очень мощный и гибкий подход к написанию тестов, однако он требует хорошего уровня программирования от тех, кто будет создавать код драйвера и код запуска тестов.

Основное преимущество этого подхода – отдельное хранение тестов в удобочитаемом виде (не только для автотестера, но и для любого другого человека). Довольно часто этот подход используется для того, чтобы создавать тесты могли не только технически подкованные специалисты, но и те, кто понятия не имеет о программировании. При этом в команде выделяются две основные роли: те, кто пишут код скриптов, и те, кто создает собственно тесты.

Основной недостаток подхода – сложность создания фреймворка.

2.5 Тесты, управляемые объектами (object-driven)

Тесты, управляемые объектами (Object-driven Testing, ODT) – это методология, при использовании которой тестируемое приложение в тестах представлено в виде класса (или нескольких классов). Для выполнения того или иного действия в приложении необходимо вызывать различные методы этого класса.

Некоторые инструменты автоматизации изначально разработаны под эту методологию (ярким примером такого инструмента является SilkTest, в котором приложение представлено в виде winclass'ов и экземпляров этих классов – window). Другие инструменты могут косвенно поддерживать эту методологию благодаря возможностям языка. В TestComplete есть два способа: использование специального объекта ODT или использование языковых возможностей. Мы воспользуемся вторым подходом для демонстрации ODT.

Конечно, в случае ODT подхода никто не запрещает нам использовать одновременно обычные функции, не являющиеся методами класса. ODT всего лишь помогает представить тестируемое приложение и работу с ним в более удобном виде.

Например, вместо функции openApp(), в которой параметром выступает имя запускаемого приложения, мы будем использовать обращение к методу App.Open(); для проверки существования окна – свойство Exists (App.Exists) и т.д.

Рассмотрим простой пример класса Калькулятора. В JavaScript для объявления класса служит то же ключевое слово, что и для объявления функций. Внутри класса с помощью специального синтаксиса объявляются свойства и методы. В нашем примере нам понадобится 3 метода: start(), calculate() и close(), и одно свойство

mainWin, с помощью которого мы будем обращаться к главному окну приложения. Вот объявление этого класса.

```
function Calculator()
{
  this.mainWin = null;

  this.start = function()
  {
    if(!Sys.WaitProcess("CalcPlus").Exists)
    {
      TestedApps.CalcPlus.Run();
    }
    this.mainWin = Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus");
  }

  this.close = function()
  {
    this.mainWin.Close();
  }

  this.calculate = function(expression)
  {
    for(var i = 0; i < expression.length; i++)
    {
      this.mainWin.Keys(expression.substr(i, 1));
    }
    this.mainWin.Keys("=");
    return this.mainWin.Window("Edit", "").wText;
  }
}
```

И простой тест, который считает в Калькуляторе переданное выражение и выводит результат в лог.

```
function TestCalcODT()
{
  // создаем объект класса Калькулятор
  var calc = new Calculator();

  calc.start();
  calc.mainWin.Activate();
  var result = calc.calculate("12+3-8");
  Log.Message(result)
  calc.close();
}
```

Как видите, код теста является интуитивно понятным и легким для написания. Самое важное в данном подходе – хорошо продумать и написать классы, чтобы при работе с ними из тестов не возникало проблем.

Объектами в ODT подходе могут выступать не только тестируемые приложения, но и другие сущности. Например, мы можем создать класс User, который будет представлять собой пользователя,

взаимодействующего с тестируемым приложением и т.п. Здесь всё зависит исключительно от ваших потребностей и фантазии.

Также стоит упомянуть, что при тестировании веб-приложений Object-driven testing зачастую называют подходом, основанным на Page Object'ах. При этом каждая страница веб-приложения описывается отдельным классом со своими свойствами и методами, поэтому по сути это одно и то же.

2.6 Тесты, управляемые моделями (model-based)

Тесты, управляемые моделями (Model-based Testing, MBT) – это отдельная наука, стоящая особняком в ряду подходов к автоматизации тестирования. В MBT используются совершенно другие инструменты и техники, на эту тему написано множество статей, и в рамках нашего учебника мы не сможем достаточно полно рассмотреть этот подход, а дадим лишь краткое описание.

Фактически MBT – это наиболее правильный подход к функциональному тестированию, так как он позволяет провести наиболее полное тестирование функциональности. Вместе с тем, MBT подход является наиболее трудоёмким.

Модель – это математическое описание приложения, описывающее его поведение и/или процесс тестирования приложения. В зависимости от сложности модели, мы можем провести более или менее точное тестирование приложения.

Для того, чтобы объяснить сказанное выше, возьмём в качестве примера Калькулятор.

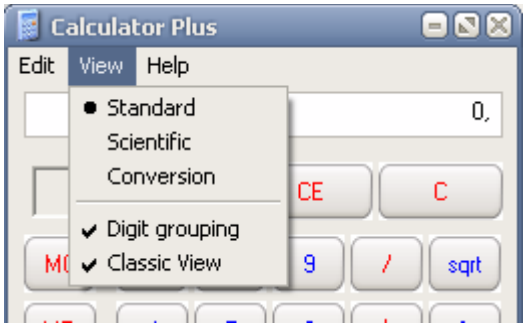


Рис. 2.7. Режимы работы Калькулятора

Как видно на рисунке 2.7, в Калькуляторе есть три режима: Стандартный (Standard), Инженерный (Scientific) и режим Конвертирования (Conversion). У каждого режима доступен свой набор команд и есть свой набор особенностей и ограничений, соответственно для каждого режима нам понадобится своя модель.

Например, в Стандартном режиме у нас есть команда «корень квадратный» (кнопка sqrt), однако перейдя в режим Инженерный мы обнаружим, что такой кнопки нет. Для вычисления квадратного корня нам потребуется возводить число в степень ½.

А вот другой интересный пример с Калькулятором. В детстве был популярен вопрос с подвохом: «сколько будет два плюс два умножить на два?». Первый ответ, приходящий в голову отвечающего, был «8», так как действия в голове выполнялись в том порядке, в каком их называл спрашивающий. И лишь позже мы понимали, что на самом деле правильный ответ – «6», так как операция умножения имеет приоритет перед сложением.

Попробуйте в Калькуляторе выполнить действия «2+2*2=» в Стандартном режиме и в Инженерном. Вы обнаружите, что одна и та же последовательность действий даёт в итоге два разных результата («8» в

Стандартном режиме и «6» - в Инженерном). Это происходит потому, что инженерные калькуляторы учитывают приоритет операций, а обычные – нет[4].

Оба этих примера – демонстрация различных моделей поведения одного и того же приложения.

Другие варианты моделей, которые можно придумать для Калькулятора Windows: модель элементарных операций (+, -, *, /), тригонометрическая модель, модель статистических вычислений, модель работы с памятью, модель систем счисления и т.д.

Для каждой модели можно предусмотреть свою точность. Например, в случае с элементарными операциями, мы можем работать либо с целыми числами, либо с целыми и дробными числами; в случае вычислений логарифмов, корней и работы с тригонометрическими функциями, мы можем определить значимое количество знаков после запятой и т.п.

И, наконец, для каждой модели нам необходимо определить или сгенерировать наборы тестовых данных. Как минимум это должны быть данные из разных классов эквивалентности. Например, в случае тестирования операции сложения, нам нужно складывать положительные и отрицательные, ноль, близкие к нулю и очень большие числа и т.д.

При разработке моделей необходимо хорошо разбираться в функциональности тестируемого приложения и архитектуре компьютера. Например, работа с целыми числами и числами с плавающей запятой на компьютере выполняется по-разному, однако с точки зрения математики сложение и вычитание целых чисел является всего лишь подмножеством аналогичных операций с дробными числами.

Выбор точности модели обычно зависит от тестируемого приложения. Например, высокая точность очень важна в финансовых приложениях и жизненно необходима в медицинских, однако ею можно пренебречь при тестировании портала знакомств (конечно, будет немного странно, если в анкете ваш возраст будет указан 26,03 года, однако в конце концов от этого никто не пострадает).

В модель приложения также могут включаться пути тестирования. Например, процесс инсталляции приложения может включать в себя несколько экранов (папка для установки, устанавливаемые модули, лицензионное соглашение и т.д.). При этом пользователь может двигаться по экранам не только вперёд, но и назад вплоть до самого первого экрана, при этом рассчитывая, что ему не придётся затем заново вводить все данные (или повторно читать 10 страниц лицензионного соглашения). Поэтому для подобных случаев необходимо предусматривать сохранение состояния окна или страницы, чтобы в дальнейшем иметь возможность проверить все данные.

Как видно из примеров выше, даже в случае простых приложений (Калькулятор, инсталлятор) у нас появляется огромное количество вариантов, с которыми необходимо как-то работать, так что же говорить о сложных приложениях? Именно поэтому подход Model-based применяется обычно для приложений, к которым предъявляются повышенные требования качества. И именно поэтому для этих целей разработаны специальные инструменты, которые выходят за рамки рассмотрения этой книги.

Тем не менее, некий простой вариант Model-based тестирования можно реализовать и обычными инструментами, которые обычно используются при регрессионном тестировании, хотя процесс этот будет довольно трудным.

[4] Такой же результат можно получить не только в калькуляторе Windows, но и в «настоящих» настольных калькуляторах.

Для создания model-based тестов вам понадобится создать несколько сущностей, которые будут взаимодействовать между собой:

- **Драйвер интерфейса** – набор функций, осуществляющих взаимодействие с тестируемым приложением (ввод и получение данных). Например, в Калькуляторе ввод данных может осуществляться как с клавиатуры, так и мышью, причем набор действий может быть весьма многообразным. А вот получение данных, в принципе, может оказаться довольно тривиальным и свестись к получению значения текстового поля.

- **Хранилище состояния приложения** – набор параметров, определяющих, в каком состоянии находится в данный момент приложение. Например, система счисления, система измерения углов, находится ли Калькулятор в состоянии ошибки и т.п. В зависимости от текущего состояния приложения, некоторые функции могут быть недоступны.

- **Средства сравнения** эталонных значений с полученными. В случае с Калькулятором здесь могут выполняться проверки не только значения из текстового поля, но также доступность или недоступность некоторых элементов управления (например, кнопки A, B, … E не должны быть доступны при любой системе счисления, кроме шестнадцатеричной).

- **Генератор тестовых данных.** Набор функций, создающих разнообразные данные для выполнения тестирования.

- **Генератор тестов.** Тесты в Model-based подходе являются абстрактными сущностями, т.е. они не завязаны на конкретные данные и/или пути выполнения приложения. Следовательно нам необходимо иметь возможность генерировать конкретные тесты из имеющихся абстрактных описаний.

Как видите, структура подхода довольно сложная, поэтому мы не будем здесь приводить пример реализации (так как даже для тривиального случая он будет весьма впечатляющим).

Отметим лишь, что даже если вы не планируете использовать model-based тестирование в своем проекте, вы можете взять на вооружение некоторые из его подходов. Например, средства сравнения эталонных данных с полученными широко используются при тестировании функциональности CRUD (Create/Read/Update/Delete), когда вы один раз создаете в тесте структуру или запись, а затем заполняете поля формы или окна значениями из этой структуры, после чего открываете запись в приложении и сравниваете сохранённые данные с эталонными, хранящимися в созданной в самом начале структуре.

2.7 Синхронизация выполнения тестов

Синхронизация – это, пожалуй, самый важный вопрос в функциональном тестировании, с которым необходимо разобраться независимо от того, какой инструмент вы используете или какой тип приложения тестируете. В зависимости от того, насколько эффективно вы будете использовать синхронизацию, ваши тестовые скрипты будут стабильными и эффективными.

Под синхронизацией понимается задержка выполнения скрипта до тех пор, пока не произойдет определенное событие (например, появится сообщение или завершится загрузка страницы). В тестах необходимо предусматривать, где именно может потребоваться добавить синхронизацию для того, чтобы скрипт стабильно работал.

Синхронизация нужна для того, чтобы скрипты могли работать при разных условиях, которые влияют на скорость работы приложения (быстродействие компьютера, скорость сетевого подключения, загрузка удалённой базы данных и т.п.). Нужно всегда помнить, что написание и отладка скриптов тестировщиком

могут проводиться на более быстром компьютере, чем компьютер, на котором в дальнейшем они будут выполняться. То же самое касается и других аспектов, таких как скорость сетевого подключения или загруженность базы данных.

Например, представим такую ситуацию. Вы создаёте в приложении новую запись и нажимаете кнопку «Сохранить», после чего на экране появляется сообщение «Данные сохранены» с единственной кнопкой «ОК». Чтобы продолжить работу с приложением, необходимо закрыть окно, нажав на эту кнопку.

Допустим, что при написании теста вы используете локальную базу данных с единственным подключением (своим). Скорость доступа к данным при этом будет практически мгновенной, поэтому данные будут сохранены быстро и сообщение появится через полсекунды.

Вы написали скрипт, который выполняет все необходимые действия и выложили его на тестовый сервер, где он будет запускаться вместе с другими тестами, но при этом тестируемое приложение будет использовать не локальную базу данных, а базу данных, расположенную на другом континенте. Написанный вами скрипт подождёт полсекунды появления сообщения и выдаст ошибку «Сообщение не появилось», после чего сделает скриншот экрана и остановит выполнение. Однако на момент создания скриншота сообщение *уже появится,* что вы и увидите в логе теста.

Отловить такие ошибки бывает очень трудно, особенно если для написания и регулярного запуска тестов используются разные окружения, более того, они могут проявляться лишь в определенные моменты времени, когда вы спите дома, а скрипты работают.

Именно для того, чтобы избежать подобных ситуаций, и используется синхронизация. Вместо того, чтобы пытаться сразу работать с окном, страницей или элементом управления, мы сначала ждём, пока этот элемент станет доступным.

Самый простой способ ожидания – это использование функций типа Sleep() или Delay(), которые попросту ждут определенное время (1, 5, 100 секунд), после чего продолжают выполнение скрипта. Однако это – самый неэффективный способ, так как каждый раз задержки будут одинаковые, хотя ожидаемое событие может произойти как раньше, так и позже. В первом случае мы просто потеряем время на ненужное ожидание, во втором – получим ту же самую ошибку, какую получили бы без использования задержки, после чего придется еще больше увеличить ожидание, тем самым увеличивая время выполнения скрипта.

Для более эффективного решения этой проблемы можно ждать появления окна, ждать окончания загрузки страницы, ждать пока какое-то свойство элемента не станет равным определенному значению, ждать окончания работы процесса и т.д., то есть привязаться к какому-то событию в тестируемом приложении, которое нам даст сигнал, что можно продолжать.

Для практической демонстрации рассмотрим следующий пример. Предположим, что в приложении Калькулятор нам необходимо ждать, пока откроется окно About. Как только оно откроется, мы попросту закроем его с помощью скрипта, а открывать его мы будем вручную в случайный момент. Максимальное время ожидания появления окна пусть будет 20 секунд.

Вот как выглядит **неправильный** пример:

```
function TestDelayWrong()
{
  var pCalc = Sys.Process("CalcPlus");
  pCalc.Window("SciCalc", "Calculator Plus").Activate();
  aqUtils.Delay(20000); // задержка на 20 секунд
  if(!pCalc.Window("*", "About*").Exists)
```

```
  {
    Log.Error("Window 'About' not found");
  }
  else
  {
    pCalc.Window("*", "About*").Close();
  }
}
```

Здесь мы ожидаем 20 секунд всегда, даже если окно About откроется через 1 секунду. А вот правильный пример:

```
function TestDelayCorrect()
{
  var pCalc = Sys.Process("CalcPlus");
  pCalc.Window("SciCalc", "Calculator Plus").Activate();

  if(!pCalc.WaitWindow("*", "About*", 1, 20000).Exists)
  {
    Log.Error("Window 'About' not found");
  }
  else
  {
    pCalc.Window("*", "About*").Close();
  }
}
```

В этом примере мы ждём **максимум** 20 секунд, однако если окно About появится раньше – тут же начнём с ним работать. Подобные Wait-методы в TestComplete есть и для других элементов управления. Кроме того, для ожидания того, что свойство принимает определенное значение, существует метод WaitProperty().

Аналогичные функции или методы существуют во многих инструментах автоматизации, если же такого нет – их нужно обязательно написать самим, иначе будет очень трудно писать эффективные автоматизированные скрипты. Чтобы написать подобную функцию, можно в цикле проверять свойство Exists элемента и возвращать True, если Exists=True, или False при выходе из цикла.

Здесь необходимо упомянуть о нескольких опасных моментах.

- Во-первых, ожидая какое-то событие, не делайте ожидание бесконечным. Обязательно предусматривайте возможность выхода по таймауту, т.е. по прошествии определенного времени, так как приложение может элементарно зависнут, соответственно зависнет и скрипт.

- Во-вторых, старайтесь при ожидании элементов не прописывать время ожидания жёстко (как мы показали в примере выше). Лучше всего определить глобальную переменную для всего проекта и использовать её. В случае, если в дальнейшем вам понадобится увеличить или уменьшить время ожидания для всех операций, вы сделаете это в одном месте, а не сотнях или тысячах строчках кода.

- В-третьих, если вам придется самим писать функции ожидания, предусматривайте небольшие задержки в цикле между проверками, иначе цикл будет выполняться очень часто, занимая всё процессорное время и отбирая его у других приложений. При этом другие приложения, в свою очередь, будут работать медленнее.

Например, пусть у нас есть абстрактная функция ожидания свойства Exists:

```
while(true)
{
  if(object.Exists)
  {
    return True;
  }
}
```

В этом случае обращение к объекты будет происходить очень часто (несколько тысяч раз в секунду) и процесс будет занимать 100% процессорного времени. Лучше этот пример переписать так:

```
while(true)
{
  if(object.Exists)
  {
    return True;
  }
  sleep(1000); // задержка на 1 секунду (1000 миллисекунд)
}
```

В этом примере есть еще одна проблема: бесконечный цикл. Если приложение зависло, этот цикл будет выполняться вечно, поэтому его стоит переписать примерно так:

```
var waittime = 20; // максимальное время ожидание в секундах
while(true)
{
  if(object.Exists)
  {
    return True;
  }
  sleep(1000); // задержка на 1 секунду (1000 миллисекунд)
  waittime -= 1;
  if(waittime < 0)
  {
    return False; // выход по таймауту
  }
}
```

Синхронизация также может быть полезна в следующей ситуации. Предположим, что мы работаем с нестабильным сетевым приложением, у которого в случайные моменты обрывается соединение с сервером, при этом на экране появляется окно с кнопкой «Подключиться заново». В этой ситуации (если позволяет инструмент автоматизации) можно создавать 2 различных потока. Первый из них будет запускать скрипты, а второй – следить, не появилось ли сообщение о разорванном соединении, при этом выполняя необходимые действия (в нашем случае – нажимая кнопку). Конечно, есть вероятность, что пока приложение будет подключаться, в другом потоке уже возникнет ошибка и текущий тест будет содержать ошибку, однако при этом приложение будет восстановлено для работы других тестов.

Ещё один полезный способ использования синхронизации – многовариантные пути прохождения теста. В некоторых случаях в тесте может быть сказано что-то вроде «выполните такие-то действия. Если произойдёт одно событие – сделайте то, а если другое – выполните это». Для таких ситуаций можно, к примеру, написать функцию, которая принимает массив окон, которые могут появиться на экране, и возвращают индекс массива,

который соответствует появившемуся окну. В скрипте при этом необходимо воспользоваться оператором switch (или его аналогом) и прописать необходимые действия для каждого случая.

Вот пример такой функции для приложения Блокнот. В этом примере обрабатывается появление окна и в зависимости от его заголовка (Find, About или какое-то другое) выполняются разные действия (окно Find закрывается кнопкой Cancel, окно About – кнопкой OK, а в остальных случаях выводит сообщение об ошибке).

```
function getWindow(proc, arrWindows)
{
  var waitTime = 10; // максимальное время ожидания в секундах
  while(true)
  {
    for(var i = 0; i < arrWindows.length; i++)
    {
      if(proc.WaitWindow("*", arrWindows[i], 1, 1).Exists)
      {
        return i;
      }
    }
    waitTime -= 1;
    if(!waitTime)
    {
      return -1;
    }
    aqUtils.Delay(1000);
  }
  return -1;
}

function TestMultiWindowWait()
{
  var pNote = Sys.Process("notepad");
  var wNote = pNote.Window("Notepad", "*Notepad");

  wNote.Activate();
  var wndIndex = getWindow(pNote, ["Find", "About*"]);
  switch(wndIndex)
  {
    case 0: /*Find window*/
      pNote.Window("*", "Find").Window("Button", "Cancel").Click();
      break;
    case 1: /*About window*/
      pNote.Window("*", "About*").Window("Button", "OK").Click();
      break;
    default: /*other window or no window*/
      Log.Error("None of the expected windows appeared");
      break;
  }
}
```

Функция `getWindow()` – это та самая функция, которая в цикле перебирает окна заданного процесса и возвращает индекс найденного окна (или -1, если ни одно окно не было найдено), а в тесте `TestMultiWindowWait` мы выполняем действие, специфичное для конкретного окна.

Обратите также внимание, что использовать синхронизацию необходимо с умом, а не просто вставлять её повсюду. Так, например, нет смысла ждать появления кнопки OK в окне, если она там присутствует постоянно с самого начала существования окна, это уже будет ненужной перегрузкой тестов. Как и в любой задаче, здесь необходимо придерживаться «золотой середины».

2.8 Нелинейное тестирование

Большинство приложений (кроме, пожалуй, самых простых) позволяют выполнять одни и те же действия разными способами. Например, чтобы вызвать диалоговое окно открытия файла, можно выбрать пункт меню File – Open, нажать клавиатурную комбинацию Ctrl-o или выбрать пункт меню с помощью клавиатуры (Alt-f-o). Действия разные, а результат один. По хорошему проверять необходимо все варианты.

Другой пример – работа с данными. Предположим, у нас есть поле ввода, в которое мы можем ввести число от 1 до 100 (или выпадающий список с несколькими пунктами). Мы обязаны проверить данные из разных классов эквивалентности (например, что в поле нельзя ввести текст, отрицательные числа или числа больше 100, а также проверить граничные значения 1 и 100), однако внутри интервала от 2 до 99 мы можем выбирать любое число: так как они находятся в одном классе эквивалентности, нет смысла проверять их все. Однако существует такой тип ошибки data related, т.е. ошибки, связанные с неправильными данными. Например, программа будет работать корректно со всеми числами, кроме числа 59. Почему? Этого мы не знаем, например в базу данных попало неправильное значение, однако мы можем даже ни разу не наткнуться на эту ошибку, полагая, что программа ведёт себя одинаково со всеми числами больше единицы и меньше ста.

В большинстве случаев нет смысла писать отдельные тесты для каждого такого случая (так как это нерационально увеличит время выполнения скриптов либо количество используемых ресурсов), однако всё же есть способ тестировать различные пути и данные, не увеличивая при этом количество тестов – это генерация случайных данных.

Предположим, мы тестируем Блокнот и в тестах мы постоянно открываем различные файлы. Чтобы открыть файл, нам необходимо выполнить одно из действий, описанных в примере выше, однако мы можем написать функцию, которая будет выполнять одно из действий случайным образом, либо конкретным способом, если передать дополнительный параметр.

Вот как это будет выглядеть для функции открытия. Прежде всего нам понадобится функция, возвращающая случайное целое число в заданном интервале (так как такой функции нет в JavaScript).

```
function randInt(min, max)
{
  return Math.round(Math.random()*(max-min)+min)
}
```

А теперь собственно функция, которая вызывает диалоговое окно Open в Блокноте либо случайным образом, либо заданным (если передать ей параметр).

```
function openFile(method)
{
  if(method == undefined)
  {
    method = randInt(1, 3);
    Log.Message("Random method selected: " + method);
```

```
  }

  var wNote = Sys.Process("notepad").Window("Notepad", "*Notepad");
  switch(method)
  {
    case 1: // by selecting menu item
      wNote.MainMenu.Click("File|Open...");
      break;
    case 2: // by pressing Ctrl-o
      wNote.Keys("^o");
      break;
    case 3: // by pressing Alt-f-o
      wNote.Keys("~fo");
      break;
    default: // unknown method
      Log.Error("Unknown method: " + method);
      break;
  }
}
```

В следующем примере мы будем бесконечно вызывать появление окна Open случайным образом.

```
function TestOpenFile()
{
  var wNote = Sys.Process("notepad").Window("Notepad", "*Notepad");
  wNote.Activate();
  while(true)
  {
    openFile();
    Sys.Keys("[Esc]");
  }
}
```

А если нам понадобится вызвать окно Open каким-то конкретным способом (например, через меню), нам достаточно вызвать функцию openFile() с соответствующим параметром.

```
openFile(1);
```

Кроме главного преимущества этого подхода (при каждом запуске используется не какой-то один способ, а каждый раз разный), у него есть и недостатки:

- **Сложность написания кода.** Для каждого случая нам необходимо писать отдельный код (в нашем примере это всего одна строка, однако в реальных приложениях мы можем рандомизировать более сложные действия, чем просто выбор пункта меню).

- **Сложность воспроизведения ошибок.** Если во время работы скрипта появилась какая-то ошибка, нам необходимо будет воспроизвести её вручную, для чего пройти все шаги теста в точности как это делал скрипт. Именно для этой цели в функции openFile() мы добавили строку, которая выводит в лог метод, выбранный случайным образом.

- **Сложность воспроизведения ситуации в автоматическом режиме.** Если ошибка была найдена и исправлена программистами, нам необходимо её протестировать, однако мы не знаем заранее, каким именно путём пойдёт скрипт в очередной раз. Поэтому если ошибка некритичная, её

достаточно проверить вручную и продолжать запускать скрипт, генерируя случайные данные. Если же ошибка критичная, можно написать отдельный скрипт, который тестирует именно этот путь, и постоянно запускать два теста: один по случайному пути и второй по жёстко заданному.

В последнем случае можно пойти дальше и создать таблицу (например, DDT или в базе данных), в которой хранить тесты и пути, которые необходимо обязательно проходить, однако такой подход довольно хлопотно организовать (хотя он и является оптимальным).

Всё сказанное выше относительно путей прохождения приложения, справедливо также и для генерации случайных данных.

Этот подход особенно хорошо работает при частых запусках тестов (например, после каждой сборки приложения или при ночных запусках), поэтому он хорошо подойдёт для smoke-тестирования, а вот для acceptance тестирования необходимо постараться проверить как можно больше путей. Но и тут можно оптимизировать процесс. Например, в рассмотренном случае для acceptance тестирования можно при вызове функции openFile() сначала проверить, что каждый из доступных способов вызывает появление на экране окна Open (каждый раз закрывая его – тоже разными способами), и лишь потом продолжить выполнение теста.

2.9 Выбор подходящей методологии

Вполне вероятно, что после прочтения этой главы у вас возникнет вопрос: «Так а какой же подход мне выбрать?». Чаще всего ответом на этот вопрос является такой: комбинируйте несколько подходов, которые вам нравятся, или которые подходят в вашем случае.

Действительно, практически не бывает проектов, в которых использовался бы лишь один подход, обычно приходится их комбинировать. Пожалуй, единственным подходом, который используется отдельно от остальных, является чистый **Record & Play**, т.е. такой подход, при котором вы просто записываете тесты и никак их не редактируете, а в случае возникновения проблем с каким-то тестом попросту перезаписываете этот скрипт. Однако, как мы объясняли выше, этот подход самый плохой из всего, что можно только представить. К этой же категории можно отнести и «простые» keyword-тесты.

Функциональную декомпозицию вам придется использовать всегда (если только вы не хотите получить огромное количество кода, смешанного в одну большую кучу, которая будет быстро разрастаться). Вам придется выносить в отдельные модули функции фреймворка, функции для работы с базами данных, функции для работы с графикой, функции, специфичные для вашего тестируемого приложения; по мере роста количества тестов вам придется помещать в отдельные модули тестовые скрипты, сгруппированные по какому-то признаку (например, в случае с Калькулятором можно выделить в один модуль все тесты, связанные с простыми действиями, в другой – тесты для проверки тригонометрических функций, в третий – тесты для работы с памятью Калькулятора и т.д.).

Чем более сложное тестируемое приложение и чем сложнее ваши тесты – тем сложнее будет структура проекта. Бояться этого не нужно, скорее наоборот: стоит бояться сваливать функции в одну кучу с мыслью «позже разберусь». Старайтесь сразу же выносить функции в соответствующие модули. Лучше иметь модуль с одной единственной функцией, чем поместить эту функцию туда, где ее никто не станет искать.

Object-driven подход имеет смысл использовать в том случае, если вы знакомы с объектно-ориентированным программированием. Тесты, спроектированные таким образом, гораздо более читабельны и более структурированы, чем просто набор функций. В некоторых инструментах (например, в SilkTest) без этого подхода вообще не обойтись, так как на нем построена вся логика написания скриптов.

Data-driven методология используется всегда, когда нужно работать с данными (особенно, если этих данных много), так как в этом случае все ваши данные будут храниться в одном месте и получить к ним доступ будет проще, чем бродить по разным модулям проекта и искать нужный массив или структуру данных. Наиболее очевидные сферы применения тестов, управляемых данными: финансовые, статистические, математические и другие приложения, в которых осуществляется множество вычислений и которые тяжело проверять вручную.

Синхронизация выполнения также используется повсеместно, так как может понадобиться в любом, самом неожиданном, месте (в коде приложения или в коде тестовых скриптов).

Нелинейный подход имеет смысл использовать в коде приложения, при этом предусматривая возможность жесткого задания пути. Например, если у нас есть функция, которая открывает окно Open (с помощью клавиатурной комбинации или мышью), то имеет смысл использовать необязательный параметр. Если параметр передан, то использовать заданный метод открытия, в противном случае выбирать метод случайным образом.

«Сложные» keyword-тесты обычно используются в том случае, когда разработкой кода скриптов и тестов занимаются разные люди. При этом те, кто разрабатывают тесты, могут быть совершенно незнакомы с программированием вообще и со структурой проекта в частности; их задача – создание тестов, а работоспособность этих тестов ложится на плечи разработчиков скриптов.

В конце раздела о функциональном тестировании (глава 6) мы приведем пример создания проекта, который будет включать в себя одновременно несколько описанных выше подходов, что поможет вам лучше прочувствовать процесс создания проекта.

3. Особенности автоматизации Web и мобильных приложений

Веб-приложения стали очень популярными в последние годы и по сложности не уступают настольным приложениям. В книге мы чаще всего приводим примеры скриптов для настольных приложений (Блокнот, Калькулятор и т.п.), однако все подходы, описываемые при этом, в равной степени относятся и к веб-приложениям. В этой главе мы рассмотрим некоторые особенности автоматизации тестирования веб-приложений, с которыми вы можете столкнуться в работе.

3.1 Тестирование в разных браузерах

Зачастую одним из требований к программе является совместимость со всеми браузерами. На данный момент существует пять браузеров (в скобках указан процент использования каждого браузера на конец 2011 года, согласно сайту http://www.w3schools.com):

- Mozilla Firefox (38,7%)
- Google Chrome (32,3%)
- Internet Explorer (21,7%)
- Safari (4,2%)
- Opera (2,4%)

Существует как минимум три уровня сложностями, с которыми придется столкнуться, пытаясь создать универсальные скрипты, которые будут работать в любом браузере:

1) Каждый браузер использует собственную архитектуру и, следовательно, имеет свои особенности работы с веб-страницами. Например, один и тот же тест, обращающийся к элементам страницы по xpath, в Internet Explorer будет работать в несколько раз медленнее, чем в Mozilla Firefox, из-за некоторых особенностей собственно Internet Explorer'а.

2) Каждый браузер регулярно обновляется и обычно существует несколько актуальных версий, широко используемых пользователями.

3) Некоторые браузеры являются кроссплатформенными, т.е. существуют их версии для Windows, Linux и MacOS.

Кроме этого всего, существуют 32-х и 64-х битные операционные системы и приложения, что ещё больше увеличивает сложность тестирования.

Естественно, невозможно протестировать приложение для всех комбинаций упомянутых случаев, но зачастую это и не нужно. Лучше всего автоматизировать тесты под один браузер (например, самый популярный в данный момент или тот, с которым будут работать пользователи тестируемого приложения), а тонкости поведения приложения в разных браузерах оставить для ручного тестирования.

Некоторые инструменты работают с несколькими браузерами. Например, Selenium может работать с любым браузером и на любой системе, а TestComplete поддерживает два браузера (Internet Explorer и Firefox). Однако это вовсе не значит, что вы сможете написать и отладить скрипты под Firefox, а затем просто поменять браузер на другой и безболезненно запустить там все тесты. Скорее всего, для портирования скриптов под другой браузер вам придётся потратить столько же времени, сколько вы потратили на их создание. Другими словами, если вы оцениваете разработку скриптов под один браузер в 100 часов, значит под 2 браузера вам понадобится 200 часов, под 3 браузера – 300 и т.д. Также имейте ввиду, что в дальнейшем вам придётся

запускать и поддерживать все эти скрипты, на что также будет уходить гораздо больше времени, чем в случае работы с одним браузером.

В качестве примера приведём класс Browser, который запускает один из браузеров, поддерживаемых TestComplete'ом, и скрипт, который выполняет простые действия в разных браузерах (открывает страничку http://ya.ru/, выполняет поиск текста и дожидается, пока страница с результатами загрузится).

```
// Browser Class
function Browser(browserName)
{
  while(Sys.WaitProcess(browserName, 1).Exists)
  {
    Sys.Process(browserName).Page("*").Keys("~[F4]");
    aqUtils.Delay(3000);
  }

  TestedApps.Items(browserName).Run();

  return Sys.Process(browserName);
}

function TestInTwoBrowsers()
{
  var browsers = ["iexplore", "firefox"];

  for(var i in browsers)
  {
    var br = new Browser(browsers[i]);
    var page = br.Page("*").ToUrl("http://ya.ru");
    page.Wait();
    page.INPUT.Item("text").Keys("Search string");
    page.INPUT.Item("INPUT").Click();
    page.Wait();
  }
}
```

Этот пример корректно отработает для Internet Explorer и Firefox, но пусть вас не расслабляет эта кажущаяся простота: это очень простой скрипт, выполняющий всего несколько действий; чем больше действий надо будет выполнить – тем сложнее будут тестовые скрипты.

3.2 Загрузка страниц и AJAX-компоненты

При тестировании веб-приложений тестировщики чаще всего сталкиваются с двумя проблемами: скорость загрузки страниц и динамически обновляемое содержимое (AJAX-компоненты). Несмотря на кажущуюся сложность таких приложений, для этих двух проблем есть два простых решения: ожидание и обновление.

Когда мы работаем с настольными приложениями, их код обычно уже загружен в память компьютера, поэтому открытие новых окно происходит практически мгновенно. В случае же веб-приложений, скорость загрузки каждой страницы зависит от её размера и от текущей скорости подключения, поэтому одна и та же страница может загружаться в разное время с разной скоростью.

Чтобы решить эту проблему, в большинстве инструментов для автоматизации тестирования предусмотрены механизмы ожидания загрузки страницы. Например, в TestComplete для этого используется метод Wait:

```
var ff = Sys.Process("firefox");
var page = ff.ToURL("http://vkontakte.ru/search");
page.Wait();
```

В этом примере скрипт не продолжит работу до тех пор, пока страница поиска не загрузится полностью.

Если вы пользуетесь инструментом, в котором нет подобного механизма, вам придется самостоятельно его разработать. Самый простой способ – это ожидание появления на экране какого-либо элемента или же ожидание свойства элемента. Иногда это может быть самый последний элемент на странице (например, футер), а иногда и элемент из середины, который сначала появляется на экране, а затем подгружает динамические данные с сервера. В каждом случае это будет свой уникальный элемент. Поэтому можно написать функцию, которая будет принимать два параметра: адрес, на который необходимо перейти, и элемент, которого нужно дождаться.

При ожидании элемента имейте ввиду, что необходимо предусмотреть возможность выхода из бесконечного цикла в том случае, если ожидание затянулось слишком долго (например, если элемент не появляется в течение пяти минут, это, скорее всего, говорит о том, что страница по какой-то причине «зависла»).

Вторая проблема – динамические элементы – решается с помощью обновления данных о страницы каждый раз, когда это необходимо.

Например, на странице поиска сайта Вконтакте есть выпадающий список, в котором можно выбрать страну, после чего появляется новый элемент управления со списком городов этой страны. Более того, сами списки по умолчанию не видны на экране и появляются только после нажатия на соответствующий элемент страницы.

Вот пример записанного и слегка упрощенного скрипта, который выбирает сначала страну, а затем город.

```
function Test1()
{
  var ff = Sys.Process("firefox");
  var page = ff.ToURL("http://vkontakte.ru/search");
  page.Wait();
  // Log.Message(page.LI.Item("LI_3").innerHTML);
  // Log.Message(page.LI.Item("LI_235").innerHTML);
  page.TD.Item("dropdown2").Click();
  page.LI.Item("LI_3").Click();
  // page.Refresh();
  page.TD.Item("dropdown1").Click();
  page.LI.Item("LI_235").Click();
}
```

Обратите внимание на две закомментированных строки вызова метода `Log.Message()`: если их раскомментировать, мы получим в логе ошибки о том, что элементы не найдены, так как ни выпадающего списка со странами, ни списка с городами еще нет на странице.

Следующая закомментированная строка – это команда обновления элементов дерева страницы (`page.Refresh()`). Если не обновить содержимое страницы после выбора страны, TestComplete может «не увидеть» элементы, которых не существовало в момент создания страницы и которые были добавлены динамически.

Во всем остальном работа с AJAX-компонентами не представляет особых трудностей и выглядит так же, как и работа с другими элементами управления.

3.3 Снимки экрана и страницы (скриншоты)

Одной из особенностей веб-приложений является то, что в отличие от настольных приложений, размер веб-страниц по вертикали может быть больше размера экрана и пользователю приходится прокручивать содержимое с помощью скроллера.

Если при этом необходимо сделать снимок экрана (например, в случае возникновения ошибки), на снимок может не попасть важная информация со страницы, которая в данный момент скрыта от пользователя.

Решение этой проблемы – создание снимка страницы вместо снимка экрана. При этом необходимо сделать снимок видимой области страницы, затем прокрутить страницу вниз, снова сделать снимок и т.д., после чего «склеить» полученные снимки в одну картинку.

Многие инструменты позволяют это делать автоматически. Например, в TestComplete есть отдельный метод Picture(), который позволяет сделать снимок любого объекта (будь то окно, элемент управления или Рабочий стол), а для объектов Page есть дополнительный метод PagePicture(), который делает снимок всей веб-страницы, прокручивая ее в случае необходимости.

В следующем примере мы выводи в лог две разных картинки: снимок экрана и снимок страницы.

```
function TestScreenshots()
{
  var ff = Sys.Process("firefox");
  var page = ff.ToURL("http://smartbear.com");
  page.Wait();
  Log.Picture(Sys.Desktop.Picture());
  Log.Picture(page.PagePicture());
}
```

В случае создания снимков страниц вместо снимков экрана возникает другая проблема: причиной ошибки, вызвавшей сбой работы скрипта, может быть событие, никак не связанное с браузером (например, появление окна Windows Update, блокирующее работу с приложением). В этом случае снимок страницы вам ничего не даст и даже более того: может оказаться, что сделать его в данный момент невозможно.

Поэтому при работе с веб-приложениями имеет смысл делать 2 различных снимка: страницы и экрана. Однако стоит иметь ввиду, что при этом объем занимаемого снимками места на диске увеличится как минимум вдвое.

Одно из решений этой проблемы – делать два типа ошибок (критичные и некритичные, Error и Warning). Некритичными можно считать ошибки браузера и в случае возникновения таких ошибок делать снимок страницы; в случае же возникновения критичной ошибки делать снимок всего экрана.

3.4. Особенности автоматизации мобильных приложений

В последнее время всё более популярными становятся мобильные приложения, соответственно всё чаще возникает вопрос об автоматизации этих приложений.

В общем, все методы, описанные в главе 2 можно с таким же успехом использовать и при автоматизации мобильных приложений, поэтому в этой главе мы просто рассмотрим различные особенности, на которые стоит обратить внимание при работе с мобильными приложениями.

Эмулятор или устройство?

Мы можем тестировать мобильные приложения двумя основными способами: на реальных устройствах или на эмуляторах, которые устанавливаются на компьютер. У каждого подхода есть свои достоинства и недостатки.

При тестировании на реальном устройстве:

4. вы приближаетесь к реальным условиям эксплуатации приложения, что, безусловно, хорошо (так как поведение эмулятора может существенно отличаться от поведения устройства)
5. скорость работы приложения на реальном устройстве обычно выше, чем на эмуляторе.

В случае использования эмулятора, однако, можно тестировать сценарии, которые невозможно протестировать на реальном устройстве (при условии, что эмулятор позволяет эмулировать такие действия):

4. поведение приложения при входящем телефонном звонке или СМС
5. поведение в случае разряженной батареи
6. работа в вертикальном и горизонтальном режимах

Какой же способ выбрать? Как всегда, ответ лежит где-то посредине: использовать оба подхода в зависимости от задачи. Например, большую часть тестов можно писать для выполнения на устройстве, а на эмуляторе тестировать только те сценарии, которые невозможно протестировать на телефоне.

Если речь идет о веб-приложении, заточенном под мобильные устройства, возможно их можно тестировать на обычном компьютере с аналогичным браузером (например, в устройствах Apple используется браузер Safari, который также доступен для обычных компьютеров, то же самое касается Google Chrome для Android). В этом случае нужно также учитывать, что возможности мобильной и десктопной версий браузеров отличаются.

Кроме того, следует учесть, что некоторые вещи все равно не удастся автоматизировать (например, работа с GPS или акселерометром). Также для работы автоматических тестов на реальном устройстве зачастую приходится вносить изменения в само устройство (например, взлом — jailbreak – устройств Apple), что может противоречить условиям тестирования конкретного приложения.

Использование сторонних сервисов

В случае, когда необходимо протестировать приложение на множестве различных устройств, на помощь приходят сервисы типа Device Anywhere, которые имеют в своем распоряжении большой парк устройств и предоставляют платный доступ к ним через Интернет. В некоторых случаях такие сервисы даже позволяют запускать автоматизированные тесты на своих устройствах.

Однако для автоматизированного тестирования подобные сервисы дороговаты (по крайней мере на момент написания этой книги, начало 2012 года), так как тесты необходимо запускать регулярно и в итоге автоматизация в таком виде становится невыгодной. Поэтому использование подобных сервисов лучше оставить для единичных ручных проверок.

Ещё одной сложностью тестирования мобильных приложений является тот факт, что на данный момент нет универсальных инструментов, позволяющих использовать один и тот же скрипт для тестирования одного приложения под разными платформами, поэтому если тестируемое приложение разрабатывается под несколько платформ одновременно (например, iOS и Android), то вам придется поддерживать разные версии одних и тех же тестов для каждой платформы (это не касается веб-приложений, их как раз можно разрабатывать так, чтобы запускать на всех устройствах).

Например, выбранный нами для примеров в этом разделе TestComplete поддерживает тестирование только Windows Mobile приложений, а что делать, ЕСли нам нужно протестировать iOS, Android или Blackberry приложение? Мы можем воспользоваться инструментом компании Experitest – SeeTest Studio, который позволяет записать действия на любом мобильном устройстве (как реальном, так и эмуляторе), а затем экспортировать скрипт в VBScript для TestComplete. Вот, например, как выглядит простой скрипт для TestComplete, записанный с помощью SeeTest на эмуляторе Android.

```
Sub Run()

  Set client = dotNET.experitestClient.Client.zctor()

  client.Connect "127.0.0.1", 8888

  client.SetProjectBaseDirectory "C:\\Users\\genka\\workspace\\project2"

  client.SetApplicationTitle "desktop"

  Report client

  client.Click "default", "element 5", 0, 1

  Report client

  client.Click "default", "element 6", 0, 1

  Report client

  client.SetApplicationTitle "5554:TestComplete"

  Report client

  client.Click "default", "element 3", 0, 1

  Report client

  client.Click "default", "element 1", 0, 1

  Report client

  client.Click "default", "element 7", 0, 1

  Report client

  client.SetApplicationTitle "desktop"

  Report client

  client.Click "default", "element 4", 0, 1

  Report client

End Sub
```

Также при автоматизации мобильных приложений следует учитывать размер проекта. Так как обычно (хотя и не всегда) мобильные приложения являются небольшими и сроки их разработки гораздо меньше, чем разработка настольных или веб-приложений, не стоит тратить слишком много времени на разработку сложного фреймворка, так как может не остаться времени на разработку собственно тестов.

4. Другие вопросы

В этой главе мы более подробно остановимся на некоторых вопросах, не имеющих прямого отношения к созданию автоматических тестов, однако весьма важных в автоматизации в целом.

4.1 Окупаемость автоматизации

Насколько выгодно заниматься автоматизацией функционального тестирования с точки зрения бюджета? Окупается ли она и приносит ли желаемый результат?

Сначала дадим ответ: да, выгодно, но только при условии, что всё делается с умом.

Посчитать в цифрах это немного сложнее и вероятность ошибки может быть высокой, так как не все может быть учтено, но мы всё же попробуем. Данная методика не претендует на точность, так как полна допущений и предположений, и сделана скорее для примера, который вы сможете адаптировать под свои условия.

В примере предполагается, что мы имеем дело со стандартными рабочими условиями (40 рабочих часов в неделю) и что каждую неделю мы получаем релиз продукта, который необходимо протестировать.

Итак, предположим, что я тестировщик и зарабатываю $10/час.

У меня есть тест, в котором необходимо выполнить несколько действий и несколько проверок, на что у меня уходит 15 минут при ручном тестировании, т.е. стоимость прохождения теста составляет $2,5.

Чтобы написать тест, который делает то же самое автоматически, мне требуется 1 час. Работать такой тест будет 1 минуту.

Если у нас есть 100 аналогичных тестов, их прохождение вручную требует 25 часов, т.е. около 3-х рабочих дней. Для того, чтобы автоматизировать эти тесты, понадобится 100 часов. Здесь лучше увеличить ожидаемое время раза в 1,5, чтобы предусмотреть возможные риски. Примем время, необходимое для автоматизации, равным 170 часов, т.е. 1 месяц.

Таким образом, стоимость разработки тестов будет равна $1700, после чего тесты можно запускать, предположив, что ту же работу, которую мы раньше выполняли вручную, выполняют тесты. Причем вместо 25-ти часов в неделю время на их прохождение занимает 100 минут (округлим до 2-х часов).

В итоге каждую неделю мы экономим 23 часа, что при стоимости моего времени $10 даёт $230 в неделю или около $900 в месяц.

Однако нельзя считать, что мы экономим именно $900, так как тесты необходимо поддерживать и в случае необходимости модифицировать, результаты их прохождения нужно проверять, иногда тесты приходится перезапускать. Пусть на это у нас уходит 1 день в неделю, т.е. 8 часов по $10 за 4 недели дает в итоге около $300. В результате мы получаем, что чистой экономии в месяц при использовании автоматизации выходит около $600.

Так как изначально на разработку у нас ушёл месяц, мы можем предположить, что затраты на автоматизацию окупятся через 2 месяца, если учитывать только трудовые затраты.

Если же, к примеру, для автоматизации был приобретен инструмент стоимостью $10.000, то нетрудно подсчитать, что автоматизация ориентировочно окупится через 18 месяцев.

Если вас или ваше руководство это не устраивает, можно рассмотреть другие вопросы, чью стоимость трудно учесть в расчетах:

- В нашем примере мы брали идеальный 8-ми часовой рабочий день, однако совершенно очевидно, что ни один человек не в состоянии продуктивно работать 8 часов подряд (если ваш руководитель не признаёт этот факт, мы рекомендуем поискать другое место работы). Если учесть перерывы на чай/кофе/перекуры/анекдоты, то можно смело уменьшить реальное рабочее время до 6-ти часов, что дает около 4-х реальных дней на прохождение 100 тестов и экономию около $1000 в месяц, что сокращает срок окупаемости в последнем примере до 11 месяцев вместо 18-ти.

- Скрипты не подвержены человеческому фактору и один раз написанные правильно, они никогда не ошибутся и не пропустят «случайно», как это может сделать человек (особенно уставший). Скрипты могут работать по ночам или на выходных, а могут работать на отдельном компьютере, в то время, пока тестировщик проверяет ту часть функционала, которая не покрыта тестами.

- Некоторые вещи невозможно или невероятно трудно проверить вручную. Чаще всего к таким задачам относят нагрузочное тестирование, но есть и другие примеры: сравнение изображений, перебор большого числа входных и выходных параметров, одновременная проверка работы сразу в нескольких частях приложения с разных компьютеров и другие. Всё это гораздо проще при использовании автоматизированных скриптов.

Пожалуй, остался лишь ещё один важный параметр, который мы не учли при расчетах. Если моя зарплата при ручном тестировании равна $10/час, то при переквалификации на автоматизацию компании придется платить мне больше (примерно раза в 1,5), так как круг моих обязанностей и ответственности, как и уровень знаний, возрастает.

Предлагаем читателю в качестве упражнения рассчитать время окупаемости с учетом повышения зарплаты тестировщику.

4.2 Логи и отчёты в автоматизации

Лог, который формируется в результате работы автоматических скриптов, является самым важным инструментом, по которому можно судить о результатах запуска скриптов.

В разных инструментах, предназначенных для автоматизации тестирования, логи организованы по-разному. Например, в TestComplete лог хранится в формате XML, в SilkTest используется свой бинарный формат, в Selenium можно включить логи, содержащие отладочную информацию, а для генерации пользовательских логов необходимо подключать дополнительные библиотеки и т. д.

Вас вполне могут устраивать встроенные возможности логирования, поставляемые с инструментом, однако чаще всего приходится либо доделывать необходимую функциональность или же писать её «с нуля» под свои нужды. Также можно воспользоваться логгерами, найденными в интернете и выложенными для свободного скачивания.

Так как требования к логу могут быть разными (в зависимости от проекта или компании) — невозможно сказать, что какой-то подход хорош, а какой-то не очень, поэтому мы просто перечислим наиболее удачные и интересные подходы, с которыми сталкивались в своей практике.

6. **Формат лога.** Наиболее универсальным можно считать формат XML, наиболее простым — обычный текстовый файл, наиболее удобным для чтения — формат HTML. Кроме того, необязательно работать только с одним форматом. Лог, например, может формироваться в формате XML для внутренних нужд команды тестирования и разработчиков, после чего конвертироваться в формат HTML для предоставления их менеджменту или заказчику (в случае такой необходимости).

7. **Кодировка.** Кодировку лучше использовать UTF-8.

8. **Снимки экрана (скриншоты).** В случае возникновения ошибки в лог всегда необходимо помещать скриншот экрана (причем всего экрана, а не только приложения, так как на тестируемое приложение и инструмент автоматизации могут влиять и сторонние программы). При работе с веб-приложениями, когда странички в высоту могут быть длиннее высоты экрана, желательно делать снимок всей страницы (например, прокручивая ее в случае необходимости, а затем «склеивая» части).

9. **Фильтры отображения.** Логи могут содержать огромное количество информации, однако чаще всего для анализа нам достаточно просмотреть лишь ошибки, возникшие во время работы скриптов. Очень удобным в таком случае может оказаться возможность отключения показа всех остальных типов сообщений (информационных, отладочных и т. п.).

10. **Информация о проверках.** Когда в скриптах выполняются проверки (т. е. Сравнение эталонных значений со значениями, полученными в приложении), очень хорошей практикой считается помещать ожидаемый и полученный результат (expected и actual) в случае, если проверка не прошла.

11. **Сохранение лога.** Насколько часто сохранять лог — это довольно сложный вопрос. Если делать это слишком часто, работа скриптов может замедлиться. Если делать только в конце запуска скриптов — есть вероятность того, что логи будут утеряны (например, если случился какой-то сбой работы инструмента или же попросту пропало электричество). Поэтому желательно предусмотреть сохранение лога на диск через определенные промежутки времени.

12. **Типы ошибок.** Вполне вероятно, что в своих скриптах вы захотите выделить несколько типов ошибок (ошибки скрипта, ошибки функциональности, ошибки, связанные с тестовым окружением и т. п.). В таком случае каждый тип ошибки необходимо как-то выделять (иконкой, цветом или оформлением).

13. **Хранение логов.** Логи (особенно со множеством скриншотов) могут занимать много места, поэтому необходимо продумать, где их хранить, чтобы с одной стороны они были постоянно доступны всем, кому они могут понадобиться, а с другой — не были обузой, занимая место на диске. Например, можно хранить все логи на отдельном сервере, специально для этого предназначенного. Кроме того, необходимо решить, так ли уж важны логи «промежуточных» запусков (например, ночных), или же достаточно будет сохранять логи быстрого (smoke) теста или приёмо-сдаточного для каждой версии. Кроме того, возможно, логи можно автоматически удалять по истечении какого-то времени.

Главное, запомните: как бы вы ни формировали ваши логи, они должны быть удобны для прочтения их человеком, их должно быть легко обрабатывать с помощью различных программ, в них должно быть легко искать нужную информацию и по ним должно быть возможно восстановить ход работы скрипта (т.е. читая лог вы должны суметь полностью повторить все шаги со 100% точностью).

4.3 Проверка отдельных дефектов

Зачастую бывает так, что тестировщик обнаружил проблему, создал дефект в багтрекинговой системе, программист её исправил, тестировщик проверил и убедился, что проблема исправлена и закрыл дефект. А через некоторое время (день, месяц, год) проблема снова появилась, причём точно такая же.

Это может происходить по нескольким причинам.

1. Если проблема всплыла довольно быстро (в течение дня или недели), скорее всего произошли какие-то проблемы при слиянии разных веток в системе контроля версий. При подобном слиянии, особенно если изменений очень много, ошибки проявляются довольно часто, так как программист не всегда может точно определить, какие именно изменения являются более новыми или более правильными.

2. Если проблема появилась снова через довольно продолжительное время (несколько месяцев), это может быть следствием того, что в первый раз для исправления проблемы программист использовал какой-то нестандартный подход. Через несколько месяцев этот код может быть изменен в результате рефакторинга, что потенциально вернёт старую проблему.

3. Проблема на самом деле не была исправлена изначально и дефект был закрыт по ошибке.

В любом случае мы имеем дело с человеческим фактором, а одна из задач автоматизации тестирования – исключение подобных проблем, поэтому в идеале желательно создавать тесты, которые будут проверять, исправлены ли конкретные дефекты.

К сожалению, сделать это не всегда возможно, так как кроме исправленных дефектов тестировщику необходимо писать тесты для проверки новой функциональности, поэтому мы рекомендуем придерживаться следующих правил при автоматизации проверок отдельных дефектов:

1. Автоматизируйте проверку только наиболее критичных и важных дефектов. Например, если ошибка заключалась в том, что размер шрифта одного единственного элемента отличался на 1 пиксель от того, что указано в спецификации, нет смысла писать для этого отдельный тест, так как это слишком мелкая проблема, которая к тому же вряд ли помешает конечному пользователю.

2. Старайтесь делать эти проверки как можно быстрее. Например, создать тест, который будет работать полчаса и при этом проверит всего один дефект – это неудачный подход.

3. По возможности встраивайте проверки дефектов в имеющиеся тесты. Зачастую старт и предварительная подготовка приложения занимают много времени, при этом уже есть тесты, которые работают с той функциональностью, в которой проявился дефект. Имеет смысл немного изменить код существующего скрипта и вставить проверку где-то в его середину отдельным шагом.

4. Не занимайтесь автоматизацией дефекта, если для написания необходимого кода проверки у вас уйдет много времени (например, больше одного-двух часов). Отступать от этого правила можно, если дефект проявляется в критичном месте приложения и подобная трата времени оправдана, либо если дефект связан со множеством проверок (тогда это попросту равносильно новому тесткейсу).

4.4 Тестирование интерфейса и дизайна

Когда мы говорим об автоматизации функционального тестирования, чаще всего имеется ввиду тестирование именно функциональности: создание, изменение, удаление данных, проверка результатов работы программы и т.п. Иногда, однако имеет смысл проверять также и дизайн приложения. В частности, это имеет смысл в приложениях, в которых к интерфейсу предъявляются повышенные требования, причем автоматизировать такой процесс оказывается легче, чем проверять его вручную.

Для примера можно взять веб-приложение. Обычно в современных веб-приложениях все характеристики элементов (размер шрифта, цвет и начертание) прописываются в стилях, т.е. эти параметры не нужно устанавливать для каждого элемента отдельно, достаточно лишь проверить, что в стилях прописаны верные значения. Сделать это автоматически проще, чем проверять вручную (даже с использованием вспомогательных средств). Дополнительным плюсом здесь будет тот факт, что проверки будут осуществляться при каждом прогоне скриптов, избавляя тестировщиков от рутинной работы.

4.5 Работа с изображениями

В процессе автоматизации иногда возникают ситуации, когда по каким-либо причинам нам приходится работать с изображениями. Это не обязательно должно быть приложение, работающее с графикой, просто

некоторые задачи проще решить, используя изображение окна или элемента управления, вместо того, чтобы пытаться получить необходимую информацию из доступных свойств элемента.

Например, если в приложении рисуется график, причем для его отрисовки используется компонент стороннего производителя, гораздо проще и быстрее будет сравнивать полученное изображение графика с неким эталонным изображением, чем писать громоздкий код, проверяющий правильность отрисовки этого графика.

Прежде чем перейти к практическим аспектам работы с изображениями, давайте выясним, когда и как это целесообразно делать.

6. Сравнение изображений необходимо использовать только в самых крайних случаях, когда без этого не удается обойтись, либо когда задача тестирования заключается именно в работе с изображениями (например, тестирование графического редактора).

7. Сравнение изображений – задача трудоемкая как при написании кода, так и при выполнении скриптов (сравнение происходит медленно).

8. Мельчайшие изменения в тестируемом приложении (например, сдвиг на 1 пиксель, который незаметен для пользователя) обычно приводит к поломке тестовых скриптов и необходимости их переделывания.

9. В некоторых случаях изображения отличаются в разных версиях одной и той же операционной системы. В качестве примера можно привести Windows XP и Windows 7: в них элементы управления выглядят по-разному, что добавляет сложности при написании тестов.

10. Если уж работа с изображением – единственно возможный вариант, по крайней мере сравнивайте не весь экран целиком, а только ту часть, которую действительно необходимо проверить (окно, элемент управления или его часть).

11. Если вам нужно проверить изображение, а сделать это автоматически очень трудно, вы можете просто выводить изображение в лог и просматривать его вручную по окончании работы скриптов. Это, конечно, не будет полностью автоматизированным решением задачи, но лучше так, чем постоянно исправлять и улучшать код проверки.

В зависимости от того, каким инструментом автоматизации вы пользуетесь, в вашем распоряжении могут быть разные инструменты работы с изображениями. Вполне возможно, что вам придется использовать сторонние приложения для некоторых операций.

В наших примерах мы используем TestComplete, в котором возможности для работы с изображениями весьма богаты: мы можем захватывать изображения отдельных элементов управления, сравнивать изображения и даже искать одно изображение внутри другого. Кроме того, мы можем задавать различные допуски при проверках, что также может существенно облегчить задачу. Если в вашем инструменте таких возможностей нет, вы можете попытаться реализовать их сами (что может оказаться довольно трудной задачей), или же воспользоваться сторонними утилитами.

В качестве примера приведем небольшой скрипт, выполненный с помощью TestComplete, и разберем результаты его работы. Этот скрипт рисует параллелограмм с двумя диагоналями в программе MS Paint, а затем пытается определить, что полученное изображение соответствует тому, что мы ожидаем. Для этого мы прдварительно нарисовали (также средствами TestComplete) и сохранили изображение параллелограмма в хранилище Regions. Вод код скрипта:

```
function TestImages()
{
  TestedApps.mspaint.Run();
  var  wPaint, toolbar, canvas;
  wPaint = Sys.Process("mspaint").Window("MSPaintApp", "*");
  wPaint.Restore();
  wPaint.Position(0, 0, 800, 600);
  toolbar = wPaint.Window("AfxWnd42u", "Tools");

  // Задаем размеры окна и области для рисования
  wPaint.Keys("^e");
  Sys.Keys("500[Tab]300[Enter]^a[Del]");
  canvas = wPaint.Window("Afx:1000000:8");
  toolbar.Click(20, 90); // click Pencil tool

  // рисуем параллелограмм с диагоналями
  canvas.Drag(50, 50, 100, 0);
  canvas.Drag(150, 50, 50, 50);
  canvas.Drag(200, 100, -100, 0);
  canvas.Drag(100, 100, -50, -50);
  canvas.Drag(50, 50, 150, 50);
  canvas.Drag(150, 50, -50, 50);
  wPaint.HoverMouse(1, 1);

  // Сохраняем изображение параллелограмма в Stores.Regions
  //Regions.AddPicture(canvas.Picture(50, 50, 151, 51), "parallelogram");

  // Пытаемся выполнить проверку правильности изображения
  var canvas_pic = canvas.Picture();
  if(Regions.FindRegion("parallelogram", canvas_pic) == null)
  {
    Log.Error("Parallelogram not found in canvas");
  }

  Regions.Compare(canvas.Picture(50, 50, 151, 51),
                  Regions.GetPicture("parallelogram"));
  Regions.Compare(canvas.Picture(50, 50, 151, 51),
                  Regions.GetPicture("parallelogram"), false, false, true, 30);
}
```

А вот так будет выглядеть лог работы скрипта.

	The mouse pointer hovered over the window.
	Parallelogram not found in canvas
	The regions are not equal.
	The "Picture1" parameter passed to Regions.Compare.
	The "Picture2" parameter passed to Regions.Compare.
	Regions.Compare result.
	The regions are not equal.
	The "Picture1" parameter passed to Regions.Compare.
	The "Picture2" parameter passed to Regions.Compare.
	Regions.Compare result.

Рис. 5.1. Лог работы скрипта, выполняющего сравнение изображений

Как видно из лога, все проверки завершились неудачей. Давайте разберемся почему так получилось.

Первая проверка (поиск изображения внутри другого с помощью метода FindRegion) просто вернула нам значение null, что говорит о том, что сохраненное ранее изображение не найдено в нарисованном нами. Хотя поиск изображения внутри другого и является удобной функцией, она не дает нам никакой визуальной информации.

Вторая проверка (с помощью метода Compare) также завершилась неудачей, однако она дает нам полезную информацию: изображение, на котором красными точками отображаются пиксели, сравнение которых завершилось неудачей. Теперь мы хотя бы знаем, где именно у нас появляются проблемы при сравнении и можем как-то повлиять на это. В нашем случае проблема появляется из-за того, что большая диагональ каждый раз рисуется немного по-другому, что незаметно для пользователя, однако критично для точных проверок.

И, наконец, третья проверка также завершилась неудачей, хотя мы воспользовались дополнительным параметром Tolerance (в нашем случае он равен 30). Этот параметр позволяет задать наибольшее количество отличающихся пикселей, при которых изображения будут считаться одинаковыми. В нашем примере количество таких пикселей равно 40 (хотя на вашем компьютере результаты могут немного отличаться), поэтому увеличение этого параметра до 40 приведёт к тому, что проверка пройдет успешно.

Как видите, даже с таким простым изображением у нас возникли проблемы. А теперь представьте себе, что вместо двух цветов (как в нашем примере) вам придется работать с полноцветными изображениями, которые на каждой операционной системе будут отображаться немного иначе (даже при установленной одинаковой глубине цветовой гаммы) и вы получите примерное представление о том, насколько сложными могут оказаться такие задачи.

Кроме того, обратите внимание на дополнительные действия, которые нам пришлось сделать, чтобы корректно работать с изображениями: нам пришлось установить строго заданный размер окна и холста; после того, как мы закончили рисование изображения, нам пришлось убрать в сторону курсор мыши (с помощью метода HoverMouse, хотя в TestComplete есть встроенные средства для игнорирования курсора мыши при захвате изображения); и, наконец, нам пришлось указывать точные координаты для захвата изображения и передаче его методу Regions.Compare, что нельзя считать очень удобным подходом.

Именно поэтому мы рекомендуем избегать работы с изображениями в автоматизации тестирования.

4.6 Полезные советы (best practices)

За несколько лет существования автоматизации тестирования были найдены различные подходы, улучшающие или упрощающие процесс внедрения автоматизации. Следование этим советам поможет вам избежать ошибок, типичных для начинающих автоматизаторов, и получить максимально быстрый результат.

1. **Не пользуйтесь средствами записи (Record & Play).** Независимо от того, насколько хорош выбранный вами инструмент, встроенные средства записи в нем все равно сильно ограничены. Средства записи хороши лишь поначалу, когда вы знакомитесь с инструментом, так как они позволяют вам максимально быстро получить представление о том, как инструмент взаимодействует с тестируемым приложением. После того, как вы разработаете свой фреймворк и набор вспомогательных функций для работы с приложением, запись скрипта и его последующая модификация оказываются более долгим процессом по сравнению с обычным написанием кода.

2. **Пишите простой и понятный код.** Не усложняйте код скриптов редкоиспользуемыми и малопонятными конструкциями или сложной структурой классов. Помните, что даже если вы – единственный, кто сейчас занимается автоматизацией на вашем проекте, то после вас с кодом будет работать кто-то другой. Есть такой тип людей, которые любят экспериментировать со всем подряд,

углубляясь в решение задач, которых изначально даже не существовало, при этом забывая о цели собственно работы. Экспериментировать – это хорошо, но не на реальном проекте. Экспериментируйте дома в свободное время, а если из этих экспериментов получилось что-то полезное – внедряйте в проект.

3. **Используйте стандарты кодирования** и другие правила, присущие используемому языку программирования. Их можно взять готовые (если используется популярный язык типа Javascript или VBscript) или создать свои, взяв за основу какой-нибудь похожий язык программирования. Используйте статические средства проверки кода, которые позволяют находить в скриптах расхождение со стандартами. Старайтесь не выкладывать в систему контроля версий код, не прошедший такую проверку.

4. **Следуйте тем же правилам, что и программисты.** Написание скриптов – это тот же процесс программирования, а значит лучшим подходом при их написании будет следование тем же правилам, которым следуют разработчики кода. Обращайтесь к программистам за помощью, если вам что-то непонятно: чаще всего они с удовольствием делятся своими знаниями и правилам, которым сами следуют. Это не значит, что вы должны следовать *абсолютно всем* правилам, о которых вам расскажут, но знание этих подходов позволит вам быстрее выработать своё видение процесса разработки.

5. **Тестируйте свой код.** Программисты для тестирования своего кода используют unit тесты. В случае с автоматическими скриптами этот подход не подходит, так как вызовы функций и методов зачастую подразумевают взаимодействие с тестируемым приложением. Так как же можно протестировать свои скрипты на предмет их правильности и стабильности?

   Самый простой способ: поменять в них ожидаемые значения на неправильные и прогнать скрипт в этом режиме. В результате в логе вы должны получить понятные сообщения об ошибках. Например, предположим, что у нас есть следующий код, проверяющий, что в текстовом поле находится определенный текст:

```
var exp = "expected text";
var act = Sys.Process(...).Window(...).wText;
if(exp != act)
{
  Log.Error("Incorrect text", "Expected: " + exp + "\nActual: " + act);
}
```

   Для проверки того, что эта проверка работает правильно, достаточно поменять в первой строке ожидаемый текст (допустим, добавить в него лишний символ) и запустить тест на выполнение. В результате в логе должно появиться сообщение об ошибке с ожидаемым и полученным результатом.

   Другой способ проверки кода – это ручное взаимодействие с тестируемым приложением во время работы скрипта. Что будет, если какое-то окно не появится? Допустим, окно появляется после нажатия на кнопку. Поставьте в скрипте небольшую задержку после нажатия на кнопку (или поставьте в этом месте брекпоинт) и во время работы скрипта закройте это окно вручную, как только оно появится на экране. Как поведет себя в такой ситуации скрипт? Получим ли мы в логе понятную ошибку? Сможем ли воспроизвести ситуацию, не запуская скрипт заново, а просто читая лог?

6. **Автоматизируйте только устоявшуюся функциональность.** Нет смысла автоматизировать тестирование функциональности, которая еще не прошла проверку заказчиком, так как в нее могут вноситься изменения. Кроме того, прежде чем автоматизировать какую-то часть приложения,

необходимо протестировать ее вручную, чтобы убедиться, что в ней нет явных ошибок и что мы не будем проверять неправильное поведение приложения.

7. **Разрабатывайте независимые тесты.** Каждый тест должен быть самостоятельным, то есть не должен зависеть от других тестов, и его должно быть возможно запустить отдельно. Если какой-то тест зависит от результатов другого теста, то в случае, если в первом тесте произошла какая-то ошибка и сгенерировались неправильные данные (или не сгенерировались вообще), то не отработает и второй тест, который вполне может быть работоспособным, и вам придется решать две проблемы вместо одной. Чтобы добиться такой независимости тестов, необходимо чтобы каждый тест стартовал с определенной точки (например, принудительно закрывал тестируемое приложение, если оно открыто, а затем открывал его заново). Никогда не используйте в новом тесте уже открытое приложение в новом тесте, так как неизвестно, что с ним происходило во время работы предыдущего скрипта.

8. **Не дублируйте в скриптах код приложения.** Если, к примеру, в приложении производятся какие-то расчеты и вам необходимо проверить результат, не выполняйте эти же расчеты в скриптах.

    - Во-первых, вы рискуете внести ошибку в расчетах в своих тестах.
    - Во-вторых, в случае изменения формул расчета вам придется модифицировать скрипты.
    - В-третьих, в случае точных расчетов точность математических расчетов в языке, используемом для автоматизации, может отличаться от точности, используемой в языке, на котором написано приложение, что также может повлиять на результат.

    Для тестирования подобных ситуаций можно просто захардкодить в скриптах правильный вариант, а в случае нескольких вычислений – воспользоваться Data-driven подходом.

9. **Не подгоняйте скрипты под приложение.** Если в приложении есть ошибка и она отлавливается скриптом (например, неправильный текст ссылки) – не изменяйте скрипт только для того, чтобы он успешно отрабатывал. Во-первых, это приведет к тому, что ошибка так и останется неисправленной. Во-вторых, если она все же будет исправлена, скрипт снова придется менять. Лучше при каждом проходе скриптов видеть эту ошибку или же отключить временно этот скрипт, если другого решения нет.

10. **Запускайте скрипты как можно чаще.** Для обеспечения стабильности скриптов их нужно запускать как можно чаще. Делать это можно, например, на отдельном build-сервере при каждом внесении изменений в исходный код приложения (для этого настраивается система continuous integration), или просто на отдельном компьютере (или виртуальной машине). Если в течение дня вы циклично запускаете скрипты на одной и той же сборке приложения – внимательно исследуйте каждую ошибку, которая появляется в результате прогона скриптов, и вносите соответствующие улучшения в свой код.

4.7 Мифы и заблуждения относительно автоматизации

Здесь нам бы хотелось перечислить несколько известных заблуждений относительно автоматизации тестирования и развеять эти мифы. В целом необходимо отдавать себе отчет в том, что автоматизация тестирования – это такой же процесс разработки, как и разработка собственно программного обеспечения (возможно, немного проще), а потому ему присущи те же особенности и проблемы, которые возникают у разработчиков при написании программ.

1. **Автоматизация позволяет избавиться от ручного тестирования.** Это в корне неправильное предположение. Автоматизация лишь дополняет ручное тестирование и упрощает некоторые задачи (например, избавляет от рутины регрессионного тестирования и от человеческого фактора при проведении различных проверок). Новую функциональность вообще невозможно протестировать

иначе, как вручную, только после этого можно что-то автоматизировать. Даже если речь идет только об автоматизированном тестировании smoke тестов (т.е. самых простых и поверхностных), нам все равно придется иногда вносить изменения в скрипты в соответствии с изменениями в тестируемом приложении. В том случае, если у нас есть набор тестовых скриптов, которые никогда не меняются и проверяют только самую стабильную функциональность, и при этом запускаются автоматически, в обязанности тестировщиков все равно будет входить проверка того, что в отчетах нет ошибок. Именно поэтому многие специалисты утверждают, что говоря об автоматизации тестирования, мы имеем ввиду не «автоматическое» тестирование, а «компьютеризированное» (англ. computer-assisted).

2. **Внедрение автоматизации дает мгновенную выгоду.** Прежде, чем мы начнем получать выгоду от автоматизации, нам придется составить план тестирования (даже если у нас есть план для ручного тестирования, его, скорее всего, придется переделать), написать тесты (а также все, что с ними связано, т.е. код фреймворка и код для работы с приложением) и отладить их. В дальнейшем нам постоянно придется поддерживать существующие тесты и писать новые. Всё это требует времени и соответствующей квалификации тестировщика (а значит, на обучение тоже, скорее всего, потребуется время). Поэтому автоматизация тестирования поначалу не только не дает выгоды, а создает убытки. Только со временем, постепенно, мы начнем получать выгоду от регрессионного тестирования, когда тестировщики смогут больше времени уделять ручному тестированию новой функциональности, практически не тратя время на проверку старого.

   В нашей практике был лишь один случай, когда автоматизация тестирования начала приносить реальную и ощутимую пользу всего через 2 недели после начала внедрения. Это случилось благодаря тому, что использовался инструмент, с которым мы работали на нескольких других проектах до этого и у нас было готово много наработок, которые просто пришлось подогнать под нужды нового проекта (т.е. мы использовали уже готовый фреймворк). Кроме того, существующие ручные тесты в этом проекте были написаны таким образом, что их было легко автоматизировать.

   Еще одним фактором такого быстрого внедрения было то, что для автоматизации не пришлось отрывать никого из задействованных в ручном тестировании, таким образом автоматизаторы могли полностью сосредоточиться на своей работе.

3. **Автоматизировать можно любой ручной сценарий.** Это также неверно по нескольким причинам:

   5. любой инструмент для автоматизации тестирования имеет свои ограничения. Например, в тестируемом приложении могут использоваться элементы управления, разработанные сторонними производителями, поддержка которых не обеспечивается инструментом (чаще всего подобные проблемы возникают с различными таблицами). В таких случаях от некоторых проверок придется отказаться и оставить их для ручного тестирования.

   6. некоторые тестовые сценарии могут содержать задачи, которые проще выполнить вручную, чем пытаться автоматизировать (например, перезагрузка компьютера в середине теста или настройка удаленного сервера, включающая множество действий). Такие задачи можно либо оставить для ручной проверки, либо автоматизировать частично.

   7. в некоторых случаях проверки невозможно автоматизировать вообще, так как они не поддаются количественному описанию. Примерами таких задач могут быть градиентная заливка объекта с переходом цвета или же просто внешний вид окна или веб-страницы, когда нужно убедиться, что дизайн «не режет глаз» или что цвет элемента нормально сочетается с цветом фона.

4. **Автоматическими тестами можно покрыть 100% функциональности приложения.** Это невозможно, так как невозможно проверить все варианты входных и выходных параметров (это затруднительно даже для элементарных программ). Пожалуй, максимум, чего можно добиться в плане покрытия кода, – это того, что тестами будет покрыто 100% строк кода приложения, однако этот подход дает наименьшую точность покрытия (так как не проверяются различные условия и логические проверки).

5. **Универсальное средство автоматизации тестирования.** Ни один из ныне существующих инструментов автоматизации не является универсальным и вряд ли появления подобной программы можно ожидать в ближайшем будущем. Для того, чтобы инструмент был универсальным, ему нужно было бы работать во всех операционных системах (Windows, MacOS, Unix), поддерживать огромное количество сред разработки (Java, .NET, Qt, Flash и т.д.), уметь работать со всеми сторонними компонентами (DevExpress, Infragistic и еще тысячами других) и предоставлять универсальные способы работы в различных браузерах (их как минимум 5). Естественно, это невозможно, так как подобный инструмент стоил бы слишком дорого, чтобы его купил хоть кто-то. В нашей книге для примеров мы используем TestComplete, который поддерживает максимальное количество сред разработки и сторонних компонентов (по сравнению с другими подобными инструментами), однако у него есть один большой минус: он работает только под Windows.

Вас не должны пугать или смущать все перечисленные ограничения, их лишь надо иметь ввиду, чтобы понимать, чего можно добиться с помощью автоматизации, а чего нет.

4.8 Автоматизация и Scrum

В последние годы всё большую популярность приобретают модели быстрой разработки программного обеспечения, в частности Scrum. Особенность этого подхода в том, что через определенные равные промежутки времени (так называемые «спринты», обычно они длятся от одной до четырех недель) мы должны иметь вполне работоспособный продукт.

Основная проблема с точки зрения автоматизации при таком подходе заключается в том, что в конце спринта обычно хватает времени лишь на то, чтобы протестировать новую функциональность вручную, но не на то, чтобы написать и отладить для неё автоматические тесты. Есть два способа, позволяющих решить эту проблему:

7. **Переносить разработку автотестов на следующий спринт.** У этого подхода есть два основных достоинства: во-первых, новая функциональность будет протестирована без спешки вручную, что позволит найти больше ошибок и вовремя их исправить; во-вторых, в начале следующего скрипта у вас все равно будет время, пока программисты напишут что-то, готовое для тестирования, и это время можно потратить на написание скриптов, тестирующих функциональность из прошлого спринта. Недостаток этого подхода в том, что фактически цель спринта с точки зрения тестирования не будет полностью достигнута.

8. **Разрабатывать тестовые скрипты одновременно с разработкой приложения.** Так как приложение разрабатывается по спецификациям, мы вполне можем написать скрипты, исходя из информации о том, как должно работать приложение, а затем, когда новая функциональность будет доступна, лишь внести небольшие изменения в тесты в тех местах, где были допущены неточности.

   Этот подход называется Behaviour-driven development, он получил распространение по мере внедрение Agile-подобных методологий и в целом согласуется с подходом TDD (Test-driven development, разработка через тестирование). Несмотря на то, что в целом этот подход более соответствует модели быстрой разработки, он является более сложным, так как требует дополнительных усилий как со стороны тестировщиков, так и со стороны разработчиков (строгое

следование стандартам именования элементов управления, разработка «заглушек» для данных – mock’ов и т.п.).

Кроме того, этот подход в целом противоречит одному из основных правил автоматизации: автоматизируйте только ту функциональность, которая протестирована как минимум один раз и одобрена заказчиком.

5. Пример создания проекта

В этом разделе мы пройдём шаг за шагом весь путь автоматизации: от постановки задачи до регулярного запуска тестов.

В качестве тестируемого приложения мы возьмем стандартный Блокнот (приложение, идущее в поставке с Windows). В качестве объекта автоматизации мы выбрали Smoke-тест (т.е. поверхностный тест, задача которого – поверхностно проверить базовые функции приложения).

В нашем проекте мы воспользуемся одновременно несколькими подходами из главы 2, которые нам упростят задачу.

5.1 Постановка задачи

Итак, нам необходимо автоматизировать Smoke-тест для Блокнота. Этот тест должен запускаться после каждой сборки приложения (а значит он должен выполняться быстро, чтобы не сильно затягивать процесс сборки) и быть первым в списке всех тестов, которые будут у нас написаны и запущены в дальнейшем.

Какие именно проверки включать в состав Smoke-теста – зависит от конкретного приложения и его назначения, поэтому здесь нельзя дать чётких рекомендаций. Обычно проверяются наиболее критичные области программы, без которых тестирование практически невозможно и требуется немедленное исправление.

В нашем случае мы решили, что достаточной проверкой в Smoke-тесте будет проверка того, что все диалоговые окна, доступные из главного меню, открываются и закрываются.

5.2 Создание тесткейсов

Теперь пришло время создать тесткейсы, по которым мы будем в дальнейшем писать скрипты. Так как у нас всего один тест, то и тесткейс будет один.

Признаком того, что меню открывает диалоговое окно, обычно является наличие символа «...» (многоточие) в конце имени пункта меню (например, «Replace...»). Мы воспользуемся этим фактом для того, чтобы дополнительно проверять, не появились ли в нашем приложении новые диалоговые окна (считается, что мы тестируем приложение, которое в данный момент находится в разработке, а значит в него в любой момент может быть добавлена новая функциональность, которую также необходимо тестировать). В случае, если найдены новые пункты меню, тестирование которых не реализовано скриптом, мы будем выводить в лог предупреждение.

Теперь напишем шаги тесткейса.

Блокнот. Smoke-тест		
Шаг №	**Действие**	**Проверка**
1	Запустите Блокнот из командной строки	Блокнот запущен. Главное окно активно. Никаких дополнительных диалоговых окон не открылось
2	Проверьте, что все элементы меню, заканчивающиеся на «...», активны	Все пункты меню активны
3	Выберите первый доступный элемент меню из предыдущего	Открывается диалоговое окно с заголовком, совпадающим с текстом

	шага любым способом (клавиатура, мышь, горячая клавиша)	меню
4	Закройте окно, открытое в предыдущем шаге, нажатием на кнопку закрытия [x]	Диалоговое окно закрыто
5	Повторите шаги 3-4 для всех пунктов меню из шага 2	Тот же результат
6	Закройте Блокнот, выбрав пункт меню File|Exit	Блокнот закрыт

Как видите, тест довольно простой и не содержит специфичной информации (например, у нас нет точно определенного списка пунктов меню, которые нужно проверять), поэтому мы постараемся сделать наш автоматический тест с одной стороны гибким и расширяемым, а с другой – не пропустить необходимых проверок.

Здесь сразу необходимо оговориться, что зачастую подобные тесты, рассчитанные на ручную проверку, нет смысла автоматизировать буквально. Например, шаг 2 (проверка активности элементов меню) может быть вполне пропущен, так как эта проверка будет выполнена неявно в следующем шаге (выбор пункта меню). Если пункт меню недоступен, у нас возникнет ошибка при попытке обратиться к нему.

5.3 Создание проекта и выбор подходов

Теперь определимся с тем, какие подходы мы будем использовать в наших тестах:

- Мы, безусловно, воспользуемся функциональной декомпозицией (глава 2.2), так как ее необходимо использовать всегда (даже в таком простом случае, как наш пример). Для более удобной навигации по проекту мы будем размещать модули в соответсвующих папках проекта.

- Мы воспользуемся object-driven подходом (глава 2.5) для написания кода работы с Блокнотом. Это также позволит нам не использовать стандартную возможность NameMapping, предоставляемую нам TestComplete'ом[5].

- Мы будем использовать data-driven подход (глава 2.3) для хранения данных (пункты меню и соответствующие им окна, горячие клавиши и т.п.).

- Для проверки открытия окон мы используем некоторые подходы синхронизации (глава 2.7.)

- И, наконец, мы воспользуемся нелинейным подходом для тестирования (будем обращаться к пунктам меню случайным образом: мышь, клавиатура, горячие клавиши).

Теперь мы можем создать проект (обратите внимание, что это первоначальный его вариант, в дальнейшем структура может поменяться в зависимости от наших потребностей).

[5] Это вовсе не значит, что NameMapping плох. Просто в этом учебнике мы стараемся избегать специальных возможностей инструментов, чтобы не пришлось сильно отвлекаться на их объяснение.

5.4 Создание структуры кода и общих функций

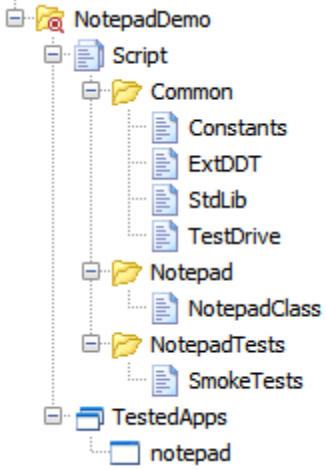


Рис. 6.1. Общий вид проекта

Эта структура не была создана сразу, она наращивалась постепенно по мере создания общего кода для работы с приложением Блокнот.

В разделе Common у нас находится код общего назначения (константы, дополнительные функции для упрощения работы с DDT, функции, относящиеся непосредственно к фреймворку и функции для расширения стандартной библиотеки JScript).

В папке Notepad мы разместим весь код, относящийся непосредственно к работе с Блокнотом. На данный момент здесь будет только один класс Notepad, однако в более сложных проектах количество файлов может быть гораздо больше.

И, наконец, в папку NotepadTests мы будем помещать непосредственно тесты.

Давайте рассмотрим каждый из файлов более подробно.

Constants

```
WAIT = 5000; // Wait timeout for windows
```

На данный момент у нас определена всего одна константа: время ожидания для окон и других элементов управления. Везде, где необходимо ожидать появления окна, мы будем в качестве таймаута использовать именно эту константу, а не числовые значения. Тогда в случае, если нам понадобится увеличить везде время ожидания (например, при запуске тестов на более медленном компьютере), нам придется внести изменение всего в одном месте.

ExtDDT

```
function getDDTData(fileName, sheet)
{
  var data = {};
  var useACEdriver = (aqFileSystem.GetFileExtension(fileName) == "xlsx");

  var excel = DDT.ExcelDriver(fileName, sheet, useACEdriver);
  try
  {
    var i;
    for(i = 0; i < excel.ColumnCount; i++)
```

```
    {
      data[excel.ColumnName(i)] = [];
    }

    while(!excel.EOF())
    {
      for(i = 0; i < excel.ColumnCount; i++)
      {
        data[excel.ColumnName(i)].push(excel.Value(excel.ColumnName(i)));
      }
      excel.Next();
    }
  }
  catch(e)
  {
    Log.Error("An error occured while reading data from excel file",
        fileName + "\n" + sheet + "\n" + e.number + "\n" + e.description)
  }
  finally
  {
    DDT.CloseDriver(excel.Name);
  }

  return data;
}
```

В файле ExtDDT у нас определена всего одна функция, которая считывает все данные из заданной странички Excel файла и возвращает эти данные в виде объекта, каждое поле которого является массивом данных определенного столбца. На данном этапе нам не нужны другие функции, однако если развивать этот проект, очень скоро окажется, что нам нужны функции для получения данных из определенной строки, столбца или интервала, поэтому имеет смысл подобные функции сразу помещать в отдельный файл.

StdLib

```
function arrayFind(arr, element)
{
  for (var i = 0; i < arr.length; i++)
  {
    if (element === arr[i])
    {
      return i;
    }
  }
  return -1;
}

function randInt(min, max)
{
  return Math.round(Math.random()*(max-min)+min)
}
```

В файле StdLib мы сохранили 2 функции, упрощающие нам жизнь при написании тестов и общего кода: arrayFind – для поиска элемента в массиве и randInt – для генерации случайного числа в заданном интервале.

TestDrive

```
var STEP = 1;
var STEP_ID;

function StartStep(description)
{
    STEP_ID = Log.CreateFolder("STEP #" + STEP + ": '" + description + "'");
    Log.PushLogFolder(STEP_ID);
}

function EndStep(expected)
{
    Log.PopLogFolder();
    Log.Message("   Expected: '" + expected + "'");
    STEP++;
}
```

Файл TestDrive предназначен для хранения функций, упрощающих структуризацию тестов и улучшающих вывод информации в лог. Функции StartStep и EndStep будут вызываться соответственно в начале и в конце каждого шага тесткейса. В результате при просмотре кода теста мы легко сможем перейти к нужному нам шагу (по его описанию), а в логе вся дополнительная информация будет скрыта в виде дочерних элементов дерева.

Примеры использования всех этих функций мы увидим ниже при написании класса Notepad и smoke-теста.

Обратите внимание, что каждый из этих модулей является независимым, так как находится на самом «нижнем» уровне иерархии кода и может быть без проблем перенесён в новый проект в случае надобности.

NotepadClass

В нашем примере это самый большой файл. Здесь хранятся все методы для работы с Блокнотом, которые мы будем вызывать из тестов.

```
//USEUNIT Constants
//USEUNIT StdLib

function Notepad()
{
  // Properties
    // define custom properties here

  // Windows and Objects
    this.Process = function()
    {
      return Sys.Process("notepad");
    }

    this.MainWin = function()
```

```
  {
    return this.Process().Window("Notepad", "* - Notepad");
  }

  this.Dialog = function(wndCaption, wndClass)
  {
    if(wndClass == undefined)
    {
      wndClass = "#32770";
    }
    return this.Process().WaitWindow(wndClass, wndCaption, -1, WAIT);
  }

// Methods
  this.run = function()
  {
    while(Sys.WaitProcess("notepad", 100).Exists)
    {
      Log.Message("Terminating running Notepad");
      Sys.Process("notepad").Terminate();
    }
    TestedApps.notepad.Run();
    if(!Sys.WaitProcess("notepad", WAIT).Exists)
    {
      Log.Error("Notepad process didn't start");
      return false;
    }
    return true;
  }

  this.close = function()
  {
    this.Process().Terminate();
  }

  this.getMenuItems = function()
  {
    var menus = [];
    var i, j;
    for(i = 0; i < this.MainWin().MainMenu.Count; i++)
    {
      var currentMenu = this.MainWin().MainMenu.Items(i);
      for(j = 0; j < currentMenu.SubMenu.Count; j++)
      {
        menus.push(currentMenu.Caption + "|" + currentMenu.SubMenu.Items(j).Caption);
      }
    }

    return menus;
```

```
    }

    this.selectMenuItem = function(menu)
    {
      var method = randInt(1, 3);
      Log.Message("Selecting menu item " + menu["MenuItem"]);
      Log.Message("Random method generated: " + method);

      switch(method)
      {
        case 1: // select by mouse
          this.MainWin().MainMenu.Click(menu["MenuItem"]);
          break;
        case 2: // select by accelerator
          this.MainWin().Keys(menu["Accelerator"])
          break;
        case 3: // select by hotkey
          if(menu["Hotkey"] != "")
          {
            this.MainWin().Keys(menu["Hotkey"]);
          }
          else
          {
            Log.Message("Hotkey isn't available, using accelerator");
            this.MainWin().Keys(menu["Accelerator"]);
          }
          break;
        default:
          Log.Error("Unknown method for selecting menu item: " + method);
          break;
      }
    }
}
```

Обратите внимание, что каждый метод, выполняющий действие (например, close, run и т.д.) начинается с глагола, а все методы, возвращающие объекты (Dialog, MainWin), этому правилу не следуют. Это сделано для того, чтобы всегда придерживаться определенных стандартов и точно знать, что делает тот или иной метод. Мы также придерживаемся правила начинать имена методов с маленькой буквы, если этот метод выполняет действие, и с большой – если он возвращает объект. Это также улучшает и упрощает чтение кода.

5.5 Написание автоматических тесткейсов

Теперь перейдем к написанию автотестов. В нашем случае это всего один Smoke-тест.

SmokeTests

```
//USEUNIT NotepadClass
//USEUNIT ExtDDT
//USEUNIT TestDrive

function TestSmokeDialogs()
{
```

```
StartStep("Start Notepad application")
  var np = new Notepad();
  np.run();
EndStep("Notepad is started");

StartStep("Make sure that Smoke test covers all existing dialog boxes")
  var expMenus = getDDTData(Project.Path + "\\Data\\NotepadData.xlsx", "SmokeTestData");
  var actMenus = np.getMenuItems();

  for(var i = 0; i < actMenus.length; i++)
  {
    if(actMenus[i].substr(actMenus[i].length-3, 3) == '...' &&
        arrayFind(expMenus["MenuItem"], actMenus[i]) == -1)
    {
      Log.Warning("Menu item '" + actMenus[i] + "' not tested");
    }
  }
EndStep("All dialog boxes are being tested");

StartStep("Verify all available dialogs can be opened and closed");
  for(var i = 0; i < expMenus["MenuItem"].length; i++)
  {
    var menu = {
                MenuItem: expMenus["MenuItem"][i],
                Accelerator: expMenus["Accelerator"][i],
                Hotkey: expMenus["Hotkey"][i]
                };

    if(expMenus["WindowCaption"][i] == "Find")
    {
      np.MainWin().Keys("Some text to enable menu Find");
    }
    np.selectMenuItem(menu);
    var dialog = np.Dialog(expMenus["WindowCaption"][i]);
    if(dialog.Exists)
    {
      dialog.Close();
    }
    else
    {
      Log.Error("Window '" + expMenus["WindowCaption"][i] + "' didn't open");
      np.close();
      np.run();
    }
  }
EndStep("All dialogs can be opened and closed");

StartStep("Close Notepad application")
  np.MainWin().Activate();
```

```
    np.close();
  EndStep("Notepad is closed");
}
```

Обычно тест считается идеальным в том случае, если он содержит только вызовы методов и функций. Однако на практике зачастую приходится отходить от этого правила (ведь ничто не может быть идеальным) и вставлять в тесты условия, циклы и т.п.

В нашем конкретном случае это тем более необходимо, так как в нашем тесте заложена возможность проверки, не появились ли в приложении новые пункты меню, вызывающие диалоговые окна, но которые не проверяются нашим скриптом. В случае, если такой пункт меню будет найден, мы получим в логе предупреждение с соответствующим текстом и примем меры для оповещения команды об этом (это будет описано дальше).

Хотя написание тестов мы и выделили в отдельный раздел, все 3 части нашего проекта (Common, Notepad и NotepadTests) писались одновременно, постепенно дополняя и расширяя друга.

Обратите внимание, что автоматический тест несколько отличается от ручного сценария, приведенного ранее:

1. В ручном сценарии ничего не сказано о проверке на то, что в новой версии приложения появились новые пункты меню. В автоматическом сценарии эта проверка выделена в одтельный шаг. Это происходит потому, что при ручной проверке человек будет просматривать все пункты меню, заканчивающиеся на троеточие и проверять заголовки окон, исходя из требований или здравого смысла. В автоматическом сценарии нам было бы труднее описать подобное поведение скрипта (хотя и возможно), поэтому мы выбрали вариант «золотой середины»: не усложнять код, но при этом информировать команду в том случае, если необходимо изменить скрипт в соответствии с изменениями в приложении.

2. Шаг с проверкой активности элементов меню не сделан вообще. Эта проверка была бы излишней, так как она неявно осуществляется при выборе пункта меню, и если один из них будет недоступен, мы узнаем об этом по ошибке в логе.

3. Шаги 3, 4 и 5 объединены в один логический блок. Было бы неразумно описать отдельно шаги 3 и 4, а затем использовать цикл для шага 5, при этом исключив из списка первый (уже проверенный ранее) пункт меню.

Все эти изменения вполне нормальны и могут иметь место в том случае, когда мы занимаемся автоматизацией тестов, которые одновременно используются для ручного тестирования. Именно поэтому не стоит начинать писать скрипты, не прочитав предварительно тесткейс несколько раз и не представив себе в голове, как будет выглядеть код.

5.6 Настройка регулярных запусков тестов

Теперь, когда код написан и отлажен, мы можем несколько раз запустить наш тест, чтобы убедиться, что он работает более-менее стабильно. Вот как будет выглядеть лог нашего скрипта:

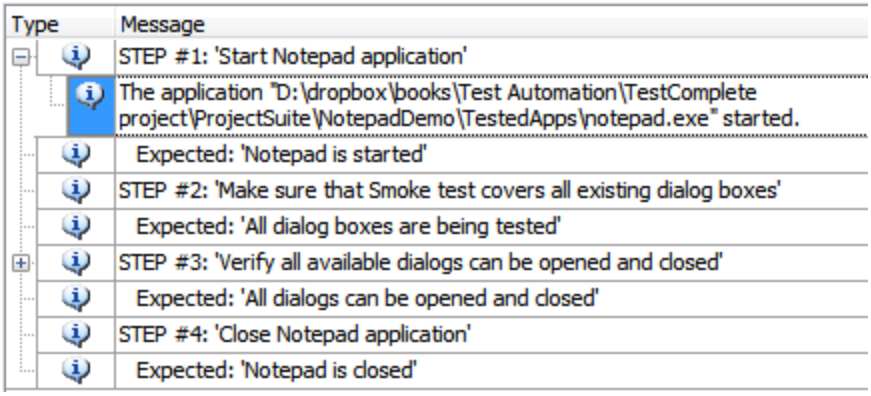


Рис. 6.2. Лог запуска теста

Обратите внимание, что вся важная информация (действия и ожадиемые результаты) у нас вынесены на первый план, а вся дополнительная информация по умолчанию скрыта, однако может быть показана в случае надобности. Например, вот как будет выглядеть лог в случае, если одно из диалоговых окон не откроется по истечении 5 секунд после выбора пункта меню:

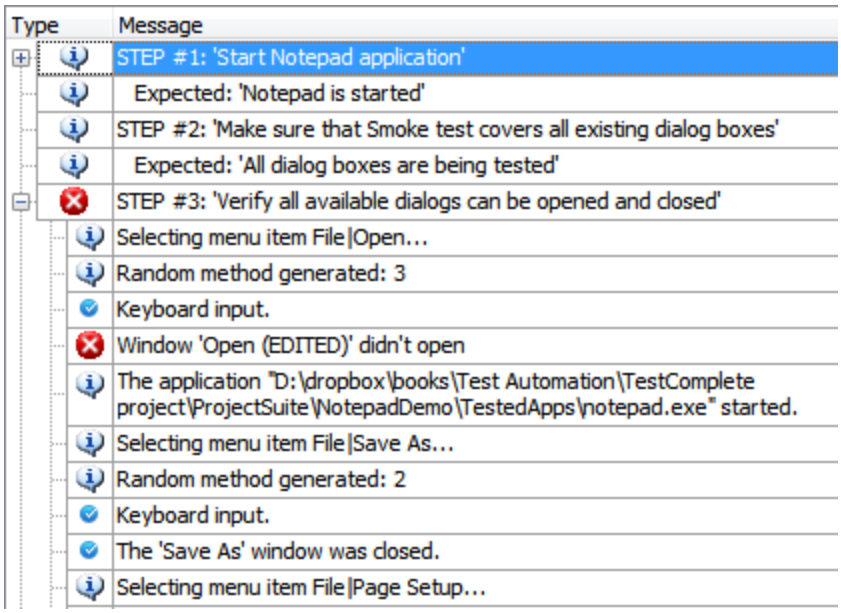


Рис. 6.3. Лог с ошибкой

Мы видим, в каком именно шаге и после каких именно действий возникла ошибка, а значит можем выполнить все те же действия, чтобы воспроизвести проблему вручную.

Теперь мы можем либо интегрировать запуск теста с системой непрерывной интеграции (Continouos Integration), либо сделать так, чтобы тест запускался с определенной периодичностью. Во втором случае нам придется также позаботиться о том, чтобы на компьютере, на котором запускаются тесты, всегда была установлена наиболее свежая версия приложения (это можно сделать, например, написав еще один скрипт).

Независимо от того, как и где мы будем запускать наши тесты, нам понадобится какой-то простой способ это делать. Проще всего воспользоваться возможностями batch-файлов. Ниже представлен листинг файла RunSmokeTest.bat, который будет использоваться для запуска нашего Smoke-теста.

```
@echo off
taskkill /fi "STATUS eq RUNNING" /f /t /im testcomplete.exe

set TC_PATH=C:\Program Files\Automated QA\TestComplete 7\Bin\TestComplete.exe
```

```
"%TC_PATH%"  .\ProjectSuite.pjs /e /r /p:NotepadDemo /u:SmokeTests /rt:TestSmokeDialogs

if %ERRORLEVEL% == 0 goto end
if %ERRORLEVEL% == 1 goto warning
if %ERRORLEVEL% == 2 goto error

:error
echo Critical error(s) were found during script run
exit /b 2

:warning
echo Warning(s) were found during script run, sending e-mail
call .\send_email.bat
goto end

:end
echo Tests ran successfully
```

Прежде всего мы закрываем процесс TestComplete'а (если он вдруг открыт), затем запускаем тест (передав его в качестве параметра в TestComplete), а в конце проверяем возвращаемое TestComplete'ом значение и в зависимости от его значения выполняем необходимые действия. Если TestComplete вернул 0, значит всё в порядке, ошибок не было. Если он вернул 1, значит в тесте были предупреждения. Этого недостаточно, чтобы считать билд нерабочим, поэтому мы отсылаем письмо (запустив файл send_email.bat) и затем выходим из пакетного файла с кодом 0 (чтобы билд считался успешным). Если же TestComplete вернул нам 2, значит при выполнении теста возникли ошибки, поэтому из пакетного файла мы выходим с кодом 2, чтобы сообщить, что что-то не в порядке (по умолчанию код выхода 0 считается успешным окончанием, любой другой – ошибкой).

Обратите внимание, что в этом примере в случае возникновения предупреждения, вызывается файл send_email.bat, который в нашем примере ничего не делает, а просто выводит сообщение в консоль. Написание этого файла выходит за рамки нашего учебника и мы оставляем эту задачу для читателя.

В конце мы можем также отправлять логи на определенные e-mail адреса в случае надобности, но здесь мы не стали этого делать, чтобы не усложнять наш пример.

Теперь этот файл мы можем добавить в качестве задачи в нашу систему непрерывной интеграции и по результату его выполнения считать сборку успешной или нет.

5.7 Заключение

В этом небольшом примере мы прошли весь путь, начиная с создания ручного сценария и заканчивая настройкой регулярных запусков.

Как видите, даже такой простой пример требует хорошо продуманных действий. Именно поэтому прежде, чем начинать реальный проект, рекомендуется сделать пилотный проект, который через месяц-два будет не жалко удалить и начать всё заново, с чистого листа.

На этом мы завершаем раздел об автоматизации функционального и регрессионного тестирования. Мы постарались охватить практически все имеющиеся подходы и возможности, которые можно использовать в автоматизации, хотя, конечно, рассмотреть удалось не всё, а любые подходы можно улучшать бесконечно.

Лучшим местом для поиска информации и получения ответов, безусловно, является интернет. Там можно найти ответы практически на все вопросы, которые у вас могут возникнуть. Лучше всего искать информацию на англоязычных сайтах, хотя и в русском сегменте интернета по теме автоматизации есть множество статей.

Желаем удачи в изучении автоматизации и постановке процессов в ваших проектах!

6. Инструменты автоматизации

В этом разделе мы поговорим о различных инструментах, предназначенных для автоматизации тестирования, их достоинствах и недостатках, а также сделаем небольшой обзор существующих приложений.

6.1 Выбор: платное или бесплатное?

Один из первых вопросов, с которым мы сталкиваемся на этапе выбора инструмента, является цена, которую компания готова заплатить за приложение, с помощью которого мы будем автоматизировать наши тесты. Цена же может колебаться от нуля (бесплатные инструменты и проекты с открытым исходным кодом) до десятков и даже сотен тысяч долларов.

Наша задача – вовсе не «выбить» из руководства как можно больше денег для своих нужд, а найти «золотую середину» – такой инструмент, который будет устраивать и нас, и компанию, в которой мы работаем.

Ниже перечислены основные достоинства и недостатки платных и бесплатных инструментов.

Бесплатные (или недорогие) инструменты:

1. Стоимость. Это очевидный плюс данной группы инструментов.
2. Расширяемость. Обычно бесплатные инструменты предоставляются с открытым исходным кодом, что позволяет нам расширять и улучшать их под свои нужды.
3. Поддержка новых технологий. Если инструмент поставляется с исходными кодами, вполне возможно, что для него сторонними пользователями будет написано множество расширений, в том числе и для новых технологий (в коммерческих инструментах добавление поддержки новых технологий обычно происходит дольше, так как больше времени уделяется тестированию и совместимости с существующими модулями).
4. Отсутствие техподдержки. Это очень важный недостаток бесплатных инструментов, так как единственным источником информации в случае затруднений использования оказываются интернет-форумы, где можно пообщаться с такими же пользователями.
5. Узконаправленность инструмента. Зачастую бесплатные и недорогие инструменты предназначены для тестирования какого-то одного типа приложений (например, для тестирования веб-приложений или приложений .NET). Соответственно если в дальнейшем нам понадобится тестировать какой-то другой тип приложения, придется снова заниматься поиском подходящего инструмента.
6. Уровень знаний и опыта, требующийся от тестировщика, использующего бесплатный инструмент, обычно выше, чем при использовании коммерческих инструментов, поскольку установка, настройка, изучение и интеграция подобных приложений обычно сложнее, чем использование платных приложений.

Платные (дорогие) инструменты:

1. Официальная техподдержка. Это очень большой плюс, особенно на первых порах, когда вы только изучаете инструмент. В любой момент вы можете получить квалифицированную помощь службы поддержки, сотрудники которой, вполне вероятно, уже сталкивались с подобными вопросами и могут быстро помочь решить проблему.
2. Поддержка большого количества технологий. В отличие от узконаправленных бесплатных инструментов, коммерческие приложения обычно позволяют работать с различными типами

приложений (веб, десктопные, мобильные), используя одни и те же подходы для всех тестируемых приложений.

3. Интеграция с другими продуктами. Обычно коммерческие инструменты предоставляют широкие возможности интеграции с другими приложениями (например, багтрекерами, системами контроля версий, системами управления проектами и т.д.), что может существенно упростить создание инфраструктуры автоматизации.

4. Качество коммерческих инструментов обычно выше, чем качество бесплатных утилит. Производители платных инструментов автоматизации обычно заботятся о качестве выпускаемых ими продуктов, их хорошо тестируют и проверяют на совместимость с различными версиями операционных систем, браузеров и т.п., что уменьшает вероятность нахождения ошибок конечными пользователями.

5. Стоимость коммерческих инструментов может оказаться существенным недостатком при выборе программы для автоматизации.

6. Внедрение поддержки новых технологий или сторонних библиотек может происходить спустя некоторое время после их выхода на рынок, что, безусловно, является существенным минусом, если в вашем тестируемом приложении всегда используются их свежие версии.

7. Невозможность самостоятельного расширения возможностей инструмента. Не все коммерческие инструменты предоставляют средства для расширения их возможностей (API или SDK). Наличие же подобной возможности может решить проблему позднего внедрения поддержки сторонних компонентов, так как вы сможете самостоятельно писать необходимые дополнения.

Если вас не устраивает цена на приложение, имеет смысл пообщаться с отделом продаж компании-разработчика (sales team), так как некоторые инструменты не продаются целиком, а разделяются на модули. Вы покупаете «базовый комплект» приложения и лишь необходимые дополнения к нему, что позволяет сэкономить деньги.

6.2 Выбор: универсальное или специализированное?

С точки зрения поддержки различных технологий и программных платформ, инструменты автоматизации делятся на 2 категории: универсальные (т.е. поддерживающие сразу несколько технологий) и специализированные (поддерживающие лишь какую-то одну из них). Как уже было сказано в предыдущей главе, коммерческие инструменты обычно поддерживают несколько платформ, а бесплатные – только одну.

Недостаток специализированных программ очевиден: такой программой мы сможем тестировать только один тип приложений, при этом не имея никакой возможности взаимодействовать с другими типами приложений. Например, если выбранный нами инструмент работает только с приложениями .NET, а нужно протестировать взаимодействие тестируемого приложения с какой-то простой Win32 утилитой (например, со стандартным Блокнотом), то сделать это мы не сможем.

С другой стороны, коммерческие приложения могут иметь свои ограничения, так как производители стараются сделать свои инструменты максимально совместимыми со всеми поддерживаемыми технологиями, при этом стараясь уменьшить различия в работе с разными типами приложений для конечного пользователя (т.е. для тестировщика). В результате такие подходы могут оказаться менее стабильными или менее производительными по сравнению со специализированными программами.

Если перед вами стоит выбор, какой из этих двух типов инструментов выбрать, необходимо не только учитывать текущие нужды компании или отдела, в котором внедряется автоматизация, но также постараться предугадать нужды компании в будущем.

Если вы работаете в небольшой компании, которая занимается исключительно разработкой веб-сайтов или какого-то одного приложения, вряд ли вам понадобится тестировать другие типы приложений, поэтому выбор стоит остановить на специализированном инструменте.

Если вы работаете в средней или большой компании, которая только начинает внедрять автоматизацию и в которой существует несколько различных подразделений, создающих различные приложения на разных платформах, то скорее всего лучше остановить выбор на универсальном инструменте. Использование одного инструмента для всех (или большинства) разрабатываемых компанией продуктов позволит тестировщикам повторно использовать код из одного проекта в других, а также упростит процесс обучения в следующих проектах (так как в первом проекте, где автоматизация внедрялась раньше всего, уже будут опытные специалисты, знакомые с инструментом автоматизации, которые смогут помочь следующим проектам).

6.3 Обзор инструментов для функционального тестирования

В этой главе мы кратко рассмотрим наиболее часто встречающиеся и используемые инструменты для автоматизации тестирования. Не стоит рассматривать эти описания как детальное описание и истину в последней инстанции, так как мы дадим лишь поверхностное описание, которое, возможно, позволит вам сэкономить время при начальном поиске подходящего вам инструмента.

Обратите внимание на следующие особенности обзора:

1. Под «встроенными возможностями» подразумеваются различные подходы и техники автоматизации тестирования, которые явно поддерживаются инструментом (например, в виде модулей или библиотек). В нашем обзоре мы указываем только основные такие возможности инструментов, а потому эти списки нельзя считать полными. Для получения полной информации об инструменте и его возможностях лучше всего воспользоваться встроенной в него справочной системой.

2. То же самое касается типов тестируемых приложений: здесь указаны только наиболее известные и часто встречающиеся технологии.

3. Цена на инструменты автоматизации может сильно различаться в зависимости от типа лицензии. Чаще всего встречаются 2 типа лицензии:

    - **Именная или закрепленная (named, node-locked).** В этом случае лицензия может использоваться на одном компьютере или одним человеком. Если необходимо установить программу на другом компьютере, на первом ее необходимо удалить или приобрести вторую такую же лицензию. Это самый дешевый тип лицензий, он обычно применяется в том случае, если в команде есть люди, полностью занятые автоматизацией и которым необходим постоянный доступ к программе.

    - **Плавающая лицензия (floating).** В этом случае программа может быть установлена на скольких угодно компьютерах, однако количество одновременно запущенных программ на разных компьютерах (или в разных сессиях одного компьютера) не может превышать количества имеющихся лицензий. Этот тип полезен в том случае, если автотесты пишутся несколькими людьми, однако они не заняты автоматизацией на 100%, поэтому могут работать с программой в разное время. Еще один пример использования плавающих лицензий – использование продукта в разных часовых поясах, в случае когда рабочее время двух команд не пересекается. В этом случае сотрудники с одной и той же лицензией работают сначала сотрудники одного офиса, а когда их рабочий день заканчивается лицензии используются сотрудниками другого офиса.

Обычно чем больше лицензий приобретается, тем ниже стоимость каждой из них. Все эти вопросы необходимо уточнять с сотрудниками отдела продаж компании.

В нашем обзоре в качестве нижней ценовой планки мы берем стоимость одной именной лицензии, а в качестве верхней планки – стоимость одной плавающей лицензии. Однако ценовая политика некоторых инструментов отличается от приведенной выше схемы, поэтому цены здесь приведены весьма приблизительные.

4. В большинстве случаев цена указана только за саму программу и не включает стоимость поддержки (т.н. подписка – subscription).

5. Также стоит отметить, что приведенная далее информация является актуальной на момент написания книги (2013 год), однако со временем картина на рынке может существенно поменяться, поэтому обязательно проверяйте всю приведенную здесь информацию на актуальность.

TestComplete

Разработчик: SmartBear Software

Сайт разработчика: http://smartbear.com/

Цена за 1 лицензию: $2000 - $4000

Языки программирования: JScript, VBScript, DelphiScript

Встроенные возможности: Record & Play, DDT, ODT, KDT, User Forms, Distributed testing, Web Services и др.

Поддержка тестирования приложений: Win32 (MS Visual C++, Intel C++), Java, .NET, Delphi/C++ Builder, Qt, PowerBuilder, Visual FoxPro, Web, Flash, Flex, Silverlight.

TestComplete – один из наиболее привлекательных коммерческих инструментов по соотношению цена/качество. При небольшой (по сравнению с другими коммерческими инструментами) цене он поддерживает практически все основные среды разработки, причем поддержка новых версий библиотек и платформ добавляется в TestComplete весьма оперативно.

Кроме поддержки большого количества технологий разработки и сторонних элементов управления, возможности TestComplete могут быть расширены за счет создания собственных расширений (на языках C++ или Delphi).

Кроме того стоит отметить официальную службу поддержки, которая помогает не только зарегистрированным пользователям (как это принято у других производителей), а всем. При этом вопросы от клиентов имеют приоритет выше, чем остальные пользователи, поэтому на их вопросы служба поддержки обычно отвечает быстрее.

SilkTest

Разработчик: Micro Focus

Сайт разработчика: http://www.microfocus.com/

Цена за 1 лицензию: $4.500 – $9.000

Языки программирования: 4Test, Java, VB/C#

Встроенные возможности: Record & Play, DDT, Distributed testing и др.

Поддержка тестирования приложений: Win32, Java, .NET, Web.

Довольно мощный инструмент, поддерживающий основные типы тестируемых приложений и рассчитанный на использование в средних и крупных проектах. Встроенный язык программирования 4Test является, пожалуй, одним из наиболее продвинутых языков программирования, встречающихся в инструментах автоматизации. Также присутствует возможность написания тестов на языках Java, Visual Basic и C# в средах разработки Eclipse и Visual Studio.

QuickTest Professional (QTP)

Разработчик: Hewlett-Packard

Сайт разработчика: http://hp.com/

Цена за 1 лицензию: $6.000 и выше (доплата отдельно за каждый модуль)

Языки программирования: VBScript

Встроенные возможности: Record & Play, KDT, DDT, Distributed testing и др.

Поддержка тестирования приложений: Win32, Java, .NET, Web.

QuickTest Professional – самый дорогой инструмент из существующих в данный момент на рынке. При этом он поддерживает наибольшее количество различных тестируемых приложений, однако за каждый модуль придется доплатить отдельно.

Компания Mercury Interactive (впоследствии приобретенная компанией Hewlett Packard) была первой на рынке инструментов автоматизации тестирования GUI и продолжает удерживать эту позицию до сих пор, несмотря на то, что в последние годы появилось большое количество более дешевых решений.

QuickTest Pro хорошо интегрируется с другими инструментами этого же производителя, поэтому обычно используется в крупных корпорациях, которые могут себе позволить покупку столь дорогого инструмента.

Squish

Разработчик: froglogic GmbH

Сайт разработчика: http://www.froglogic.com/

Цена за 1 лицензию: от $4.500 (за одну лицензию на один тип тестируемого приложения, т.н. Edition)

Языки программирования: JavaScript, Python, Perl, Ruby, Tcl

ОС: Windows, Linux/Unix, MacOS X, iOS

Встроенные возможности: Record & Play, DDT, Distributed testing и др.

Поддержка тестирования приложений: Win32, Java, .NET, Qt, Web.

Squish – это, пожалуй, единственный на данный момент кроссплатформенный коммерческий инструмент с поддержкой наиболее часто используемых технологий. Кроме кроссплатформенности большим плюсом инструмента является список языков программирования, которые можно использовать для написания тестов. Например, языки Python и Ruby очень мощные и содержат практически всё, что может понадобиться при написании тестов и создании хорошего фреймворка.

Еще одним плюсом можно считать его среду разработки: она построена на базе Eclipse'а и разобраться в ней человеку, знакомому с Eclipse, не составит никакого труда.

Ranorex

Разработчик: Ranorex GmbH

Сайт разработчика: http://www.ranorex.com/

Цена за 1 лицензию: $1.100 – $3.600

Языки программирования: C#, VB .NET

Встроенные возможности: Record & Play, DDT, Distributed testing и др.

Поддержка тестирования приложений: Win32, Java, .NET, Qt, Delphi, Web, Android/iOS и др.

Один из самых молодых инструментов, однако при этом один из наиболее интересных с точки зрения поддержки различных типов тестируемых приложений. Отдельно стоит отметить нативную поддержку тестирования под iOS и Android, так как автоматизация тестирования мобильных приложений набирает обороты и требуется всё чаще.

Еще один плюс – используемые языки программирования: весьма популярные и очень мощные, с их помощью можно создать весьма мощный фреймворк.

Rational Functional Tester (RFT)

Разработчик: IBM

Сайт разработчика: http://www.ibm.com/

Цена за 1 лицензию: $3.200 – $13.200

Языки программирования: C#, VB .NET

Встроенные возможности: Record & Play, DDT, Distributed testing и др.

Поддержка тестирования приложений: Win32, Java, .NET, Web, PowerBuilder и др.

Довольно мощный инструмент, обычно применяемый в больших корпорациях совместно с другими продуктами IBM. Особо ничем не выделяется среди подобных инструментов.

Selenium (WebDriver)

Разработчик: открытое сообщество

Сайт разработчика: http://seleniumhq.org/

Цена: бесплатный с открытым исходным кодом

Языки программирования: Java, Python, C#, Ruby

Поддержка тестирования приложений: Web-приложения

Поддержка браузеров: Internet Explorer, Mozilla Firefox, Google Chrome, Opera, Safari и др.

Дополнительные возможности: расширение Selenium IDE для браузера Mozilla Firefox, обеспечивающее возможности Record & Play и преобразование записанных дествий в код на поддерживаемых языках программирования.

Этот инструмент стоит немного особняком по сравнению с уже рассмотренными инструментами, так как он, во-первых, совершенно бесплатный (что является несомненным плюсом), а во-вторых, предназначен для тестирования исключительно веб-приложений практически для любых браузеров и платформ.

На данный момент это наиболее популярный инструмент в мире для тестирования веб-приложений, для него существует огромное колическтво документации на различных языках, расширений, сообществ и т.п., что делает его очень привлекательным для использования.

Sikuli

Разработчик: открытое сообщество

Сайт разработчика: http://www.sikuli.org/

Цена: бесплатный с открытым исходным кодом

Языки программирования: visual scripting, Jython

ОС: Windows, Linux/Unix, MacOS X, iOS, Android

Встроенные возможности: Record & Play, DDT, Distributed testing и др.

Sikuli попала в наш обзор потому, что является в своем роде уникальным явлением. В отличие от всех других инструментов, которые работают на низком уровне, обращаясь к элементам управления и оперируя их свойствами, Sikuli работает с изображениями, скриншотами элементов управления и окон. Достоинства Sikuli – возможность работать с любым приложением в любой операционной системе. Главный недостаток – работа с изображениями обычно весьма нестабильна и скрипты приходится часто менять. Однако бывают случаи, когда никакой другой способ не подходит и использование Sikuli вполне оправданно.

Другие инструменты

В этом разделе мы кратко перечислим другие инструменты, которые можно использовать для автоматизации тестирования, но которые, однако, не получили сильного распространения по тем или иным причинам.

- **MS Visual Studio – Coded UI.** Инструмент для автоматизации тестирования веб и .NET-приложений. Не очень популярный, так как лицензия на Visual Studio весьма дорогая, а возможности инструмента весьма ограничены.
- **AutoIt.** Очень простой бесплатный инструмент для тестирования Win32 приложений. Для написания скриптов используется язык, похожий на Basic.
- **WinTask.** Недорогой инструмент для тестирования Win32 и веб-приложений.
- **Test Automation FX.** Простой инструмент для тестирования Win32 и .NET-приложений. Интегрируется с Visual Studio.
- **Vermont HighTest Plus.** Простое приложение для автоматизации тестирования Win32 и веб-приложений.

Устаревшие инструменты

Здесь мы перечислим инструменты, которые когда-то были популярны, но в данный момент уже считаются устаревшими и не поддерживаются разработчиками. Однако у вас все еще есть возможность встретиться с ними в проектах, где автоматизация существует уже несколько лет и переписывать скрипты, используя более современные инструменты, считается нерентабельным.

- **Rational Robot** – предшественник Rational Functional Tester
- **WinRunner** – предшественник QuickTest Pro
- **Selenium RC** – предшественник WebDriver'а, поддерживается с целю обратной совместимости, не рекомендуется для создания новых проектов ввиду различных ограничений

6.4 Какой инструмент изучать?

Если вы только начали знакомиться с автоматизацией тестирования, но у вас нет конкретной задачи для автоматизации, то вопрос «что изучать?» для вас будет весьма актуальным. В этом разделе мы постараемся на него ответить.

Функциональное тестирование

- Если вы собираетесь работать в какой-то компании, где автоматизация тестирования уже внедрена, вам нет нужды гадать. Просто поинтересуйтесь, какие инструменты используются в этой компании, и начните изучение с них.
- Если вы готовитесь внедрять автоматизацию на своем проекте «с нуля», то вам необходимо рассмотреть несколько вариантов. В этом случае всё зависит от того, какие приложения вам придется автоматизировать и сколько денег компания готова потратить на автоматизацию.
- Если вы хотите заниматься автоматизацией тестирования десктопных приложений в одной из стран СНГ, но еще не определились с компанией, в которой будете работать, начните изучать TestComplete. Из-за невысокой цены этот инструмент чаще всего выбирают компании, аутсорсящие разработку в наши страны. Еще один плюс в изучении TestComplete заключается в том, что в этом инструменте представлены почти все подходы, использующиеся в автоматизации (DDT, KDT и др.).
- Если вы собираетесь заниматься автоматизацией веб-приложений, то наиболее вероятным инструментом, который вы будете использовать, является WebDriver. Если вы не уверены насчет выбора языка программирования, который будете использовать вместе с WebDriver, имейте ввиду, что среди поддерживаемых WebDriver'ом языков наиболее часто используется Java, а самым простым для изучения является язык Python.

Нагрузочное тестирование

- Если вы просто хотите познакомиться с нагрузочным тестированием, выбирайте JMeter. Он бесплатный, прост в установке (его достаточно распаковать из архива) и по нему есть большое количество учебников и тренингов.
- Во всех остальных случаях вам придется изучать тот инструмент, который подходит вам для работы.

Тестирование мобильных приложений

В настоящее время ни один из инструментов для автоматизации тестирования мобильных приложений не является лидером.

- Если вы хотите просто познакомиться с предметом автоматизации, то для обучения лучше взять один из инструментов, предназначенных для функционального тестирования, а затем использовать полученные знания, пользуясь любым инструментом для тестирования мобильных приложений.

- Во всех остальных случаях выбирайте тот инструмент, который подходит для вашего проекта или задачи.