APPENDIX A

Real-World Examples

In Chapter 6, we've discussed how to design the test automation to fit the architecture of the specific application. This appendix describes three examples based on real-world applications that I was involved in designing test automation solutions for.

Note that these examples are only *based* on real-world cases. Some details have been changed for clarity purposes, to protect Intellectual Property, and for anonymity purposes. A couple of ideas that are presented here did not come to full fruition in the real projects, mainly due to priority changes, internal politics, etc., but the technical feasibility of all of them were at least proven.

Example 1 – Water Meters Monitoring System

One of my clients develops electronic water meters and a system for controlling and monitoring the meters in towns and districts. The meters themselves, including the electronics and the embedded software, are developed and tested separately from the control and monitoring system for the towns and districts.

They called me to help them build a test automation infrastructure for the control and monitoring system. This system is a website that can display information about the meters and their statuses over a Google Maps widget, in addition to more standard ways like tables, forms. and charts. The system has the architecture depicted in Figure A-1.

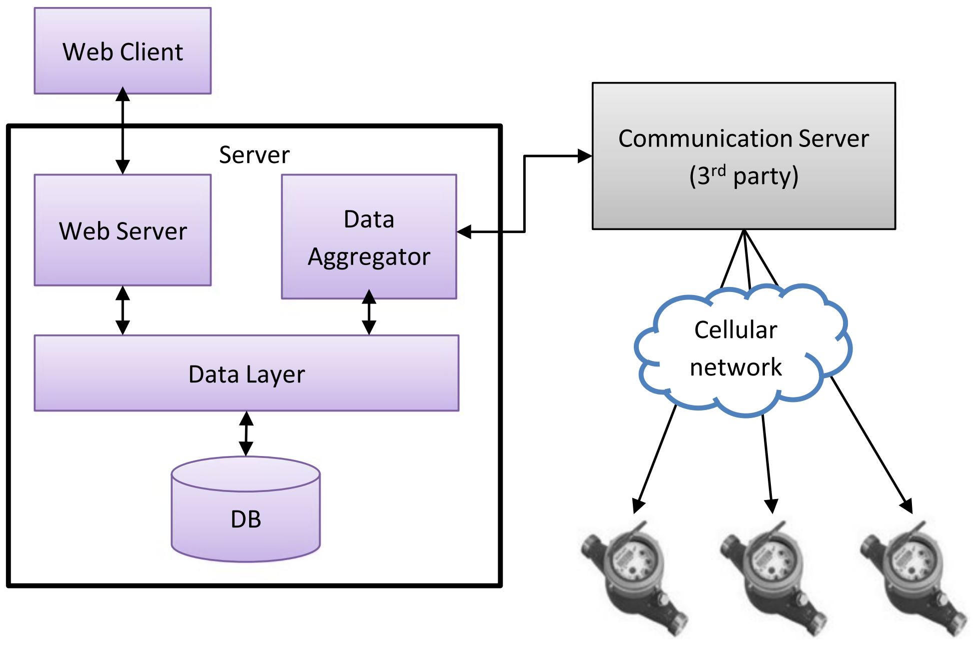

Figure A-1. *Architecture diagram for water meters monitoring system*

Here's a description of the components and the interactions between them:

- The Web Client is the HTML pages along with JavaScript and CSS files that the browser uses in order to show the UI and interact with the user. Like any other website, these HTML pages and related files, as well as the data to display are retrieved from the Web Server on the server side. The main page also displays a Google Map widget and uses the Google Maps JavaScript API to add markers in the locations of the meters on the map.
- The server side contains the following components:
  - The Web Server is the server side of the website, which provides the HTML, JavaScript, and CSS files to the clients' browsers, as well as Web API that provides the data that needs to be displayed. The Web Server mainly retrieves the data and receives events on important updates from the Data Layer. In addition, it can send commands to the meters, also through the Data Layer.

    - The Data Layer manages the access to the data and provides relevant notifications to the layers above it about updates of data. The Web Server uses the Data Layer to retrieve data from the database, get notifications and send commands, while the Data Aggregator uses it to store and update data in the database and receive commands to send to the meters.
    - The Data Aggregator component receives a stream of near real-time data from all of the meters throughout the district, through the Communication Server. It aggregates it and transforms it to the structure that the Data Layer requires. In addition, it sends commands to the meters according to requests from the Data Layer. Before sending a command, the Data Aggregator first translates it from the internal data structures of the application to the protocol that the meters can understand.
- The Communication Server, which was developed for my client by a subcontractor, behaves like a specialized router between the application and the meters. The Communication Server itself doesn't read or change the content of the messages but only dispatches them to and from the right meter, through a cellular network.
- Finally, the meters themselves are the physical electro-mechanic devices that measure the water flow, alert on problems, and accept commands like open or close the tap. These devices are equipped with a cellular antenna to communicate with the Communication Server. Even though these devices also have software in them, this software is developed in a completely separate group in the company and in different cadence. Therefore, for the purpose of testing the Control and Monitoring system, the software on these devices can also be considered as 3rd party, and not an integral part of the system.

When I was called to help the QA team establish a framework for their automated tests, I asked them what was important for them to test. The initial response was: "end to end." However, when we dove deeper, it turned out that what they really care about are the UI and server pieces, which are all the software that they build themselves in that group, while the Communication Server and the Meters themselves are of less interest as they were considered stable and reliable.

Simulating the Communication Server

Then I asked them how they performed their manual tests today, and it turned out that they're using a database that is a copy of one of their clients and performs all of the tests through the Web Client's UI. Only at our meeting they realized that they're not testing the Data Aggregator component at all, even though a big part of the logic is there. When we discussed how we should approach it from a test automation perspective, we came to the conclusion that we need to create a simulator for the meters and the Communication Server. This is the only way we can also test the Data Aggregator component and also is necessary in order to create reliable tests that cannot affect one another.

At first it seemed like an unrealistic mission, because the QA team and their manager were stressing that on one hand, because only the developers have the knowledge of the protocol, then they're the ones that should develop the simulator; but, on the other hand, the developers are too busy and will have no time for that in the foreseeable future. But then I suggested that the developers will only provide the documents of the protocol and a little bit of knowledge transfer, and that the automation developers will actually develop the simulator. So, the QA manager called the relevant developer and asked him if he can help us with that, and he promptly agreed! It turned out that they have a very detailed document of the protocol, and for any unclear or outdated details he was happy to assist us. Figure A-2 shows the final architecture of the test automation.

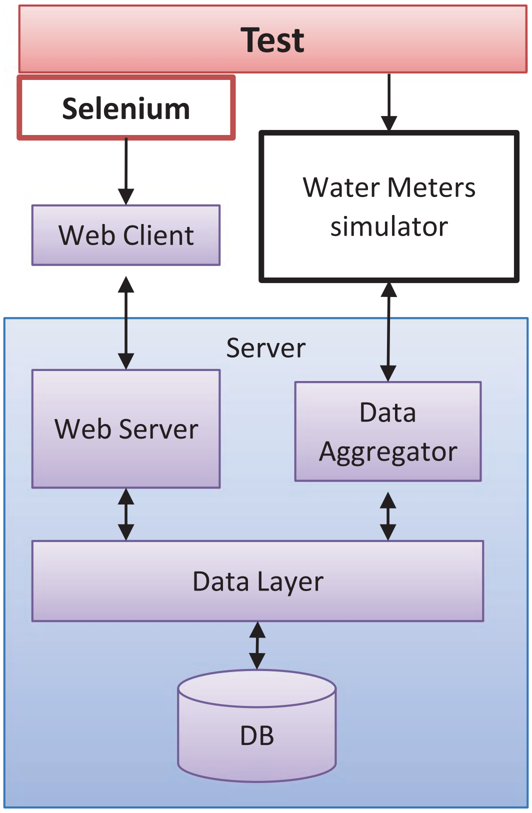

Figure A-2. *Final test architecture for water meters monitoring system*

Dealing with Google Maps

The other architectural challenge was with the Google Maps component. Like any web page that contains a Google Map component, the architecture is as shown in Figure A-3.

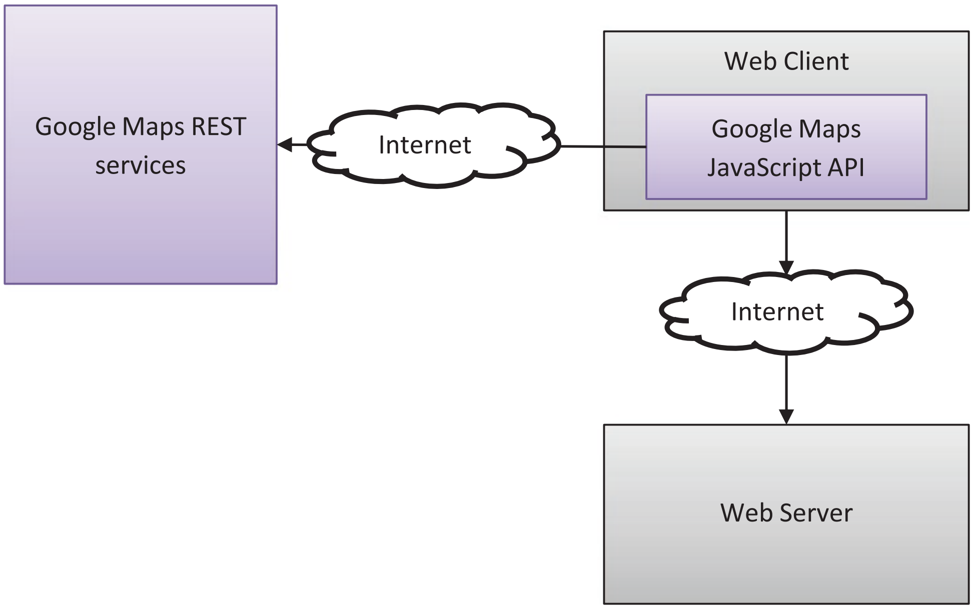

Figure A-3. *Web Client architecture with Google Maps*

As Selenium was the obvious choice for our UI automation technology, when it comes to a graphical component like Google Maps, it isn't trivial to work with. Google Maps is based on an HTML 5 SVG (Structured Vector Graphics) element, which contains all the raw data that represents lines and shapes required to display the map. While Selenium has access to this data, it's unfeasible to reason about it and to identify the *meaning* of what the user sees. However, Google Maps also uses a JavaScript API behind the scenes that provides more meaningful information and operations. Selenium on its part can send any JavaScript code that we'll give it to be executed directly in the browser and get the result back, so we can use Selenium to query and manipulate the JavaScript API of Google Maps, and thus we can get a meaningful notion of what the user sees.

Figure A-4 shows the architecture of the solution with regard to the Google Maps component.

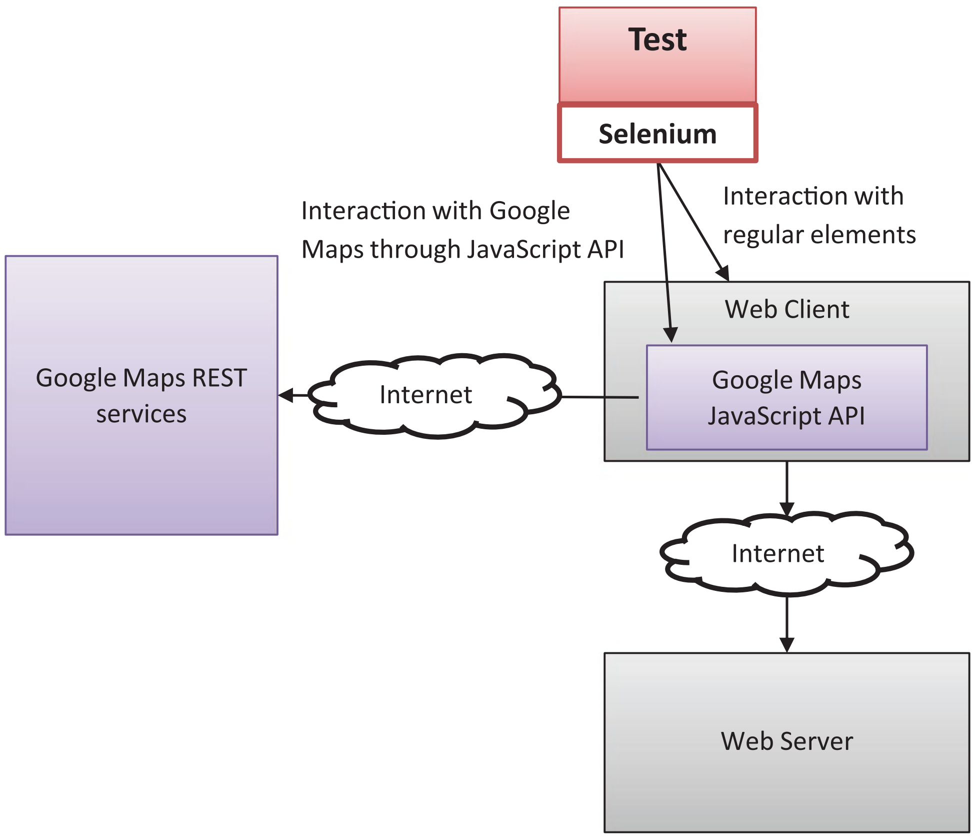
Test
Selenium
Interaction with Google Maps through JavaScript API
Interaction with regular elements
Web Client
Google Maps JavaScript API
Google Maps REST services
Internet
Internet
Web Server

Figure A-4. *Test architecture for Web Client with Google Maps component*

Example 2 – Forex Trading System

To the customer of my second, example I came after they have started implementing automated testing, but they reached out to me because they had some issues, mainly with the stability of their tests. After investigating and fixing some apparent issues, we started discussing the business processes that they use around the automated testing. Pretty late in the conversation, I realized that they don't run the tests on Sundays. It may look obvious to you, but if you consider that in Israel the work week is Sunday to Thursday (i.e., Sunday is just another regular working day), you'd probably understand why I was surprised. The people I talked to took it for granted that tests cannot be run on Sundays, because the stock exchanges don't operate on Sundays. Obviously, I understood why the real system doesn't work on Sunday, but I still didn't understand why tests cannot run on Sundays, and this was true both for manual tests as well as the automated tests. Eventually I realized that they depend on a third-party service that provides the real-time data from various stock exchanges throughout the world, and most of the features of the application are unavailable when there's no trade. This is all fine as far as the production system is concerned, but it turned out that the same service is used in the testing environment too, and that's what prevented them from testing on Sundays.

While this service by itself was pretty stable, the dependency on that service also implied that the tests had some awkward dependencies on specific, real stock symbols, and from time to time the stocks that the tests used had to be replaced due to changes of some properties of these stocks in the real market. The tests didn't only depend on the existence of particular stock, but also on their properties. Even though the existence of the stock almost never changed, their properties did change from time to time, requiring them to change the tests accordingly or to choose another stock for the test. In addition, there were certain situations where specific stocks were temporarily unavailable due to real trading constraints. Obviously, the test didn't have any control over that, and altogether, this was a significant part of what made the tests unstable.

Furthermore, it turned out that the most basic functionality of the system cannot be validated, because the test had no control over the most important input of the system, which is the data that is received from this trading service. For example, they couldn't predetermine the loss or profit of a trading transaction, because they couldn't tell how the stock would behave.

After that conversation I realized I must have a better understanding of the architecture of their system and so I asked them to show me an architecture diagram. The architecture diagram that they showed me was similar to the one in Figure A-5.

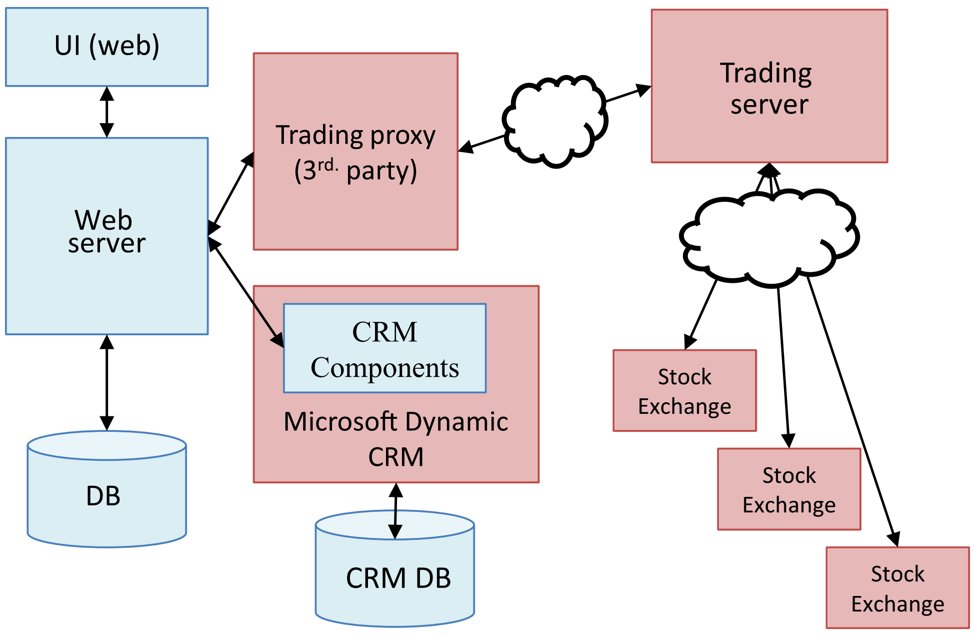

Figure A-5. *Forex trading system architecture*

The Solution

Due to the conclusions from that discussion, we decided to implement a simulator for the Trading Server (and its proxy). This allowed us to have better control of what's going on in the tests and to simulate scenarios (including the very basic ones!) that we couldn't test before. Obviously, it made the tests more reliable and allowed the tests to run at any time, including Sundays.

Instability Caused by CRM

Another component that affected the stability of the tests was the CRM application. The main system was connected to a Microsoft Dynamic CRM system that stored the information about their users. The Microsoft Dynamic CRM application was also extended and tailored to their needs (the "CRM Components" in the diagram). The team that worked on this part of the system was separated from the team that worked on the main Web Server, but the testing environment of the main Web Server was configured to work with the same testing environment of the CRM team. Due to technical constraints, the CRM Components developers used their testing environment not only for testing per se, but also for debugging during development. This meant that the CRM application was mostly unstable. As a result, whenever the main Web Server had to use the CRM component, it was communicating with the unstable CRM application that, in turn, made the tests of the Web Server unreliable too.

Isolating the Environments

Because the CRM components are tightly coupled (by design) to Microsoft Dynamic CRM, and because the goal was to test the system end to end, including the CRM Components, we've decided not to simulate the CRM as done for the Trading service. Instead, in order to stabilize the tests, we've decided to detach the test environment of the Web Server from the test/dev environment of the CRM Components, and instead to create another CRM installation inside the normal test environment. An automated build (that runs nightly but can also be triggered on demand) was created to build the code in the source control, and upon success, deploy both the Web Server *and* the CRM Components to the test environment. This allowed the developers of the CRM Components to debug their code on their original environment, and only after they verified their changes, check in the code to the source control system, thus leaving the test environment clean and stable enough to run the automated tests. Figure A-6 shows the final architecture of the test environment.

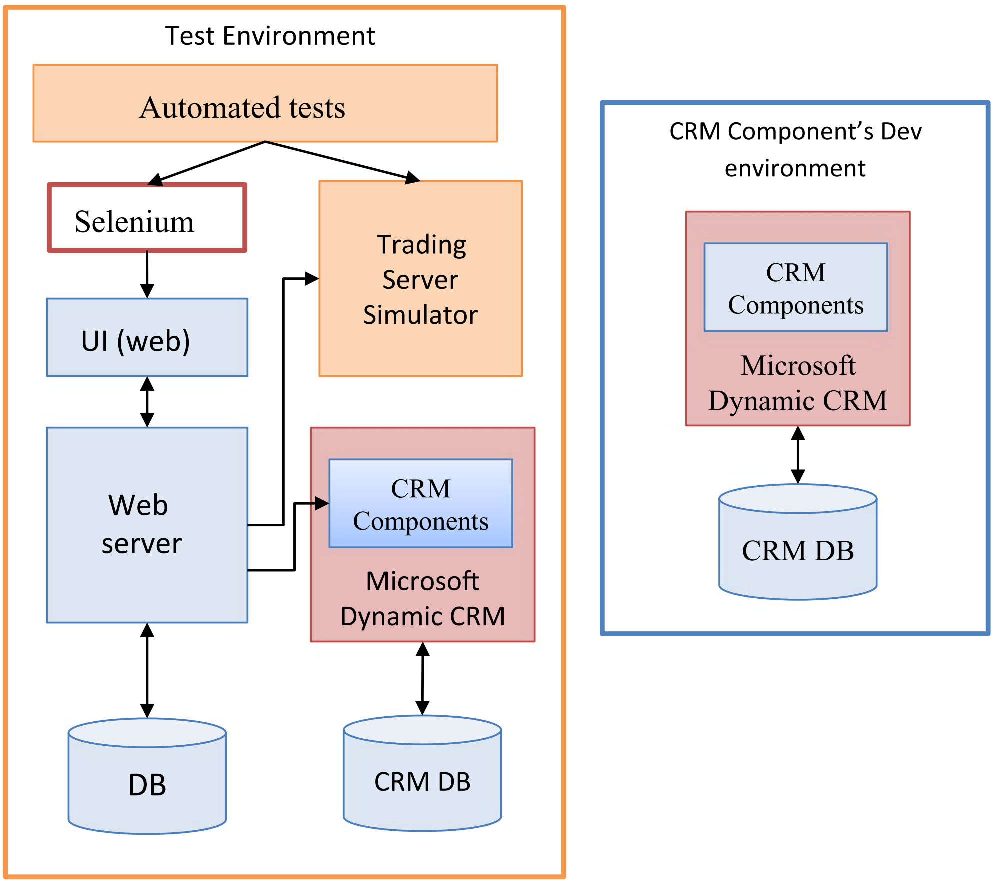

Figure A-6. *Final architecture of test environment*

Testing the Mobile Application with Abstract Test Scope

In addition to the normal web application, they started to develop a new mobile application. The mobile application has basically provided the same functionality, but the UI was organized differently to fit to the smaller form factor of a smartphone. The UI technology that was used for the mobile application was of a hybrid application, meaning that it's like a web browser embedded inside a shell of a native application. That means that technically we could use Selenium also to test the mobile application, but because the UI was organized differently, then we couldn't reuse the existing tests as is.

Therefore, in order to support testing both the regular web application and the mobile one, we decided to refactor the test code to use the Abstract Test Scope pattern that was described in Chapter 6: The test methods themselves shouldn't need to change, but we extracted a set of interfaces for every class and method that represented a business functionality (that uses the regular web application), and created a new set of classes that implemented these interfaces, but using the mobile application. We've also added a setting to a configuration file that determined whether we should use the normal web application or the mobile one, and according to that setting, the test automation instantiated the appropriate classes.

Example 3 – Retail Store Management

Our third example is of a client/server application for managing retail stores. Unlike the first two examples, which are central web sites with a single deployment, the application in this example is an off-the-shelf product that retailers can buy, install, and run on their own hardware. Because this software was an off-the-shelf product, and because it was a major and important software that the retailers used to run their businesses, it was important that it would be highly customizable and extensible. In addition, we supported several different deployment options to fit retailers of different sizes and needs. Figure A-7 shows the architecture for a full deployment of the system.

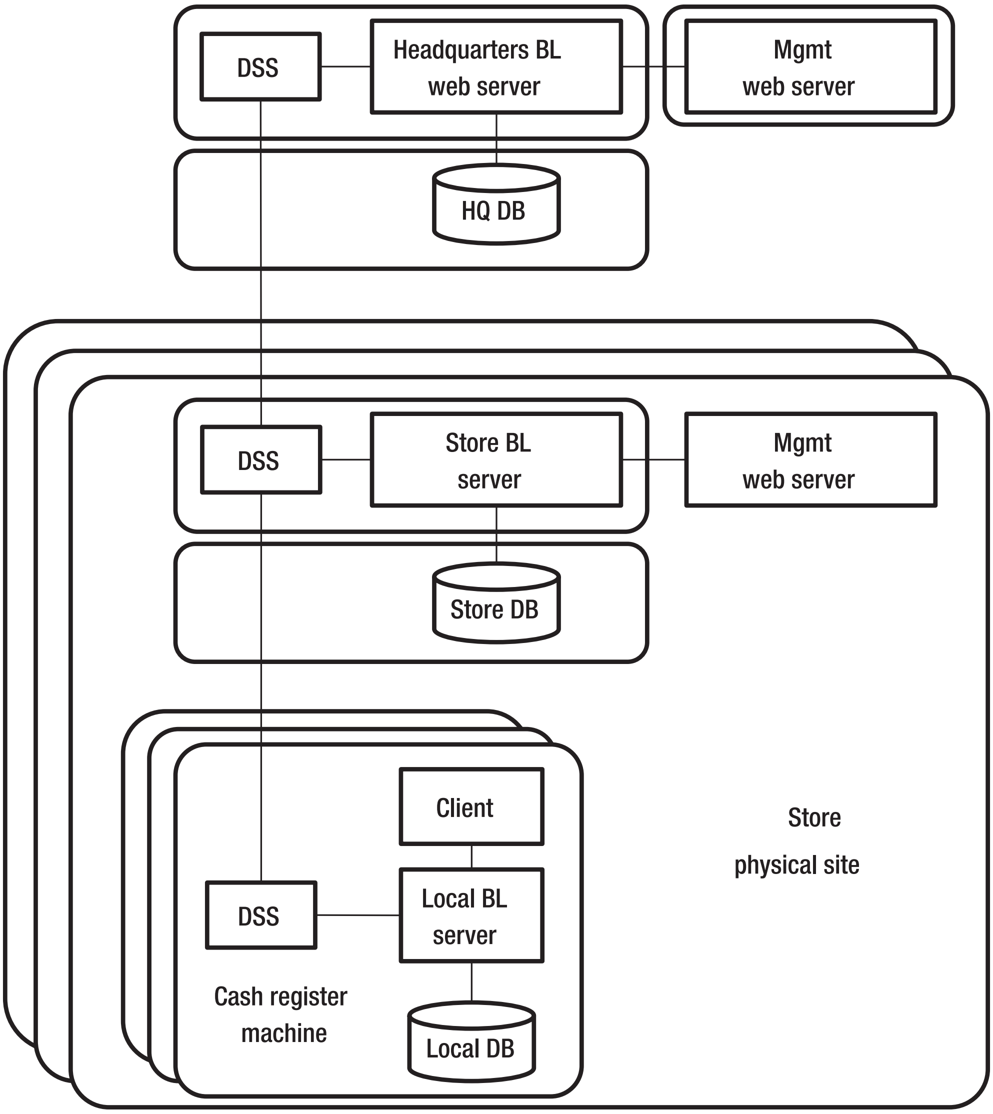

Figure A-7. *Architecture of the application in full depoloyment*

Description of the Architecture

While this architecture may look daunting at first, many of its components are simply different instances of other identical components. Let me first explain the high-level architecture described in the diagram, and then I'll explain each component.

In a full deployment configuration, which is suitable for a large chain retail store, the application is deployed on all cash register machines, a store server for each store (branch) in the chain, and a central server in the headquarters. As shown in the diagram, different components of the store server and the headquarters server may be deployed on separate machines. In addition, for redundancy purposes there can be more than one instance of the central headquarters server, but that's beyond the scope of the diagram. In particular, each deployment instance consists of at least:

- A Business Logic (BL) server – this is a REST/HTTP server that contain all of the business logic. In the full deployment option, each cash register contains its own local server and database for the sake of scalability and availability in case of a network failure. All of the BL servers, including the central servers of the stores and the headquarters. are essentially identical. The REST API was also intended to be used by advanced customers so that they can develop adopters to other systems or even create their own UIs.
- A database – the database contains all of the data that the server needs. Among other things, each instance contains the data of all products that are available in the store and their prices, and all the data of the sales transactions that are related to that machine: cash registers normally store the transactions that were handled at that same register during the same day, the store DB stores a copy of the transactions that were handled by all the cash registers in that store (branch), and the headquarter DB stores a copy of the transactions of the entire chain. In the Headquarters and Store servers, the database can be deployed on separate machines.

- Data Synchronization Service (DSS) – this component is responsible to synchronize the data between the different servers. Generally, most data changes (e.g., updates to products and prices) are transferred from the higher tiers (Headquarters, Store) to the lower tiers (cash registers), while sales transaction logs are transferred upwards. The DSS has dedicated business logic that allows it to decide which data it should transfer to which other server and which it shouldn't. For example, some products and prices can be applicable only to some stores and not the others.

Because that in the full deployment configuration the BL server is deployed both on the cash registers, at the store levels and at the headquarters, it was also called a "3-tiers" deployment.

In addition to the components that exist in every instance, the cash register machines have a dedicated UI client application that the cashiers use. This component is mostly a thin UI layer that talks to the local server that is installed on the same machine.

Lastly, the Management web server can also be deployed and connect to Store and Headquarters servers. This server provides a web interface for managing all data, including products and prices and also viewing and querying sales transaction logs and some analytical dashboards.

Minimal Deployment

As mentioned above, the application supports different deployment configurations according to the size and the needs of the particular customer (store). So above I explained the most complex configuration, and now I want to describe the simplest one.

The most minimal deployment consists of only one BL server, a database, and one client, all on one single machine, which is the cash register. Typically, one Management server will also be present, but even that's not obligatory as instead the data can be transferred from a third-party management system using the REST API. Figure A-8 shows the minimal deployment configuration.

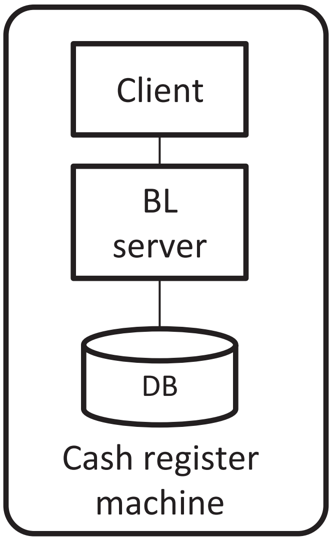

Figure A-8. *Minimal deployment configuration of the application*

The internal structure of the Server and Cash Register applications were very similar to the typical client/server architecture mentioned at the beginning of Chapter 6. However, it's important to note a few things:

1. The Service Layer was used as the public REST API. Customers used this API mainly for integrating this system with other systems and automating business processes.

2. The Business Logic layer was also exposed as a .Net API, mainly to allow customers to extend, customize, and override the default behavior of the system. This layer was made out of many components, some of which had interdependencies.

3. Each data entity (e.g., Product, Price, Sales, Customers, Cashiers, and many more) that had to be transferred to other servers using the DSS, had implemented an interface for serializing and de-serializing itself as XML. We'll talk about its consequences a little later.

Organizational Structure

In order to understand the considerations for the different test automation solutions, it's also important to understand the organizational structure (in Chapter 8 we've discussed the relationships between the organizational structure, architecture, and test automation). The application was developed by a group of about 200 people: about half of them are developers, others being mainly testers and product managers. There was a dedicated team for the Management web application, another team for the Cash Register client, and yet another one for the DSS. The BL server was developed by a few other teams, more or less corresponding to the different components in the Business Logic layer, but each of them was also responsible for the corresponding parts in the Service and Data Access layers. The project was managed using a Scrum methodology.

Test Automation Solutions

Because the Business Logic layer was exposed as a public API it was important (and natural) to write most of the tests as **component tests**. For each component, its tests would mock the other components. In addition, these tests were mocking the DAL, while other tests would test the DAL of each component separately. Naturally, as these tests were very light and fast, they were being run as part of the CI build (see Chapter 15). This build compiled and ran all of the tests of all the components together.

However, because of the interdependencies between the components, and because the Service Layer also exposed a public REST API, it became important to test the integration of the entire server as a whole on a regular basis. For this reason, an infrastructure for **integration tests** was also created. The general guideline was that for each User Story the teams should write at least one integration test in addition to the few component tests that they were doing anyway. These integration tests were testing the entire server and database as one unit, by communicating with it only through the REST API. The infrastructure of these tests created a dedicated empty DB before all the tests started to run for isolation purposes (see Chapter 7 about isolation) and used this DB throughout the tests. This allowed developers to run these tests on their local machines before check-in. These tests were also added to the CI build by installing the new version of the server on a dedicated test machine and running all the tests there.

While these tests were slower than the component tests, they were still pretty fast. While tens of component tests were executed in a second, most integration tests took between 0.5 seconds to 1 second. At the beginning it wasn't an issue, but when the number of tests raised to thousands, it took 30–40 minutes to run all the tests, and that started to become an obstacle. Even though it doesn't sound much, when developers were under pressure to release a new feature, they often skipped running all the tests before check-in, and then, mostly due to a conflict with a change of some other developer, the build failed. At that moment no one (of 100 developers!) was allowed to check- n their changes until the build was fixed. This was a huge bottleneck and time waster. Add to that the build itself took even longer due to the overhead of compilation and running the component tests (which was summed up to about 50 minutes) and you'll see how frustrating it could be.

The solution was to allow the tests to run in parallel. On developers' machines the tests were split between 4 threads (which was the number of cores on each dev machine). In the build, we split the tests to 10 different machines, each running 4 threads! That allowed us to reduce the local run time to about 10 minutes and the overall build time, including the compilation and component tests, to about 15 minutes! Of course, that each thread had to work with a separate instance of the application and with a separate instance of the database. Because the tests always started with an empty database – it wasn't a big issue.

Date/Time Simulator

There was a bunch of features that were dependent on dates and times. Fortunately, mainly due to the extensibility requirements, the system incorporated a dependency injection (DI) mechanism. In addition, partly thanks to the component tests, but also thanks to a proper architectural decision that was made early on, all uses of the `System.DateTime` class in .Net were abstracted behind an interface that a singleton object has implemented. This allowed us to develop a date/time simulator component that was injected into the system using the DI mechanism and control it from the tests through a specifically created REST end point. Figure A-9 shows the architecture of integration tests including the Date/Time simulator.

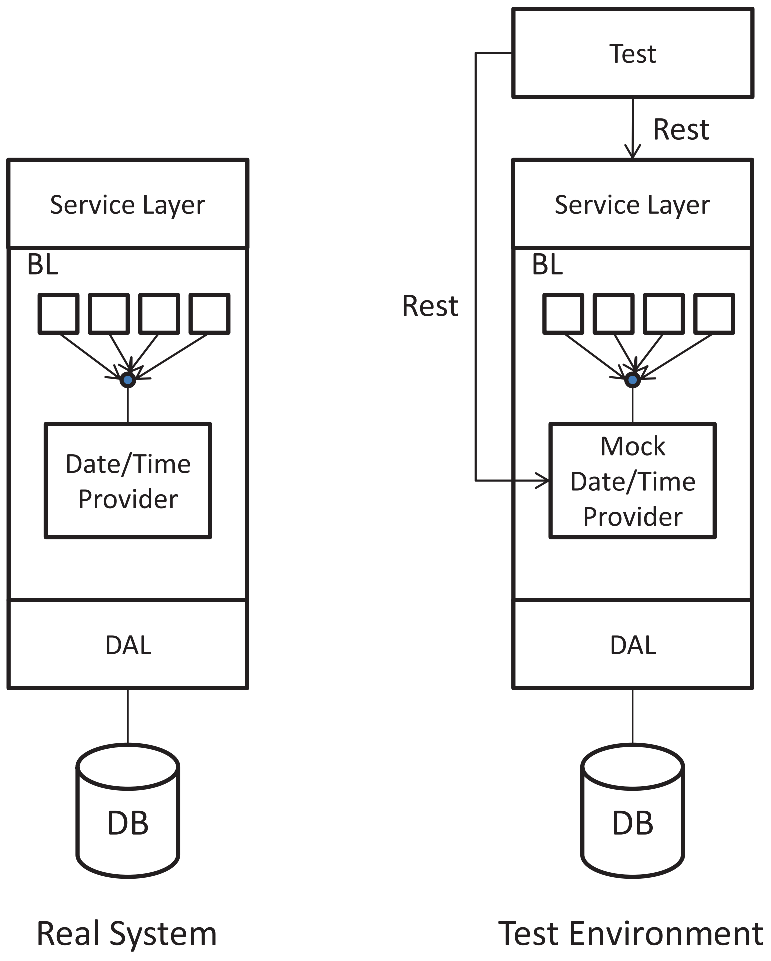

Figure A-9. *Architecture of the integration tests including Date/Time simluator*

3-Tiers Tests

The Management web application, Cash Register client and DSS teams were writing unit tests but no higher scope tests. However, manual testers often found bugs that were happening only in a full (3-tiers) deployment. This was mainly because of the special serialization and de-serialization code that should have been implemented separately for each entity and was not being used in the single-tier integration tests nor in the component tests.

Luckily, the infrastructure of the integration tests distinguished between REST requests that normally come from the Cash Register and requests that normally come from the Management application. This allowed us to make a pretty small change in the infrastructure of the tests to allow them to run in a 3-tiers configuration. Instead of directing all of the requests to the same URL, the requests that normally come from the Management application were directed to the HQ server, and the requests that normally come from the Cash Register were directed to the server of the cash register. Because the 3-tiers were configured and connected through the DSS, this allowed the existing tests to run in this configuration and thus test the DSS and all of the serialization and de-serialization code seamlessly, without having to change the tests! This can be considered as a form of an abstract test scope, as described previously, but even without having to implement two separate sets of classes for each configuration.

In fact, the change was a little more complicated than just directing the requests to different URLs, because it was necessary to wait for the data to propagate to the other tiers. For that, we used special monitoring API of the DSS that allowed us to know when the data has completed to propagate, so we could wait just the necessary amount of time and not more. However, because these tests were much slower (about 1 minute per test, as opposed to 1 second…), it didn't make sense to run them in the CI, so we created a different build process that ran nightly.

End-to-End Tests

Regarding the Management application and the Cash Register client application, even though they were covered by unit tests, it wasn't enough to make sure that they're working correctly along with the server. Occasionally there were discrepancies between the content of the requests that the clients were sending to what the server has expected, or similarly between the responses that the server sent and what the client was expecting. Therefore, at some point we decided that we cannot avoid automatic end-to-end tests. These tests were using Microsoft Coded UI for the Cash Register client, and Selenium to interact with the Management web application and were also running in a dedicated nightly build process. In most cases, a dedicated team (reporting to the QA manager) were writing these tests. Because the integration tests were already testing the 3-tiers deployment, it was enough for the end-to-end tests to test a single server but with the combination of the Management server and the Cash Register client. So, a single server gave us the missing coverage that we needed.

APPENDIX B

Cleanup Mechanism

As explained in Chapter 7, writing robust test cleanup code is not trivial and can be error prone, though it's possible to create a mechanism that solves the majority of these problems. This appendix explains how to create such a mechanism step by step and the recommended way to use it. Note that this mechanism already exists built in the Test Automation Essentials project (see Appendix C), so at least for MSTest you can use it almost out of the box, while for other frameworks for .Net (e.g., NUnit or xUnit), you should adopt the mechanism to the framework yourself. For any other language outside the .Net framework (like Java, Python, JavaScript, etc.), you should be able to implement a similar mechanism according to the explanations in this appendix.

Understanding Callbacks and Delegates

The cleanup mechanism is based on a programming concept called *Callback*. Different programming languages implement callbacks in a different ways and syntaxes, but all general-purpose languages have some way to implement it. A callback is a way to reference a function or method and keep it for a later use. In C# this is achieved through *delegates*, in Java through an interface with a single method (typically `Consumer` or `Function`), in JavaScript and Python you can simply assign a function to a variable without calling it (i.e., without the parenthesis after the function name). After you store a reference to a function in a variable, you can call the function through the variable. This allows you to assign different functions to the variable according to different circumstances and then call the chosen function simply by calling what the variable references. Some languages have a short syntax for declaring short inline anonymous functions, often called *Lambdas*, so you don't have to create and declare a whole new method if you just want to store a reference to that simple method in the callback variable.

In c#, the predefined delegate type `System.Action` is defined as follows:

```
public delegate void Action();
```

This definition means that a variable of type `Action` can reference any method that takes no argument and doesn't return a value (i.e., return void) as a callback.

Listing B-1 shows an example of delegates and lambda expressions in C# that use the `Action` delegate type.

Listing B-1. Example of the concept of Callbacks using Delegates and Lambdas in C#

```
private static void Foo()
{
    Console.WriteLine("Foo");
}

public static void Main()
{
    // The following line doesn't call Foo, just stores the callback for
    // later use.
    Action x = Foo;

    // do other stuff...

    // The following line actually calls Foo
    x();

    // Assign a lambda expression (anonymous inline method) to x for later
    // use. The empty parenthesis mean that it takes no arguments, and the
    // code between the { } is the body of the method.
    x = () => { Console.WriteLine("This is a lambda expression"); };

    // do more stuff...

    // Now the following line calls the body of the lambda expression.
    x();
}
```

Building the Cleanup Mechanism

Let's get back to our cleanup problem. I'll describe the full solution step by step, in order to explain the need of each nuance of the final solution. But before discussing the solution, let's recall what's the problem is that we're trying to solve.

The Problem

If, for example, the test performs five operations that should be undone (e.g., creates entities that should be deleted), and the fourth operation fails, then we want only the first three to be undone, because the fourth and fifth operations didn't actually happen. Let's take a more concrete example: suppose we're building an e-commerce website, and we want to test that when buying three items and adding a coupon, the system will give the user the appropriate price reduction. In order to test this scenario, we should first add three products to the catalog and also define the coupon details. Listing B-2 shows the original test method. The first four statements perform the operations that we'll want to clean up when the test ends.

Listing B-2. The original test method, without the cleanup code

```
[TestMethod]
public void CouponGivesPriceReduction()
{
    var usbCable = AddProductToCatalog("USB Cable", price: 3);
    var adapter = AddProductToCatalog("Car AC to USB adapter", price: 7);
    var phoneHolder =
            AddProductToCatalog("SmartPhone car holder", price: 20);
    var coupon = DefineCouponDetails("12345", percentsReduction: 20);

    var shoppingCart = CreateShoppingCart();
    shoppingCart.AddItem(usbCable);
    shoppingCart.AddItem(adapter);
    shoppingCart.AddItem(coupon);

    var totalBeforeCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(30, totalBeforeCoupon,
                "Total before coupon added should be 30 (3+7+20)");
    shoppingCart.AddCoupon(coupon);

    // Expect 20% reduction
    decimal expectedTotalAfterCoupon = totalBeforeCoupon * (1 - 20 / 100);
    var totalAfterCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(expectedTotalAfterCoupon, totalAfterCoupon,
                    "Total after coupon");
}
```

The Basic Solution

In order to leave the environment clean, we need to delete the products we added to the catalog and the coupon definition. But as we said, if one of the operations failed, we don't want to perform its cleanup code or the cleanup code of the operations that didn't happen yet. Therefore, the basic solution is to manage a list of callbacks (`Action` delegates) that each does an atomic cleanup operation. Immediately after successfully performing an atomic operation that creates an entity or changes the state of the environment, we should add to that list a delegate of a method (or lambda) that undoes that operation. In the cleanup method of the testing framework, we'll loop over the list of delegates and invoke them one by one. As we're using MSTest, we'll do it in the method that is decorated with the `[TestCleanup]` attribute (virtually all other unit testing frameworks for any language have a similar method). Let's first define the generic cleanup mechanism inside our test class, as shown in Listing B-3.

Listing B-3. Basic cleanup mechanism

```
private readonly List<Action> _cleanupActions = new List<Action>();
private void AddCleanupAction(Action cleanupAction)
{
    _cleanupActions.Add(cleanupAction);
}

[TestCleanup]
public void Cleanup()
{
    foreach (var action in _cleanupActions)
    {
        action();
    }
}
```

Now we can use it inside of our test method as shown in listing B-4.

Listing B-4. Using the cleanup mechanism inside the test method

```
[TestMethod]
public void CouponGivesPriceReduction()
{
    var usbCable = AddProductToCatalog("USB Cable", price: 3);
    AddCleanupAction(() => RemoveProductFromCatalog(usbCable));
    var adapter = AddProductToCatalog("Car AC to USB adapter", price: 7);
    AddCleanupAction(() => RemoveProductFromCatalog(adapter));
    var phoneHolder =
                AddProductToCatalog("SmartPhone car holder", price: 20);
    AddCleanupAction(() => RemoveProductFromCatalog(phoneHolder));
    var coupon = DefineCouponDetails("12345", percentsReduction: 20);
    AddCleanupAction(() => RemoveCouponDefinition(coupon));

    var shoppingCart = CreateShoppingCart();
    shoppingCart.AddItem(usbCable);
    shoppingCart.AddItem(adapter);
    shoppingCart.AddItem(coupon);

    var totalBeforeCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(30, totalBeforeCoupon,
                "Total before coupon added should be 30 (3+7+20)");

    shoppingCart.AddCoupon(coupon);

    // Expect 20% reduction
    decimal expectedTotalAfterCoupon = totalBeforeCoupon * (1 - 20 / 100);
    var totalAfterCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(expectedTotalAfterCoupon, totalAfterCoupon,
                "Total after coupon");
}
```

If, for example, adding the "phone holder" product (the third3 statement) fails, then the only cleanup action that we've added so far is for removing the "USB cable"(in the second statement), and therefore this is the only cleanup action that will be called by the `Cleanup` method. Note that the `AddCleanupAction` method-calls in the test code *don't call* the `RemoveProductFromCatalog` and the `RemoveCouponDefinition` methods. `AddCleanupAction` only *adds a reference to these methods* to the `_cleanupActions` list, which is used to invoke them only in the `Cleanup` method.

However, there are two problems with this cleanup approach:

1. The test method is now cluttered with a lot of technical code, which makes it much less readable.
2. Whenever we want to call `AddProductToCatalog` or `DefineCouponDetails`, we need to remember to add the appropriate cleanup action. This is very error prone and introduces duplication because each call to `AddProductToCatalog` or `DefineCouponDetails` should be followed by a call to `AddCleanupAction` with the corresponding cleanup callback method.

Gladly, the solution to both of these problems is very simple: Just move each call to `AddCleanupAction` to inside the appropriate method that creates the entity that we need to clean. In our case, we should move these calls to `AddProductToCatalog` and `DefineCouponDetails`. Listing B-5 shows how to add the call to `AddCleanupAction` to `AddProductToCatalog`:

Listing B-5. Adding the call to AddCleanupAction to AddProductToCatalog

```
private Product AddProductToCatalog(string name, decimal price)
{
    Product product = ... /* original code of AddProductToCatalog comes
    here (e.g. send HTTP request or add the data directly to the database,
    as appropriate, and returning a new instance of an object that
    represents the new product) */

    AddCleanupAction(() => RemoveProductFromCatalog(product));

    return product;
}
```

Now we can revert the test method back to exactly how it was in Listing B-2, but still have the cleanup code execute appropriately when the test completes.

Reusing the Cleanup Mechanism

Because we probably need this mechanism in most or even all of our test classes, it makes sense to extract this behavior to a common base class. Listing B-6 shows the cleanup mechanism in the base class.

Listing B-6. Moving the cleanup mechanism to a common base class

```
[TestClass]
public abstract class TestBase
{
    private List<Action> _cleanupActions = new List<Action>();

    public void AddCleanupAction(Action cleanupAction)
    {
        _cleanupActions.Add(cleanupAction);
    }

    [TestCleanup]
    public void Cleanup()
    {
        foreach (var action in _cleanupActions)
        {
            action();
        }
    }
}
```

And make all test classes derive from `TestBase`.

Handling Dependencies Between Cleanup Actions

That's all nice and fine, as long as the changes we're doing to the environment are independent from one another. Let's modify our example a little: suppose we want to support a different type of coupon, one that is associated with a specific product, and grants a discount only if that particular product is purchased *n* times. In this scenario, we have to define a coupon that refers to a particular product, and therefore the product must be created first, and when we define the coupon we specify the previously created product to be associated with it. If the application enforces referential integrity between the coupon definition and the product, then it should throw an error if you would try to delete the product while there's still a coupon definition associated with it. The test code in Listing B-7 shows this scenario. Notice how the `usbCable` object is passed as an argument to `DefineCouponDetailsForProduct`, which associates it with the new coupon. Assuming that `DefineCouponDetailsForProduct` adds a cleanup action for deleting this coupon, this code would throw an error in the `Cleanup` method, saying that "Product 'USB Cable' can't be deleted because there are active coupon definitions associated with it." Note that the exact line that throws the exception is not shown in the listing, but we can conclude that it should be thrown from `RemoveProductFromCatalog` (shown earlier in Listing B-5), which is called implicitly from the `Cleanup` method (shown in Listing B-6) using the delegate, because it would be called before the delegate that deletes the coupon is called.

Listing B-7. A test method that should fail due to the order of cleanup actions

```
[TestMethod]
public void CouponGivesPriceReduction()
{
    var usbCable = AddProductToCatalog("USB Cable", price: 3);
    var coupon = DefineCouponDetailsForProduct("12345", usbCable,
                                   minimumCount: 4, percentsReduction: 20);

    var shoppingCart = CreateShoppingCart();
    shoppingCart.AddItem(usbCable, count: 4);

    var totalBeforeCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(3 * 4, totalBeforeCoupon,
                    "Total before coupon added should be 12 ($3 * 4)");
    shoppingCart.AddCoupon(coupon);

    // Expect 20% reduction
    decimal expectedTotalAfterCoupon = totalBeforeCoupon * (1 - 20 / 100);
    var totalAfterCoupon = shoppingCart.GetTotal();
    Assert.AreEqual(expectedTotalAfterCoupon, totalAfterCoupon,
                    "Total after coupon");
}
```

Fortunately, the solution to this problem is also pretty simple: we just need to **call the cleanup methods in the reverse order to which they were added**. It may look like this solution is accidental and specific to our case, but if you think about it, whenever we create a dependency between entities, we either first create the independent entity (i.e., the entity upon which the other entity depends), and only then create the dependent entity, or we create both entities independently (in any order), and only then create the dependency between them, as a separate operation. In the first case, which is like we saw in the example, deleting the entities in reverse order first removes the dependent entity (which also eliminates the dependency itself) and then deletes the first (independent) entity, which we don't have any dependency on already. In the second case, when we create the dependency between the entities, we should add the cleanup action that removes this dependency (without deleting any of the entities). When the cleanup actions will be called in reverse order, then the dependency itself will be removed first, and only then the entities will be deleted. Listing B-8 shows a very simple implementation of this solution.

Listing B-8. Reversing the order of cleanup actions to resolve dependencies

```
[TestCleanup]
public void Cleanup()
{
    _cleanupActions.Reverse();
    foreach (var action in _cleanupActions)
    {
        action();
    }
}
```

Now the cleanup will be done in the right order, and no exception will occur.

Note You can obviously implement the cleanup mechanism using a Stack instead of a List, and avoid calling `Reverse`. This will probably be slightly more efficient, but it's pretty much negligible.

Handling Exceptions in Cleanup Actions

The basic idea of the cleanup mechanism is to perform exactly the relevant cleanup actions, no matter if the test failed or not, and if it failed, no matter in which line. However, what happens if one of the cleanup actions themselves fails? Unfortunately, nothing can be 100% safe, and the cleanup actions may throw exceptions too. In some cases, for example, if the test changes a global setting that affects all tests, and the cleanup action that should have reverted this setting to its original state, have failed, then it may cause the rest of the tests to fail because of it. But in most cases, it's not, and it's safe to continue to run the other tests. In addition, sometimes a failure in one cleanup action may cause *other cleanup actions* to fail as well. But nonetheless, because we can't be sure, then it's still worthwhile trying to call the other cleanup actions anyway.

In any case of failure, we must report all of the details of the exception in order to be able to investigate and fix the root cause (as described in Chapter 13). In case an exception is thrown *out* from the `Cleanup` method, the unit testing framework will report it automatically as part of the test results. But because we want to continue running the other cleanup actions, then we need to catch each exception. Therefore, we need to collect all of the exceptions that were thrown from cleanup actions and throw them only at the end of the `Cleanup` method, wrapped together using an `AggregateException` if there's more than one. If there was only one exception, then we can throw that particular exception as is. Listing B-9 shows the `Cleanup` method with the proper exception handling.

Listing B-9. Cleanup method with proper exception handling

```
[TestCleanup]
public void Cleanup()
{
    _cleanupActions.Reverse();
    var exceptions = new List<Exception>();

    foreach (var action in _cleanupActions)
    {
        try
        {
            action();
        }
        catch (Exception ex)
        {
            exceptions.Add(ex);
        }
    }

    if (exceptions.Count == 1)
        throw exceptions.Single();

    if (exceptions.Count > 1)
        throw new AggregateException(
            "Multiple exceptions occurred in Cleanup.", exceptions);
}
```

Note in .Net, it's recommended to use a list of `ExceptionDispathInfo` class instead of list of `Exception`, in order to preserve their original stack-traces. I didn't use it here in order to keep the example simpler. You can look at the implementation of this mechanism in the Test Automation Essentials, project on GitHub to see how `ExceptionDispathInfo` is used. See Appendix C for more details about the Test Automation Essentials project.

Summary

As described in Chapter 7, one technique to help achieve isolation is to clean everything that the test created when it's finished. But we also discussed the reasons why it's difficult to write cleanup code that would work correctly in all situations. The cleanup mechanism described here ensures that the exact relevant cleanup actions are called whether the test passed or failed, regardless of the exact line in which the test failed. It also ensures that cleaning up entities that depend on one another are cleaned in the right order to avoid data integrity problems and exceptions. Finally, we made sure that any exception that might occur inside a cleanup action does not prevent other cleanup actions to run, and that all of the exception details are reported so that we can investigate and fix the root cause.

APPENDIX C

Test Automation Essentials

This appendix describes the Test Automation Essentials open source project that I created, which contains various useful utilities for developing test automation in C#.

Background

When I started to work as a consultant and had to establish the infrastructure for test automation for multiple clients, I noticed that many things that I had implemented for one client was useful to other clients as well. As you probably realized by now, I hate writing duplicate code, so I looked for a way to share this code between my clients. Because naturally my clients don't share the same source control repository, I determined to make it an open source project, hosted on GitHub, for the benefit of all. Over time, whenever I wrote something that I though could be beneficial for other clients or other people in general, I added it to the project. The source code of the project can be found at `https://github.com/arnonax/TestEssentials`.

Most of the public classes and methods in these libraries have pretty extensive XML comments (similar to JavaDoc comments in Java), which Visual Studio displays as tooltips when you hover over them, in order to make it easy to use. Gradually I'm trying to cover with XML comments the rest of the classes and methods that I still haven't.

A. Axelrod, *Complete Guide to Test Automation*, https://doi.org/10.1007/978-1-4842-3832-5

Project Structure

Because my clients and the projects I was involved in use different test automation technologies, I divided the project into small and modular libraries, so that anyone can use only the libraries that fits his needs and the technologies he's using.

Accordingly, the solution consists of several C# projects:

- **TestAutomationEssentials.Common** – this project contains very generic and reusable utilities. It has no dependency on any specific technology (except for the .Net Framework itself) or library, and therefore can be used in any project. In fact, many of the utilities in this project are not even specific to test automation and can serve any .Net project. Most other projects in the solution have a dependency on this project.
- **TestAutomationEssentials.MSTest** – this project provides utilities that are useful for projects written using MSTest version 1, which has been in use up until Visual Studio 2015.
- **TestAutomationEssentials.MSTestV2** – this project provides functionality identical to **TestAutomationEssentials.MSTest**, but for **MSTest V2** (MSTest version 2), which is used since Visual Studio 2017, or as a NuGet package for projects created using Visual Studio 2015. In fact. this project does not contain any C# source files of its own, but rather links all the source files of **TestAutomationEssentials.MSTest**. Therefore. these projects are essentially always identical, except for the version of MSTest they reference.
- **TestAutomationEssentials.CodedUI** – this project provides utilities for Coded UI (Microsoft's UI Automation framework) based projects.
- **TestAutomationEssentials.Selenium** – this project provides utilities that are useful for projects that use Selenium.

In addition to these five projects that provide utilities that are useful for any project, the solution contains also the following projects that are used only internally:

- **TestAutomationEssentials.UnitTests** – contains unit tests (and some integration tests) for the TestAutomationEssentials.Common and TestAutomationEssentials.MSTest projects.
- **TestAutomationEssentials.MSTestV2UnitTests** – contains unit and integration tests for the TestAutomationEssentials.MSTestV2 project.
- **TestAutomationEssentials.TrxParser** – an internal project that is is used by the unit test projects.

Tip Looking at the unit tests can sometimes help you understand some of the utilities better. In addition, you may find it interesting to look at and debug the tests in `TestBaseTest` (and its base class) and try to understand how they work. As these tests test the integration of the `TestBase` class in `TestAutomationEssentials.MSTest` with the MSTest framework itself, they generate test code on the fly, compile it, and run the compiled code using MSTest – all within the test itself. Pretty complicated and tricky, but cool…

Note About the Unit Tests and XML Comments

Most of the code in this project was first written as part of real projects, and only then was extracted from these projects into Test Automation Essentials. As the original projects were all test projects, the code was covered (indirectly) by the tests of my client's application. When I moved the relevant code to Test Automation Essentials, I tried to retrofit unit tests for most of it to ensure that I don't break compatibility whenever I change something. However, that takes some time and in some cases it's not trivial. For that reason, by now I mainly covered with tests only the Common and MSTest (and MSTestV2) projects, but I'm working on covering the Selenium project as well.

I do a similar thing with the XML Comments, though it's typically easier and faster to do than writing unit tests.

NuGet Packages

In order to use these utilities, you don't need to get the source code from GitHub (even though you can do that as well), but rather use these libraries as NuGet packages (similar to the way we've added the Selenium WebDriver library to the project in Chapter 12). There are five libraries available, one for each project, which you can easily add to your project and start using what you need.

Features and Utilities

Following is the description of the main features and utilities contained in each of these libraries.

TestAutomationEssentials.Common

This library contains various small but very useful utility methods (mainly extension methods that were mentioned in Chapter 13), plus a few bigger features:

1. Support for configuration files. This was explained and was used in Chapter 14.
2. A `Logger` class that implements the concept of nested logging, described in Chapter 13.
3. A generic implementation of the Cleanup mechanism described in Appendix B.
4. A `Wait` class (pretty similar to `WebDriverWait` in Selenium) that provides a few methods for waiting until a given condition occurs.

Here's a more detailed explanation of the more interesting classes and methods in this library.

ExtensionMethods Class

This class contains many general-purpose extension methods, which can make your code a bit simpler and easier to read. Listing C-1 demonstrates a few usage examples.

Listing C-1. ExtensionMethods examples

```
// SubArray example:
string[] cartoons = {"Mickey", "Minnie", "Goofy", "Pluto", "Donald" };
string[] dogs = cartoons.SubArray(2, 2);

// Dictionary.AddRange example:
var numbers = new Dictionary<int, string>
{
    {1, "One" },
    {2, "Two" },
    {3, "Three" }
};
var moreNumbers = new Dictionary<int, string>()
{
    {4, "Four"},
    {5, "Five"},
    {6, "Six"}
};
numbers.AddRange(moreNumbers);

// IEnumerable<T>.IsEmpty() example:
IEnumerable<int> values = GetValues();
if (values.IsEmpty())
{
    // Do something...
}
```

DateTimeExtensions

This class also contains useful extension methods but mainly around date and time manipulation. I especially like the extension methods that makes the code read more fluently, like for example, instead of writing: `var duration = TimeSpan.FromSeconds(3);` you can simply write: `var duration = 3.Seconds();` The difference is pretty small (and a few people even find it confusing at first), but when you get used to it and read the code, it makes it much more fluent.

TestExecutionScopeManager

This class implements the cleanup mechanism that is described in Appendix B but it's not bound to a specific unit testing framework; and it allows you to nest execution scopes, to support multiple levels of cleanup, like `[TestCleanup]`, `[ClassCleanup]`, and `[AssemblyCleanup]`. If you're using MSTest there's no need to use this class directly, as TestAutomationEssentials.MSTest (and MSTestV2) already use this class internally and provide a single `AddCleanupAction` method. However, you can use this class if you need it for other unit testing frameworks or for any other purpose.

This class also uses the `Logger` class to write whenever an execution scope begins and ends.

Wait

This class exposes a few static methods for waiting for a given condition to be met. These methods throw a `TimeoutException` in case the condition didn't get met after a specified timeout. The condition is provided as a delegate, or a lambda expression (explained in Appendix B) that returns a Boolean, which indicates whether the condition has been met or not.

This class has a few overloads of methods named `Wait.Until` and `Wait.While`, which as their names imply, the first one waits *until* the condition is met, and the second one waits *while* the condition is met (i.e., until it is *no longer* met). Some of the overloads of these methods accept an error message to use in the `TimeoutException`, though if you're using a lambda expression, then you can use another overload that doesn't take an error message and automatically parses the expression in order to use it in the description. Listing C-2 shows two usage examples of the `Wait` class.

Listing C-2. Examples of using the Wait class

```
// Wait.Until example
Wait.Until(() => batchJob.IsCompleted, 2.Minutes());

// Wait.While example, with error message
Wait.While(() => batchJob.InProgress, 2.Minutes(), "Batch job
is still in progress after 2 minutes");
```

In addition to the `Until` and `While` methods, there are also `If` and `IfNot` methods. These methods correspond to the `While` and `Until` methods respectively but do not throw a `TimeoutException` when the specified period elapses. Instead they simply continue normally. This can be useful in cases that you want to wait for something to occur but may as well miss it. For example, let's consider the case where clicking a button starts an operation that may take fa ew seconds, during which a "please wait" message should appear, and we want to wait until the operation completes. But the same operation may also be very fast at times, in which case the message appears only for a blink of an eye, and the test may miss it altogether. So, in order to prevent the test from failing if the operation was very fast, you can use the `Wait.IfNot` to first wait for the message to appear, providing a short timeout (e.g., 1 second), followed by a `Wait.Until` to wait for the message to disappear, with a longer timeout (e.g., 1 minute). If the test missed the message because it was too fast, then the `Wait.IfNot` will wait in vain for the message to appear, but only for 1 second and without failing, and then the `Wait.Until` will return immediately because the message no longer appears.

TestAutomationEssentials.MSTest

The most significant class that this library provides is the `TestBase` class. This class is designed to be used as the base class for all of your tests. In case you already have a class hierarchy of tests, simply derive the lowest ones (your own test base classes) from this class.

This class adapts the MSTest specific initialization and cleanup methods to the `TestExecutionScopeManager` class from TestAutomationEssentials.Common. In addition, it exposes a virtual method that you can override and which is called only when the test fails (`OnTestFailure`), something that MSTest does not provide out of the box. This is very useful for adding any information that can help you investigate and diagnose the failures. In fact, if you import the namespace `TestAutomationEssentials.MSTest.`**`UI`**, then you'll get a slightly different implementation of `TestBase` whose default implementation of `OnTestFailure` takes a screenshot upon a failure (the screenshot is of the entire desktop, not of a browser page as it in Selenium).

LoggerAssert

In addition to the `TestBase` class, this library also contains few other utility classes. One of them is the `LoggerAssert` class, which writes a message to the log whenever the assertion is evaluated, whether it passes or fails. While generally there should only be no more than a couple of asserts at the end of each test, there are cases where few asserts in the middle of the test may still be useful (e.g., inside a loop, which is also generally discouraged in a test, but there are exceptions…). In these cases, it's useful to see in the log the assertions that passed, and not only the one that failed. Note that in order to use it properly, you should phrase the messages as expectations, (e.g., "x should be 3") and not as error messages ("x was not 3"), because these messages would appear in the log also in case of success.

TestAutomationEssentials.CodedUI

The primary goal of this library is to make working with Coded UI, through code and without UIMaps (see Chapter 3) much easier. TestAutomationEssentials.CodedUI exposes an API that resembles the one of Selenium WebDriver. Listing C-3 shows a usage example.

Listing C-3. Example of using TestAutomationEssentials.CodedUI

```
var customerDetailsPanel =
    mainWindow.Find<WinPane>(By.AutomationId("CustomerDetails"));

var customerNameTextBox =
    customerDetailsPanel.Find<WinText>(By.Name("CustomerName"));

var text = customerNameTextBox.DisplayText;
```

In addition, it provides few useful extension methods lik: `myControl.RightClick()`, `myControl.DragTo(otherControl)`, and the very useful method `myControl.IsVisible()`, which for some reason Coded UI does not provide out of the box.

TestAutomationEssentials.Selenium

This library wraps Selenium WebDriver to make the use of Selenium somewhat easier in most cases and much easier in more specific ones.

Note Currently this library is a bit too “opinionated,” which makes it hard to add it retroactively to existing projects. I’m considering changing or at least softening some of these constraints, but I have no specific details about it yet. Stay tuned…

WaitForElement

To use most of the features of this library, you need to create an instance of the `Browser` class, which wraps your `IWebDriver` object. Using this object (and though other classes that derive from `ElementsContainer`, more or that later), you can find elements using the `WaitForElement` method. This method is similar to the familiar `FindElement` method of Selenium, but with two main differences:

1. It always waits for the element to appear (and not just to exist). You can specify the timeout or use the default of 30 seconds.
2. It takes a `description` argument, which is used for automatically logging click operations, typing, etc. Providing a descriptive name in this argument helps make the log very readable.

`Browser.WaitForElement` returns a `BrowserElement` object, which on one hand implements Selenium’s `IWebElement` interface, but on the other hand, it also derives from `ElementsContainer`, which means that you can use `WaitForElement` to look for child elements inside of it, with the same benefits over the standard `FindElement` method.

Handling Frames and Windows

In my opinion, the case in which this library is most compelling is for web applications that use multiple windows or iframes (which is a web page displayed within another web page). Normally with Selenium you need to use `driver.SwitchTo().Window()` and `driver.SwitchTo().Frame()` to use other windows or frames, but the annoying thing is that if you want to use an element you already found in one context (i.e., a window or a frame, including the main page), you can’t use that element after you switched to another context, unless you switch back to the original one. If you don't do that, you'll get a `StaleElementReferenceException` (see Chapter 14), but managing the active context can be cumbersome and complicate your code. Listing C-4 demonstrates this problem.

Listing C-4. The problem with SwitchTo

```
// Navigate to a website that contains an iframe
webDriver.Url = "http://www.some-site.com/outer.html";

// find an element on the outer page
var outerButton = webDriver.FindElement(By.Id("outerButton"));

// switch to the inner frame
webDriver.SwitchTo().Frame("frame1");

// find an element on the inner frame
var innerButton = webDriver.FindElement(By.Id("innerButton"));
// clicking the inner button works normally
innerButton.Click();

// (See next comment)
//webDriver.SwitchTo().DefaultContent();

// Clicking the button on the outer page now would throw a
// StaleElementReferenceException because the outer page is not the current
// context. You must uncomment the above statement, which switches back to
// the outer page, in order to avoid the exception
outerButton.Click();
```

Note that in this example the problem doesn't look so severe, but in more complex situations, managing the current context can make the code very cumbersome and error prone.

The Solution of Test Automation Essentials to the SwitchTo Problem

The classes that derive from `ElementsContainer` (including `Browser` and `BrowserElement`) contain a method `GetFrame` that finds a frame and returns a corresponding `Frame` object (which is part of the TestAutomationEssentials.Selenium library). The `Frame` class also derives from `ElementsContainer`, so you can look for nested frames within the frame that this object represents (i.e., to find an iframe within another iframe). In addition, `Browser` has an `OpenWindow` method that invokes an operation that you provide to it (e.g. ,clicking a button), which should result in a new window being opened and returns a `BrowserWindow` object that represents the new window. As you might have guessed, `BrowserWindow` also derives from `ElementsContainer`. The cool thing is that Test Automation Essentials manages the current context for you, so you don't have to worry about it. Whenever you want to use an element that you found on any window or frame, you should be able to do that, even if it's not on the active context (though given that this element still exists of course). Test Automation Essentials will take care of switching for that context for you. Listing C-5 shows how the previous example would look with Test Automation Essentials.

Listing C-5. Working with Frames using Test Automation Essentials

```
// create an instance of Browser that wraps our webDriver object
var browser = new Browser(webDriver, "Site with iframe");

// Navigate to a website that contains an iframe
browser.Url = @"http://www.some-site.com/outer.html";

// find an element on the outer page
var outerButton = browser.WaitForElement(By.Id("outerButton"), "Outer
button");

// Find the inner frame
var frame = browser.GetFrame("frame1", "inner frame");

// find an element on the inner frame
var innerButton = frame.WaitForElement(By.Id("innerButton"), "Inner
button");
// clicking the inner button works normally
innerButton.Click();

// Clicking the outer button now works seamlessly!
outerButton.Click();
```

Contributing to the Project and Porting to Other Languages

The nice thing about open source projects is that anyone can help improve it and influence its direction. Because of the nature of this project, I mostly added to it things that *I* needed, and therefore there could be many things that should have been right there but are not. Likewise, while this project is very modular, it doesn't have a very clear boundary for what should be included in it and what shouldn't. So, any good idea that can benefit others can be included in it. Obviously, like any other software, this project can also have bugs that need to be fixed. Therefore, you can contribute to this project by many different means.

It is always advisable that before you go on and send a pull-request (see Chapter 13), you discuss it first by posting an issue though GitHub, or by contacting me directly. Note that posting an issue is not necessarily a bug report, as it can also be an idea for improvement or a request for a new feature. If you do send a pull-request, please make sure that it has clear XML comments for its public API; and at least for the Common and MSTest libraries, I expect nearly everything to be covered by unit or integration tests.

I also consider porting some of the stuff to other programming languages, mainly Java, but maybe also Python or any other language that I see a need for. I'll probably do it when I'll have a real and significant test automation project that needs it, but you're welcome to add it too. Note that some things, especially in the Common library are relevant specifically to C#, though other utilities may be useful for other languages.

APPENDIX D

Tips and Practices for Programmer's Productivity

As a consultant who has worked with many test automation developers, I found that many of them are thirsty for tips and practices for working more effectively and for improving their programming skills. In this appendix, I gathered some of these tips and practices that I believe will be valuable to most people.

Some of these tips are relevant to all programmers, not just test automation developers, and some are more specific to test automation and even to Selenium.

Prefer Using the Keyboard

If you want to work more efficiently when working with code, I suggest that you get used to using the keyboard, much more than the mouse. It takes some time to get used to it, but once you do, you'll be much more productive when writing and reading code! Here are some tips that can help you get used to it. Most of these tips assume that you're using Microsoft Windows. If you're using Mac, Linux, or any other operating system, there are probably equivalent shortcuts, but they may be different. Searching the web for equivalent shortcuts in these operating systems will likely yield the desired results.

1. As it's difficult to change habits, you must put some constraints on yourself at first in order to get used to use the keyboard instead of the mouse. Therefore, I suggest that you deliberately put the mouse farther away from you. Only grab it when you don't find how to achieve your task with the keyboard, and then put it back away.

A. Axelrod, *Complete Guide to Test Automation*, https://doi.org/10.1007/978-1-4842-3832-5

2. Pressing the **Alt** key, highlights shortcuts for menus and buttons, by adding an underline under the letter of shortcut. To activate the shortcut, press **Alt** + the highlighted letter. For example, to open the **File** menu in virtually all applications, press **Alt+F.**

3. After the menu is open, you'll see the shortcuts of the menu items. When the menu is open, you can simply press the highlighted letter, even without pressing **Alt**. For example, after pressing **Alt+F** to open the **File** menu, press **O** to activate the **Open...** menu item.

4. Once the menu is open, you can navigate the submenus using the arrows. Press **Enter** to select the menu item. The arrow keys are useful also for navigating most tree controls. For example, to expand a folder in File Explorer, you can press the right arrow.

5. Many menu items also have additional shortcut keys displayed next to them, usually with the **Ctrl** key + some other key. These shortcuts can be invoked directly, without first opening the menu. For example, to open a new file, directly press **Ctrl+O**.

6. Most UI controls (elements) in most applications are capable of being "focused," which means that they're the first control to receive the keyboard input. There's only one focused control at any given moment. The concept of "focus" is most noticeable with textboxes, but other controls, like buttons, checkboxes, drop-down lists, etc., can also have the focus, allowing you to interact with them using the keyboard. Use the **Tab** key to move the focus between elements on a window. Use **Shift+Tab** to move in the opposite order. Press **Enter** or the **Space Bar** to press the focused button. Use the **Space Bar** also to toggle checkboxes, or select items in a multiselect list-boxes. Press **Esc** to click the "**Cancel**" button on dialogs.

7. Most keyboards have a **Context Menu** key (typically located between the right **Alt** and **Ctrl** keys), which is similar to a right-click with the mouse. Note though that when you use the mouse, right-clicking opens the context menu that corresponds to the location where the *mouse cursor* is, while pressing the Context Menu key on the keyboard opens the context menu where the keyboard focus currently is.

8. Use **Alt+Tab** and **Alt+Shift+Tab** to switch between open windows back and forth. In many applications, including Visual Studio, you can use **Ctrl+Tab** and **Ctrl+Shift+Tab** to switch between open documents. Keep the **Alt** (or **Ctrl**) key pressed to see the list of the open applications or documents.
9. In a text editor or textbox, use **Ctrl+→** and **Ctrl+←** to move the caret (the text entry cursor) whole words at once. Use the **Home** and **End** keys to move to the beginning and end of a line. Use **Ctrl+Home** and **Ctrl+End** to move to the beginning and end of the document.
10. Use **Shift+→**, **Shift+←**, or **Shift** + any of the navigation combinations described above to select the text between its current location and the location where the navigation keys will take you to. For example, **Shift+Ctrl+→** will select the next word.
11. Press **Shift+Del** and **Shift+Backspace** to delete the next or previous word (or from the middle of a word to its end or its beginning).
12. Search the web for a list of shortcuts for your favorite IDE or other applications that you work with often. Print these and put it in from of you. Also search the web for specific keyboard shortcuts for actions that you can't find the shortcut to. If you're using Resharper, I highly recommend you to search for "Resharper Default Keymap PDF" on the web, print it, and put it near your keyboard. From time to time examine this document to find useful shortcuts and try to use them often until you start using them naturally. You'll also likely learn addition features of Resharper (or of your IDE) when you examine this list.
13. The keyboard mappings for many applications, especially programming IDEs are customizable. So, you can assign direct keyboard shortcuts to actions that you use often and that don't have ones by default.

Poka-Yoke

In Chapter 11 I declared `MVCForum` as a read-only property and not as a regular property (with public setter). Declaring a property as read-only is more restrictive than declaring it as read/write, and therefore may seem like an inferior option. Well the thing is that I don't want that anyone to ever change this property from outside of the class. (Note that changing the value of a reference-type, means replacing the reference with a reference to another object, and not making changes to the existing object.) Similarly, there are many other design decisions that a programmer can make to restrict or allow the usage of some code construct, and some programmers mistakenly think that the more you allow – the better. So why should I prefer to restrict? To answer that question, I'll tell you what "poka-yoke" is and why it's so important.

Poka-yoke is a Japanese term that means "mistake-proofing" or "inadvertent error prevention." The term is part of the Lean manufacturing philosophy developed by Toyota in the middle of the 20th century. A poka-yoke is any mechanism that helps an equipment operator to avoid (*yokeru*) mistakes (*poka*), as it turns out that preventing mistakes is often much easier and cheaper than inspecting and fixing defects later on. While originally the term and the idea were used in industrial manufacturing, it holds very true for software design as well.

There are countless examples of language features and techniques in software design that enable poka-yoke. In fact, the first principle of object-oriented design – encapsulation – is one. Encapsulation is what allows you to specify whether a member of a class will be **private**, so only other methods of the same class would be able to access, or **public** if the member is planned to be used by methods in other classes too. Using strong typing (i.e., having to specify the exact type explicitly) is a very strong Poka-yoke mechanism in the languages that support it (e.g., C# and Java). In addition to this, using strongly typed collections that take advantage of the **Generics** language feature are more error preventing than collections of objects. Another technique to avoid mistakes is to avoid using `null` as a valid value, as I'll explain shortly.

While using poka-yoke sometimes limits flexibility, I strongly prefer to allow flexibility only in places and manners that I explicitly choose, and not by default. This is the reason I preferred to make the `MVCForum` property read-only.

Avoid Nulls

Most object-oriented languages allow object variables (be it local, field, static, etc.) to have a special value of `null`, indicating that it references no actual object. This is usually also the default value of an uninitialized object variable. However, probably the most common *unexpected* exceptions (which actually represent a problem in the code, or better put: bugs) is `NullReferenceException` in .Net (and `NullPointerException` in Java), which is caused by the use of `null` values. Especially if you treat nulls as valid values, these bugs can be difficult to investigate, because their cause is often in a different place from where the exception itself is thrown. Not only that, but the cause is that the assignment of a real object *didn't happen!* Investigating why something didn't happen is much harder than investigating why something did happen… Note that if you avoid using nulls as a general rule, but you get a `NullReferenceException` nevertheless, then it's usually very easy to find the variable that you forgot to initialize.

For these reasons, many modern languages (especially *functional* languages, like F# and Scala) intentionally prevent the use of `nulls` by default. In some of these languages, you can still use nulls, but only if you explicitly declare the variable to allow it.

Because most of us are still using "regular" object-oriented languages (C#, Java, Python, Ruby, etc.), the compiler doesn't hold us from using nulls. However, with a little self-discipline, we can avoid it ourselves. Simply initialize every variable right when it's declared (or in the constructor), and avoid assigning `null` or returning a `null` from a method. If you get a value from "outside" that may be null, (somewhere that you don't have control over, like arguments in a public API, or a return value from a third-party method), then you should check it for null as soon as possible and handle it appropriately. This can be done by throwing an exception if you can't handle it, or "translating" it to a different object that represents an empty object (AKA the "null object pattern." A simple example is an empty list rather than null).

Adhering to these rules will help you avoid many potential bugs!

Avoid Catching Exceptions

Most beginner and intermediate developers use `try/catch` blocks much too extensively, because they think that this makes their code more robust and reliable. Well, it's true that when you catch an exception, you prevent it from propagating to your caller (and if your code is the "main" method or a test method, then you prevent the exception from crashing your application or failing the test), but that only gives the *illusion* that the code is more robust. The reason is that if something went wrong and without knowing exactly what and why, and you just ignore it, you'd very likely be hit by a ricochet of that problem later on. If you ignore all exceptions, the program will never crash, but it also may won't do whatever it's expected to do! Catching an exception just in order to ignore it (i.e., using an empty `catch` clause) is often called "swallowing exceptions," and in 99% of the cases, it's a very bad idea…

Moreover, if you swallow an exception, it will make your life much harder when you come to debug or diagnose a problem. This is even worse if it happens only occasionally in production! Therefore, the most naïve approach to this problem is simply to write the exception to a log file. This is significantly better than swallowing the exception completely, and in some cases it's a good idea, but in general, as far as the user (either the end user or the caller of a method) is concerned, this has the same effect of swallowing the exception, because you normally won't look at the log until a user experiences and reports a problem. If you decide to write an exception to the log, make sure to log all of the information about the exception, including the stack-trace and original exception if it's different from the one that got caught (`InnerException` in .Net or `getCause()` in Java). In .Net, it's best to use the `ToString()` method because the string it returns already contains all of the relevant information.

So how *should* you handle exceptions? In most cases, the correct answer is "you shouldn't"! For exceptions that you expect to happen at times (e.g., "file not found" when the application tries to read from a file that the user might have deleted or renamed), you better avoid the condition by checking for it in advance and handle it appropriately (e.g., check if the file exists, and if not, advise the user what to do). For general exceptions that you can't predict and you can't specify why they may happen, or in case you have nothing to do about it (like "out of memory"), let them propagate down the stack. If you let the program crash immediately (or let the test fail due to the exception), you automatically get most of the information that you need in order to investigate the root cause. If you do it properly, then you'd most likely find the fault and fix it quickly.

Clearly, if it was *always* such a bad idea, the `try/catch` construct wouldn't make it into all modern languages. So, when does it make sense to catch exceptions? There are a few cases:

- You're able to specify exactly what could go wrong, and handle it gracefully, but you can only do it after the fact. In many cases, runtime conditions may change during the operation (e.g., the file is deleted *while* you try to write to it). At other cases you do have a way to check for the condition before starting the operation, but it simply doesn't pay off, for example, due to performance (i.e., checking if you can do something that takes a long time may require that the check itself take the same time by itself). Other cases may be due to technical limitations (e.g., you don't have an API that can tell you in advance if the operation is about to succeed or fail).
- In these cases, you should catch the most specific type of exception, and narrow the scope of the `try` block to only the specific operation that you expect to throw that exception. This way you avoid catching exceptions that you don't expect and only catch those you do. The `catch` block should perform the specific handling that is expected to resolve the issue or advise the user or the caller of the method *how to resolve it*.
- Adding a global error handling that presents all unexpected failures to the user in a special way and/or attaches additional information to the exception. For example, some teams prefer to use a proprietary reporting mechanism for test results and want all the failure information to be written to it, along with the exact date and time. In addition, you may want to add some system information or specific information about the SUT.
- In this case you should have only one `try/catch` block, but in contrast to the previous case, in this case the `try` block should encompass the entire program (or test case) and you should catch *all* exceptions. In this case, make sure to report all the information you have about the exception to allow efficient investigation.

- You should also think about a fallback exception handling, in a case where an exception occurred *inside* your normal exception handling code. For example, if you fail to write to a custom report, you should still fall back to writing the failure to the console, or to the normal exception handling that the testing framework provides. Usually the handling should simply be throwing a new exception containing the information of both the original exception and the information about the secondary exception that happened inside the exception handler. This way you'll be able to investigate both the reason for the original failure, and also the problem in the exception handler. Obviously, the secondary exception handler should be much simpler than the first one, so the chances that *it* fails should be very small.
- You have valuable information to add to the exception that may be useful for the investigation. For example, you have a complex method that performs multiple operations on the database, calls external REST services, etc. In such a method a lot of things can go wrong, but the native exceptions that may be thrown may be too low level to be useful and won't tell you how to reproduce the error. In this case you can catch all exceptions (or a more specific but still general exception) and re-throw a new exception that wraps the original exception but adds information about the parameters that the method received, or some internal state of the current object.
- You should very rarely write such handlers in advance. In most cases, you should first let the exceptions handled normally, and only if you see that you're missing some important information that you need in order to find the fault, then you need to add the additional information to the exception for the sake of the next time. Needless to say, you should keep all the information about the original exception, especially its stack-trace.
- You perform an operation on a large number of rows (or other form of data that contain independent entities), and you don't want a failure in one row to stop the process and prevent it from doing the operation on other rows. In such case you should wrap the operation for a single row in a try block, and inside the catch block, add the

exception to a list. When the loop completes, you should throw an exception that contains all of the caught exceptions (.Net has an `AggregateException` class just for that), or report the failures to the user by other means.

One common use of exception handling is for retrying the failed operation. This is a valid reason to use try/catch, and it falls into the first category above. However, it's important to note that you understand exactly why and on what you retry. For example, if you try to read from a file and the file does not exist, retrying won't help without a human intervention! If you catch too broad exceptions and retry the entire operation, you create two new problems:

1. You're wasting a lot of time retrying something that has no chance to succeed.
2. You're very likely to miss other problems (including bugs in the SUT!) and lose valuable knowledge and information about the nature of these problems. As mentioned above, you'll have a harder time investigating the problem, and it may cause secondary problems that will just confuse you.

Occasionally I hear about test automation projects that introduced a mechanism that automatically retires a failing test. In my opinion this is a smell for a bad architecture, lack of isolation, or simply a poor test design. **Implementing an automatic retry of tests implies that you trust the tests less than you trust the SUT**, and you assume that most failures will be caused by the test (or its environment) and not by the SUT. An automatic test should be predictable and repeatable. If it fails, you should be able to investigate and fix the problem as fast as possible.

Until now we only discussed when to catch exceptions but haven't mentioned when to throw your own exception. This one is simpler – always throw an exception from methods if you detect a condition that prevents the method from doing whatever it was intended to do. When you do that, make sure to provide all the relevant information that can help whoever needs to understand what happened. Note that in object-oriented languages (that support exceptions), you should never return an error code or a Boolean to indicate success or failure. This is what exceptions are for!

In addition, avoid throwing an exception for the purpose of normal program flow. Exceptions should only be used to indicate some kind of problem! If in order to understand the normal program flow, one should follow the `throw` and `try/catch` constructs, then you're probably doing it wrong. The most obvious case that you should avoid is throwing an exception that you catch inside the same method.

Choosing the Most Appropriate Locator

The following tips are specific to Selenium. However, the main ideas and tips are relevant to all UI automation and may also be applicable for other cases in which the test needs to identify a piece of data from the application, like retrieving values from JSON, XML, or a database.

Selenium finds elements by matching a string value to a specified locator type. Selenium supports the following locators:

1. **Id** – identifies elements whose **id** attribute matches the specified value. According to the HTML standard, if this attribute exists, its value must be unique across all elements in the page. So, theoretically, if an element has an **id** attribute, it should ensure that we can uniquely identify it using this locator. However, this requirement is not being enforced by browsers and sometimes pages contain more than one element with the same **id**

2. **Name** – identifies elements using their **name** attribute. The name attribute should uniquely identify an element within an HTML form

3. **LinkText** – identifies elements by matching their inner text to the specified value

4. **PartialLinkText** – same as **LinkText**, but also matches elements in which their text *contains* the specified value, even if the value is only a substring of the link text

5. **TagName** – identifies elements whose tag name matches the specified value. Tag names are rarely unique, but in case they are, this locator can be used

6. **ClassName –** identifies the element using any of its class names. The `class` attribute of an HTML element can contain zero or more names of CSS classes, separated by a whitespace. This locator can take *only one* class name and only find element(s) that have this class

7. **XPath** – identifies elements according to the XPath specification.[1] The XPath specification provides a special query syntax to locate elements within an XML or HTML documents

8. **CSS Selector** – identifies elements according to the CSS selector pattern.[2] CSS Selectors were originally used to identify elements in a CSS document in order to apply a given style to them. However, the jQuery JavaScript library also makes use of this syntax to identify elements.

If you want to find a single element, you must use a locator and value that matches only that one particular element. If a locator matches more than one element, Selenium returns the first one, but this is not necessarily the one you want. Anyway, while the locator you use must identify the element uniquely, this is not enough: in order to keep the tests reliable and easy to maintain, we must also ensure that the **locator that we choose will always identify the relevant element, or at least which is least likely to be changed**.

Id and **Name** are considered the best locators if they exist, because they uniquely identify the elements (or at least *should*). This is indeed true in the majority of cases. However, some frameworks, or even proprietary JavaScript code, sometimes generate these ids at runtime, randomly or sequentially. While they ensure uniqueness, these values are most likely to change between runs if they're generated randomly. If they're generated sequentially, then they'll probably change only if elements are added or removed, either at runtime or due to a change in the code. So, my advice is to use an id or name locator only if its value is a *meaningful* name that is related to the business functionality of the element that you're looking for. In other words, use it only if its value was apparently chosen by a human. For example, if the id of an element is `submitButton`, then this makes a perfect locator. However, if the id of an element is `button342`, then this would probably mean that this is a *bad* locator!

[1] https://www.w3schools.com/xml/xpath_syntax.asp
[2] https://www.w3schools.com/cssref/css_selectors.asp

The same rule also applies to class names: if the class name is meaningful and is related to the element that you want to find, then it's great to use it. But if the class name has nothing to do with the `meaning` of the element, then avoid it if you have a better way.

While the **XPath** and **CSS Selector** locators have the most complex syntax, they provide the most flexibility. These locators allow you to specify a combination of characteristics of the element itself as well of its ancestors and even siblings, all as one string. Chrome can generate these strings for your and copy them to the clipboard, directly from the context menu of the element in the Develop Tools. You can then use these strings with your Selenium code to identify the element you want with it. While this sounds very compelling, and indeed many Selenium automation developers use it, it's often **not a good idea** whatsoever. Chrome attempts to find the best locator using a heuristic that searches for the simplest combination of characteristics that is also unique. However, this heuristic cannot predict what's likely to change in the future or even in the next time the page will be loaded and may even suggest a locator that is currently unique, but in another run won't be. Locators that are less likely to change are locators that contain strings that represent the real meaning of the element. For example, if a button has a `button-login` class name on it, even though class names need not be unique, its name suggests that it identifies a Login button. Therefore, you should learn the CSS Selector and XPath syntax and use your own judgment when combining the most appropriate pattern to use.

The last (but not least) tip is to limit the scope of the search to only a specific container. In other words, you should first identify a unique element that contains the element you're looking for, and then identify the element you're looking for inside of it. The famous `FindElement` method in Selenium works both on the `IWebDriver` object as well as on any `IWebElement` object, so you can use it to identify an element within another element. You can have as many levels of such nesting, and the typical approach is to use a Page Object for each such container that the end user can identify. The advantage of identifying elements only within their containers is that you only have to worry about the uniqueness of the elements within the container and not among all elements on the page, and in addition it removes the duplication from the locators of all the elements that reside in the same container. While this duplication is usually only of a substring (e.g., the start of an XPath expression), if it will need to change it will need to be changed for all of the elements inside this container.

Hard-Coded Strings in Test Automation: Yea or Nay?

Many developers treat hard-coded strings as a bad practice. There are some good reasons for it but also some misconceptions around it. In fact, in most cases you cannot avoid completely the use of string literals (AKA string constants, or hard-coded strings) in the code. For example, even if you want to keep these strings in an external file, you'd probably need to have the name of the file somewhere in your code. But then again, one can keep this limited set of constants in one source file, in order to keep it separated from the actual logic of the application. However, it is important to understand why and when it is a bad practice to use string literals mangling inside the source code, and when it is acceptable or even a good thing.

There are several reasons why hard-coded strings are considered bad practice:

1. Separation of concerns – strings are mostly used for some kind of human interaction, while the algorithm itself is not. The wording or spelling of a string that should be displayed to the user may need to change separately from the algorithm.

2. Localization (Internationalization) – applications that are localizable (i.e., applications that are designed to support multiple languages and cultures), or even may need to be localizable in the future should extract any string that the user may be exposed to, to some external resource that can be easily switched (either at runtime or at compile time).

3. Configuration – in many cases, global string values that are not intended for user interaction (at least not directly), tend to have different values for different environments or configurations. A database connection string is a good example. Hard-coding these strings make the code inflexible and won't be able to run on different environments.

4. Duplication – if you need to use the same string in multiple places in the code and you don't put it in a common place, then you introduce a duplication. If at any point you need to change this string, you would now need to change it in all places.

5. Length – sometimes strings can get quite long. and if they're mingled with the algorithm, they hinder the readability of the code.

Those reasons take some developers to the conclusion that *all* strings literals should be extracted from the code into a separate file. The most common claim I hear for that, is that it allows them to change these values without re-compiling them. However, re-compiling is an issue only for an end user who doesn't have access to the source code and development environment or is not proficient in the programming language and is afraid to change something that he doesn't understand.

Let's consider three common usages for string literals in test automation, and let's analyze where it makes the most sense to put them:

1. Environment information – Depending on your isolation strategy (see Chapter 7), some of this information may be different for every environment that you want to support running the automation on, though some values may be fixed for all environments (or there's only one environment you're supporting). Even though re-compiling is not a big deal in test automation, you'd probably want to keep the information that varies from one environment to another in a configuration file, or at least in a separate source file. This will help you manage their changes separately from the source files in your source-control system. As a rule of thumb, these configuration files should be kept small with very few, easy-to-understand values; otherwise it will soon become a nightmare to maintain by itself.

   The "fixed" values however are anything that is common to all environments and should not change for a predictable reason in the foreseeable future. It doesn't mean that it *can't* change in the future, but for now there's no reason to assume that it will happen soon – for example, a URL of a third-party service provider. In my opinion, it's fine that such data will be kept directly in the code, as long as it appears only at a single place and close to its usage, for example, as a named constant. If this URL will change at some point in the future, it shouldn't be difficult to find the place in the code where it's defined and change it. By contrast, if you put that value in a configuration file and you need to change it after couple of years, you probably won't remember that it's there anyway, and you'll have to debug or analyze the code in order to find where the value comes from in order to know where to change.

Therefore, if the value will be read from a configuration file, it will be less obvious to find it than if it is directly in the code. If you have multiple instances of this configuration file (e.g., for different environments) hanging around, probably not under source control, then even when you know what to change, you still need do it for every instance of this configuration file. After all, because you can never know what would change and when, you'd probably have lots and lots of such configuration values that will just be more difficult to manage and maintain. Bottom line: keep things simple!

2. Locator values (Ids, XPath, CSS Selectors, etc.) – Whether you're doing UI automation (e.g., using Selenium) or REST API tests (or maybe some other kinds of tests), you need to refer to elements on the screen or in the response message. In order to do this, you need to specify the id, name, XPath, etc., of these elements. Some people think that putting these strings in an external configuration file makes their code more maintainable. Frankly, I may be able to see how it makes the "code" more maintainable, but the maintenance of these files often becomes a headache in and on itself. If more than one automation developer works on the project, such a big file that contains all of these elements soon becomes a merge hell. In fact, such files don't solve anything, but rather move the maintainability problem from one place to another, which is less appropriate for it. Keeping the locators with their string literals close to their usages is much easier to maintain than having all of the locators defined in one place while their usages are spread across the entire code base. Encapsulating the details inside the class or the method that use it is the solution to make your code more maintainable, and the Page Object pattern is the most common way to encapsulate the locators inside a class.

3. Input values – Tests typically have to type in or otherwise pass strings as inputs to the SUT (like entering the discussion heading in Chapter 11). Some people like to keep these values in a separate file to give them flexibility over the values of these strings. If you're creating a data-driven test, then that's fine. But if you don't intend to run the same test with different values, which you specifically define to verify different cases, then there's probably not a good reason to extract it from the code. The need to extract such values to an external file is often a symptom for a design that depends on specific data that exists in the database. So, if at some point the data will change, then you won't have to change the code, only the external file. But again, this is just moving the maintenance problem from one place to another. This file becomes a contention point and will be harder to maintain than if the test used its own data.

4. Expected results – Similar to input values, the expected results are values that we expect as outputs from the SUT. The exact same considerations that apply to the input values apply also to the expected results.

5. SQL statements or JavaScript snippets – Sometimes you need to invoke an SQL statements, a JavaScript command, or something along these lines from the test code. If these scripts are long, putting them in separate files will indeed improve the maintainability of the code. However, for sparse one-liner statements, embedding them directly in the code, close to their usage, and not duplicated, improves encapsulation and makes the code easier to understand and to follow.

All in all, in the majority of cases, strings used in test automation code better be in the code, as long as they're not duplicated and appear close to their usage in the code, rather than in an external file. Use external configuration files only for values that *have to be different* for different environments.

Index