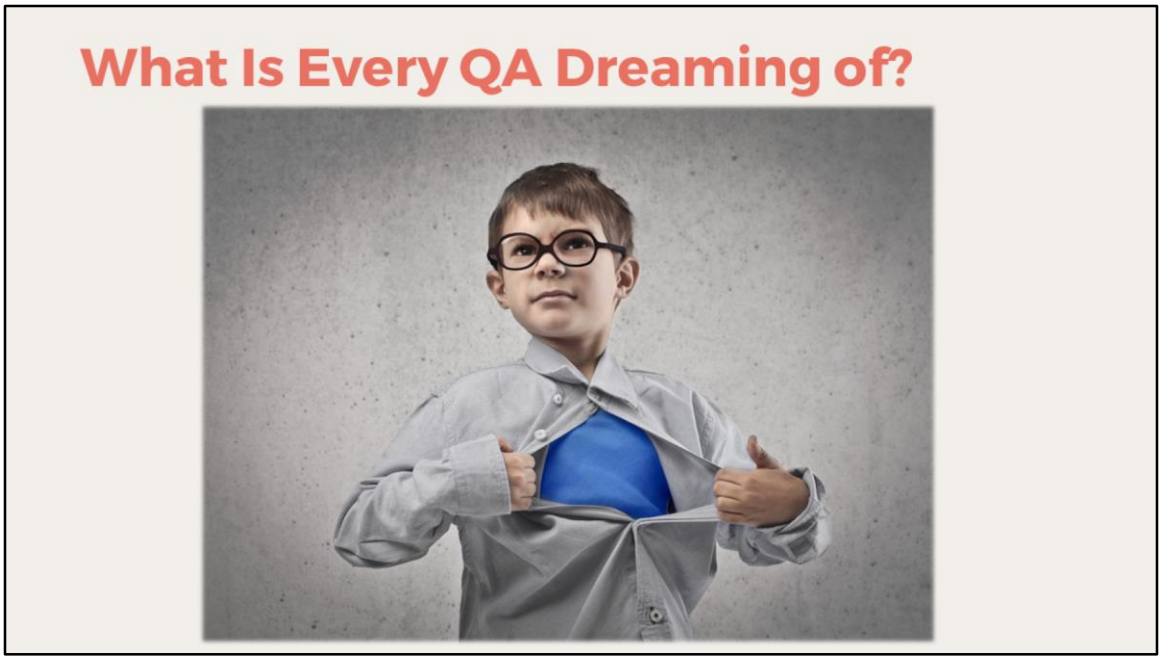
Today, I’m going to talk to you about my dreams and how they came true.



What is every QA dreaming of? I can tell you what my dream was and still is... Imagine a world where you go to work, open your email, and see a tests' notification. A green one. All tests have passed the nightly execution. **You want to boost your ego even further, so you open the tests execution statistics**, and observe that your tests are twice more than half a year ago, but are executing almost at the same speed. **You are euphoric, because now during the next few weeks**, **you don't have to spend your time maintaining these tests**. **You have spare time to write automation** about the new features even before they are ready. Who doesn't want this? I'm sure I want it, and I have it.

This presentation is all about that. Boosting the quality of your tests through proven design patterns and principles.

I am going to present you the three design patterns that helped me to accomplish my dream.

The Lecturer

- Anton Angelov
- Quality Assurance Architect
- Telerik, a Progress Company
- Site: http://automatetheplanet.com
- @angelovstanton

SEETEST 2015 SOUTH EAST EUROPEAN SOFTWARE TESTING CONFERENCE SOFIA

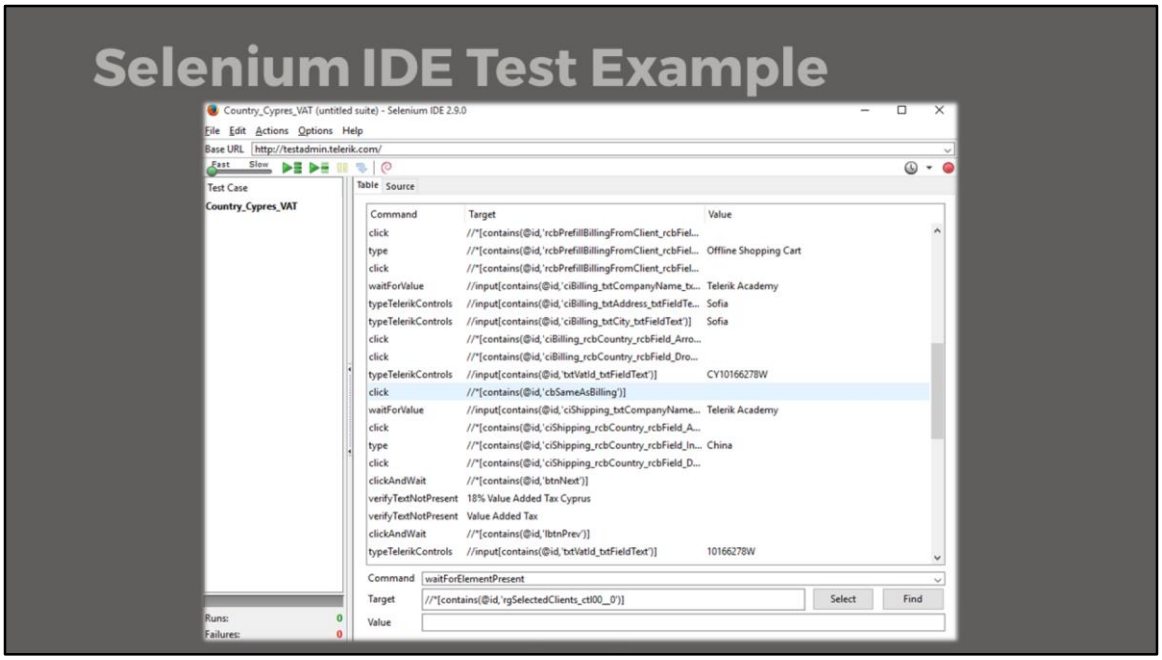
Telerik A PROGRESS COMPANY

CODE PROJECT For those who code

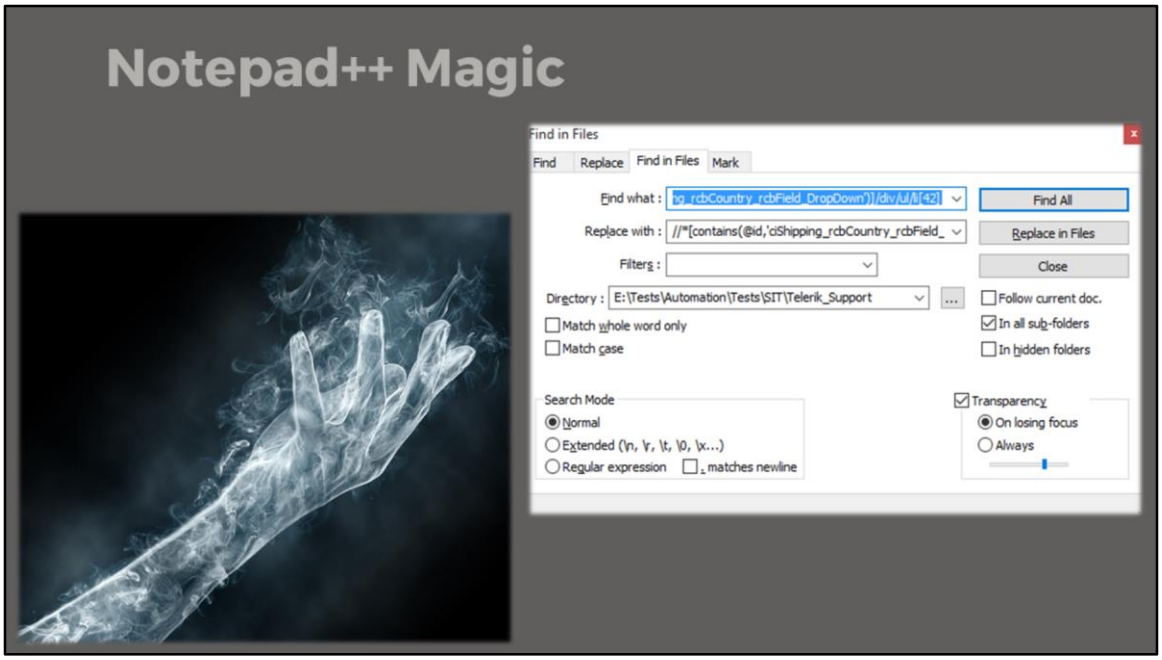
DZone MVB Most Valuable Blogger

I'm Anton Angelov, a Quality Assurance Architect in Telerik. And I love sharing. Because, in my opinion, sharing is what is boosting our sector so much in front all others. I'm a proud owner of automatetheplanet.com where I share all my ideas about code and tests. As a shareaholic, I'm a most valuable Blogger at **DZone** and a platinum author at **Code Project**.
Today I will tell you a story. It is the story of how my teammates and I succeeded in accomplishing my dream.

Unfortunately, as in every fairy tale, it didn’t happen from the first time. There were a lot of obstacles down the road, like not maintainable and not reliable tests. Our code reuse was remarkably poor, and the tests weren’t so fast at all.



In order to believe me, I'm going to show you our first tests. We needed to automate our shopping cart process, and at that time, we were using Selenium IDE. Those tests were **executed only on a nightly basis through** Selenium Remote Control.
There was something totally wrong with those tests, **and it was that, at first, they were using hard-coded locators** (like full-length HTML IDs). As a result, **two to three times a week I had to spend a couple of hours to fix those.**

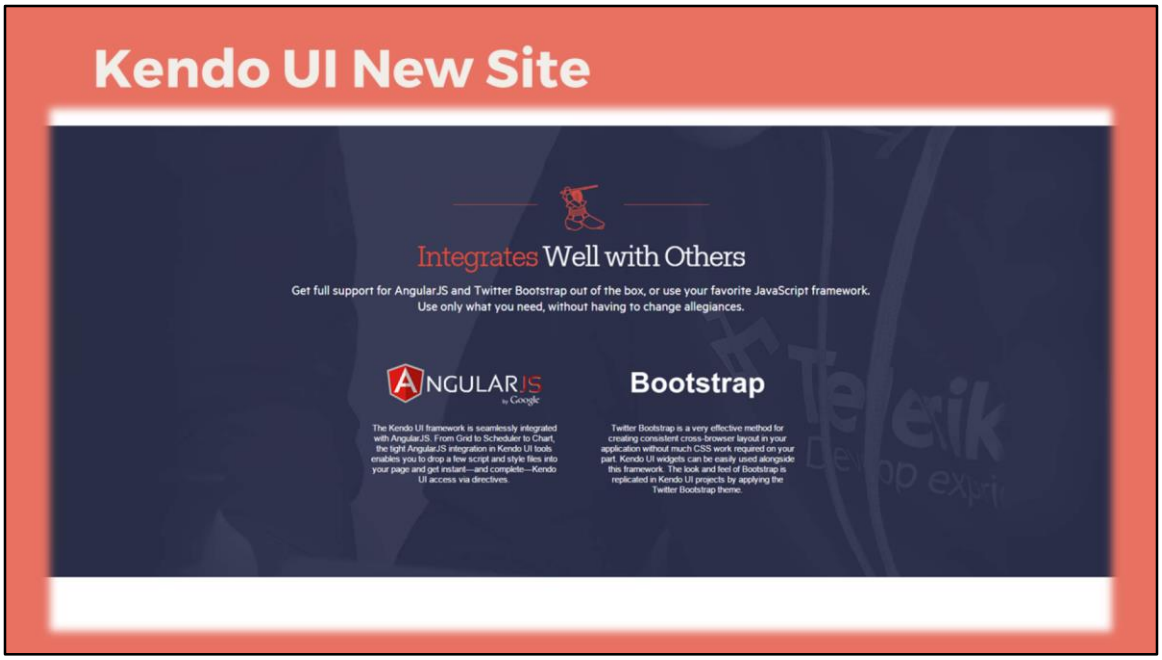


For the job, I used my favorite tool at the time; you guessed it right- **Notepad++**. It can replace a word within multiple files.

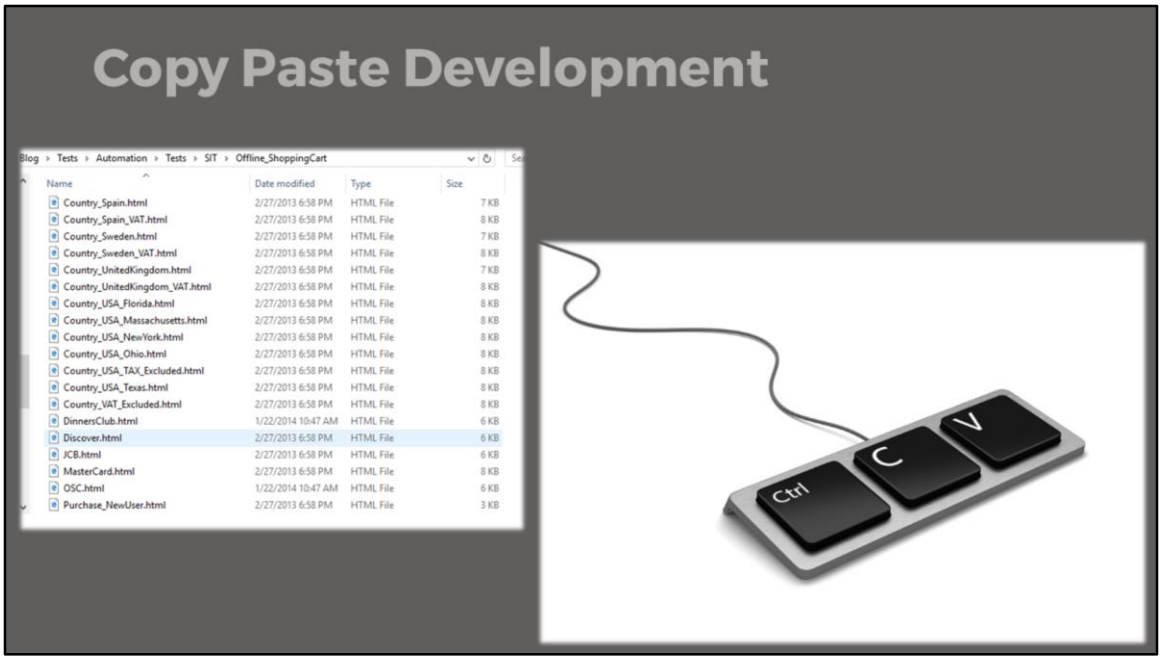
Selenium IDE Custom Locator

```
PageBot.prototype.locateElementByPartialId = function(text, inDocument) {
    return this.locateElementByXPath("//*[contains(./@id, 'Z')][1]"
.replace(/Z/,text), inDocument);
};
```

We partially resolved the locators' problem with the introduction of a new custom locator- 'partialId', written in pure JavaScript. We used it as a Selenium IDE extension. With the new custom locator, everything was so great that I almost forgot about the Notepad++ magic.



However, our company decided to create a new product, called Kendo UI. We got the task to create a brand new site with its dedicated shopping cart. Guess what? The shopping cart was identical to our existing one, but the locators of the pages' elements were different.
So how do you reuse Selenium IDE tests?



Simple. Copy them, of course! And change the locators with a little bit of Notepad++ magic. This was only one of many mistakes that we made, in full discrepancy with one of the main superb programming principles DRY- 'Do Not Repeat Yourself'.

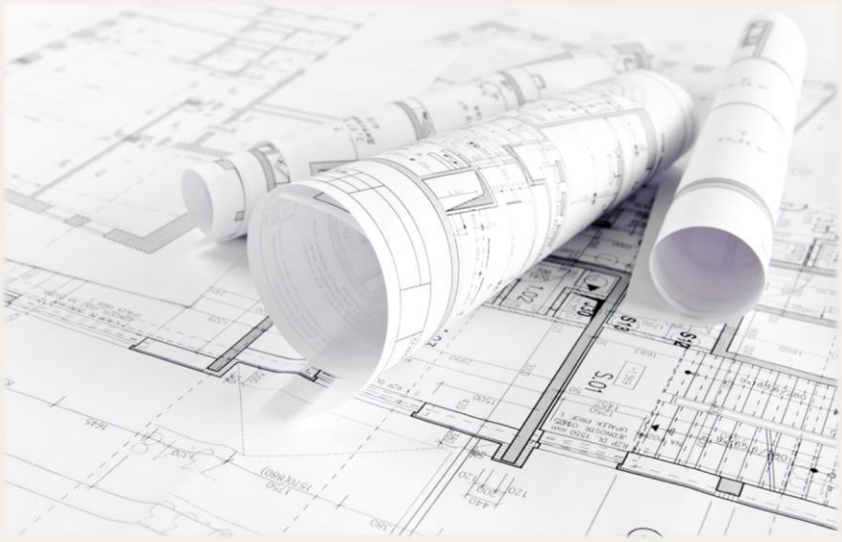
Don't Repeat Yourself Principle(DRY)

Of all the principles of programming, Don't Repeat Yourself (DRY) is perhaps one of the most fundamental. The principle was formulated by Andy Hunt and Dave Thomas in the book '*The Pragmatic Programmer'*. The developer who learns to recognize duplication, can produce much cleaner code than the one who uses the copy/paste development a lot. **Duplication results in increased probability of bugs and adds complexity to the system**. Moreover, **duplication makes the system more difficult to understand and maintain**.

The Rescue Team

Apparently, we needed help. Two guys from the so-called, ‘Tooling Team’, joined ours for a while to aid our automation efforts. **By the way, this team, at that point of time, was part of a bigger group of teams, working on our largest product.** Because of that they had and still have a dedicated team responsible for maintaining their automation tools and frameworks. We had the best on our side.

The New Project

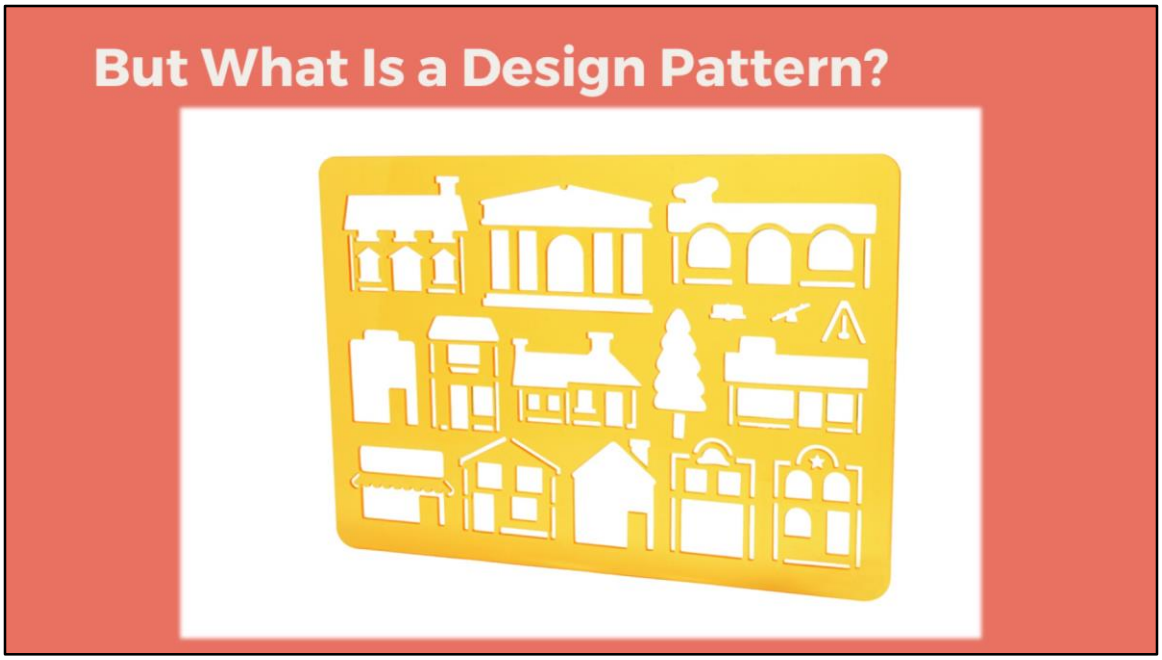


For just two months, they couldn't do miracles, but after that period, we had a fully working C# project running Test Studio Framework tests. For those who **haven't heard of it, it is the thing that is powering the Telerik Test Studio IDE**. It is similar to WebDriver but much more powerful, because it provides a lot of handy goodies, like the capability to automate WPF controls, plus many other things that you have to write manually in **Selenium**.

First Design Pattern Learned

```
public class SupportFacade
{
    public void SelectSupportProduct(string name)
    {...}

    public int GetTicketId )
    {...}

    public int GetTicketFirstIdYourSupportTicketsGrid()
    {...}

    public string SubmitSupportThreadContent(TextGeneratorFacade textGenerator)
    {...}

    public string FillSupportThreadSubject(TextGeneratorFacade textGenerator)
    {...}

    public void FillSupportThreadTitle(TextGeneratorFacade textGenerator)
    {...}

    public void AttachFileSupportTicket(bool shouldAttachFile, string filePath, string currentAttach
    {...}
```

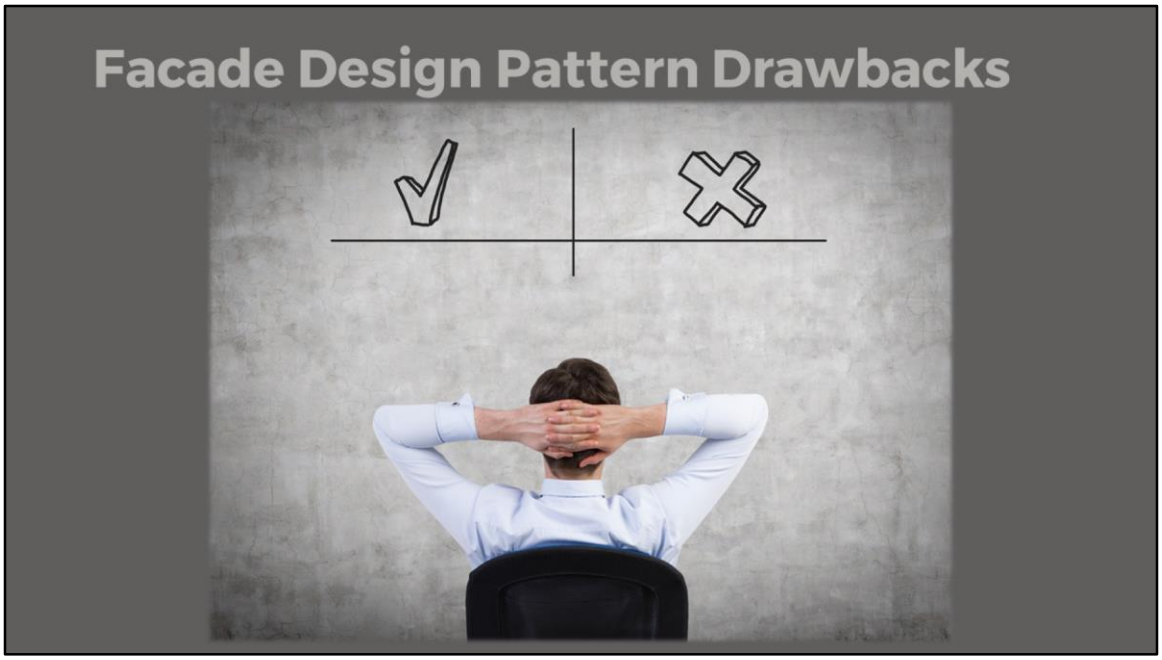
One of the most prominent parts of the project was that, for every functional area of the site, there was a dedicated class called Façade, so that it can be easily reused amongst the UI tests.
The Façade Design Pattern was the first pattern that we used in our tests. **All pages' elements and framework's stuff were hidden there.**
I am going to tell more about it just in a bit.
Here I copied one of our **old facades that had lots of code utilized in our support system's automation**. There you can find the logic for getting a submitted ticket by id, filling ticket subject, attaching a file to a ticket and so on.



But what is a design pattern? Before proceeding with the definition of the Façade Design Pattern, I am going to tell you exactly what a design pattern is.
It is not a code snippet. It is not an algorithm. It is not a component or library. Each pattern describes a problem, which occurs over and over again in our code**. It then describes how to solve this problem** so you do not need to reinvent the wheel every time. You can think of it as a template for solving particular design problems.

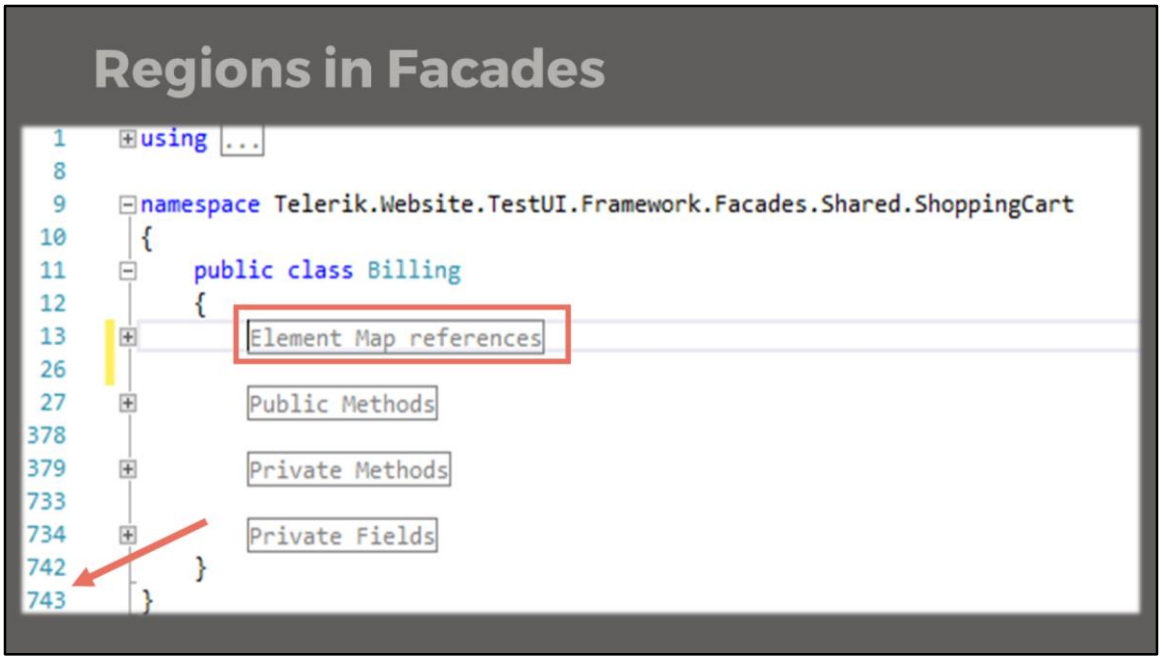
1. Facade Design Pattern

A facade is an object **that provides a simplified interface to a larger body of code, such as a class library**. It makes a software library easier to use and understand, is more readable, and reduces dependencies on external or other code.



It doesn't have any real drawbacks, as it provides a unified interface to a set of interfaces in a subsystem. However, the biggest problem for us was **the size of the façade files**. They got enormous, like thousands of lines of code.

Regions in Facades



In the presented example over 700 lines.
We had to use a special language feature to solve the problem. This feature creates regions inside the files so that you can collapse or expand the code in the editor.

We used another design pattern for accessing our facades and other logics.
The Fluent Interface is an implementation of an **object-oriented API that aims to provide more readable code**. It reduces syntactical noise and expresses the meaning of the code more clearly. It is implemented by using method cascading (aka. method chaining).

Fluent Interface Code Example

```
public void VerifyEmptyCreditCard()
{
    BAT.OpenTelerik().LoadFacades().CreateShoppingCartFacade().InitializeMainShoppingCart()
                                                              .ProcessOrderConfirmation().VerifyInvalidCardMessage();
    BAT.OpenTelerik().LoadFacades().CreateShoppingCartFacade().InitializeMainShoppingCart()
                                                              .ProcessOrderConfirmation().VerifyEmptySecurityCodeMessage();
    BAT.OpenTelerik().LoadFacades().CreateShoppingCartFacade().InitializeMainShoppingCart()
                                                              .ProcessOrderConfirmation().VerifyEmptyMonthValidationMessage();
    BAT.OpenTelerik().LoadFacades().CreateShoppingCartFacade().InitializeMainShoppingCart()
                                                              .ProcessOrderConfirmation().VerifyEmptyYearValidationMessage();
}
```

“Method chaining/cascading” is a common technique where each method returns an object, and all these methods can be chained together to form a single statement. **The presented code was used to assert credit card validation messages.** I really liked it at the beginning. As a result, in the next year, we utilized it to the maximum extent. It was easy for the new people joining our team to write new tests. However, it was hard for them to orient themselves in the complicated large amounts of files that were supporting our Fluent Interface.

Fluent Constants (1)

```
public class BillingInfo
   {
       public string FirstName
       {
           get
           {
               return "FirstNameB";
           }
       }

       public string LastName
       {
           get
           {
               return "LastNameB";
           }
       }

       public string PhoneNumber
       {
           get
           {
               return "359894644746";
           }
       }
```

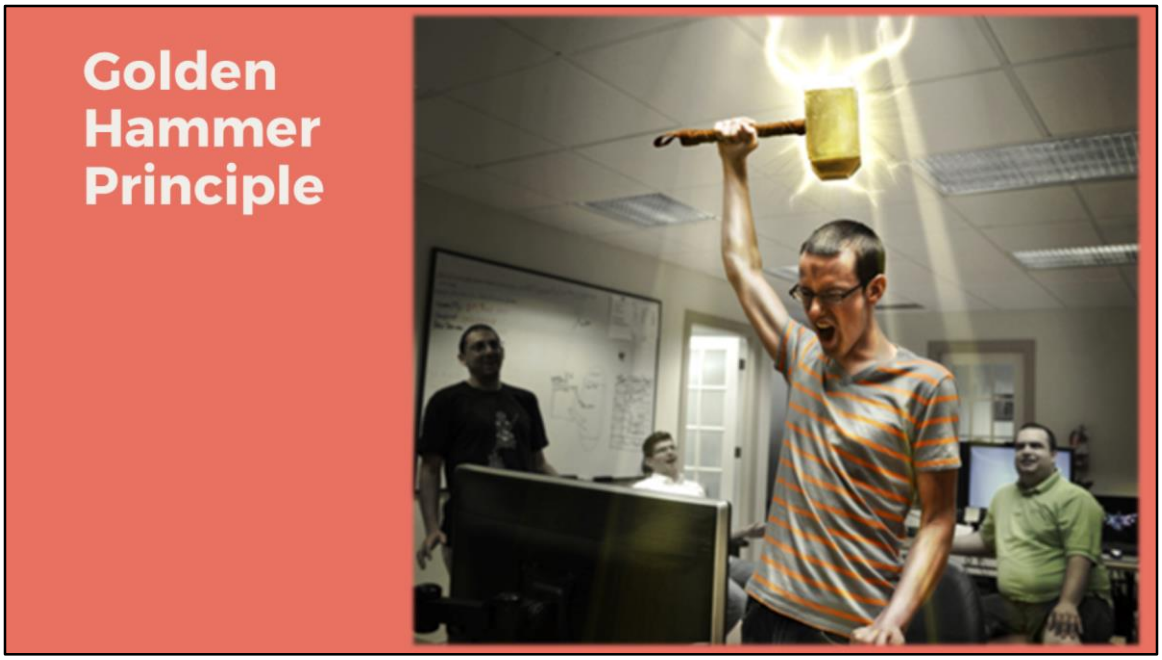
Another drawback of how we implemented it was that we used it for everything. I mean everything- even for getting constants.

In the example you can see the default values used in the filling of the client's billing information.

Fluent Constants (2)

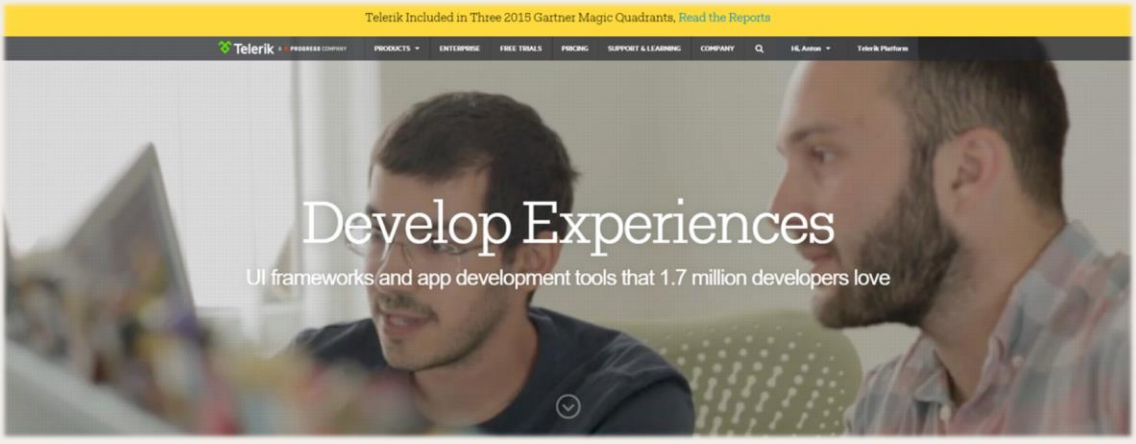
```
public void FillShippingInformation(Invoice xmlInvoice, string country)
{
    this.FillShippingInformationInternal(xmlInvoice,
BAT.Info.Payment.ShippingInfo.FirstName, BAT.Info.Payment.ShippingInfo.LastName,
BAT.Info.Payment.ShippingInfo.Company, BAT.Info.Payment.ShippingInfo.Address,
BAT.Info.Payment.ShippingInfo.City, country, String.Empty, String.Empty);
}

public void FillShippingInformation(Invoice xmlInvoice, string country, string state, string zip)
{
    this.FillShippingInformationInternal(xmlInvoice,
BAT.Info.Payment.ShippingInfo.FirstName, BAT.Info.Payment.ShippingInfo.LastName,
BAT.Info.Payment.ShippingInfo.Company,BAT.Info.Payment.ShippingInfo.Address,
BAT.Info.Payment.ShippingInfo.City, country, state, zip);
}
```

All of this led to one big chunk of unreadable code. As the code tells, **we are passing multiple default values to the base method for filling shipping information**. These values are accessed through the main Fluent interface. **Now it is time for your newest design principle.**

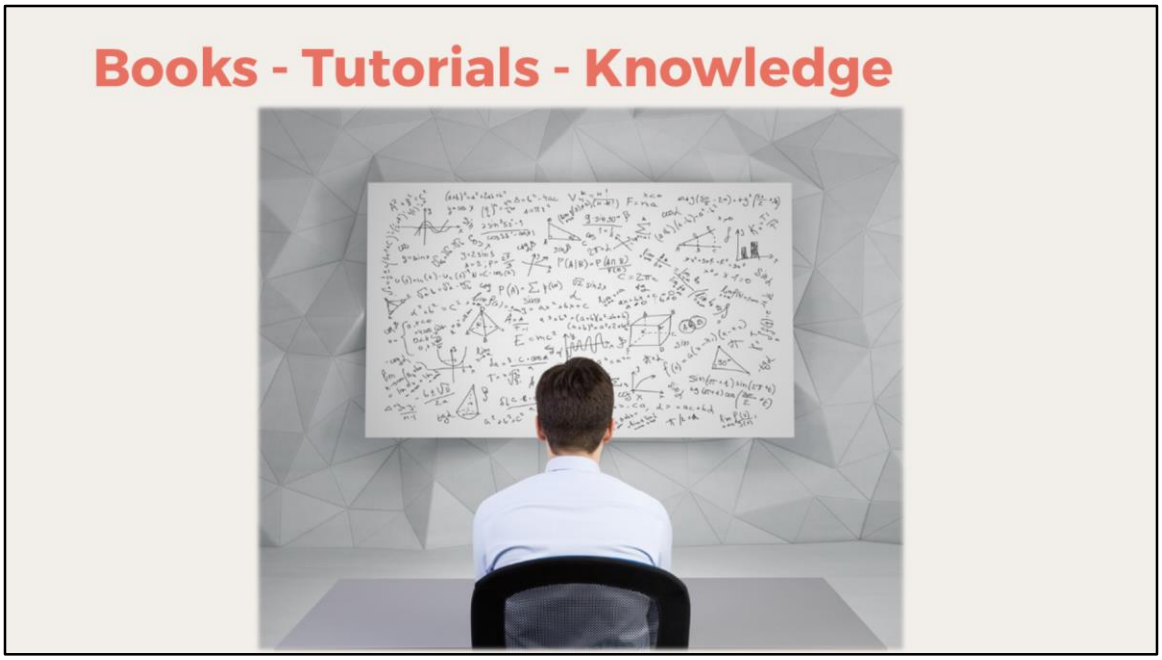


The Golden Hammer is a language or a tool that the developers feel comfortable with. Because of that they want to use it for everything. As the saying goes, “If all you have is a hammer, everything looks like a nail.” Our golden hammer was the Fluent Interface.

New Master Site Telerik



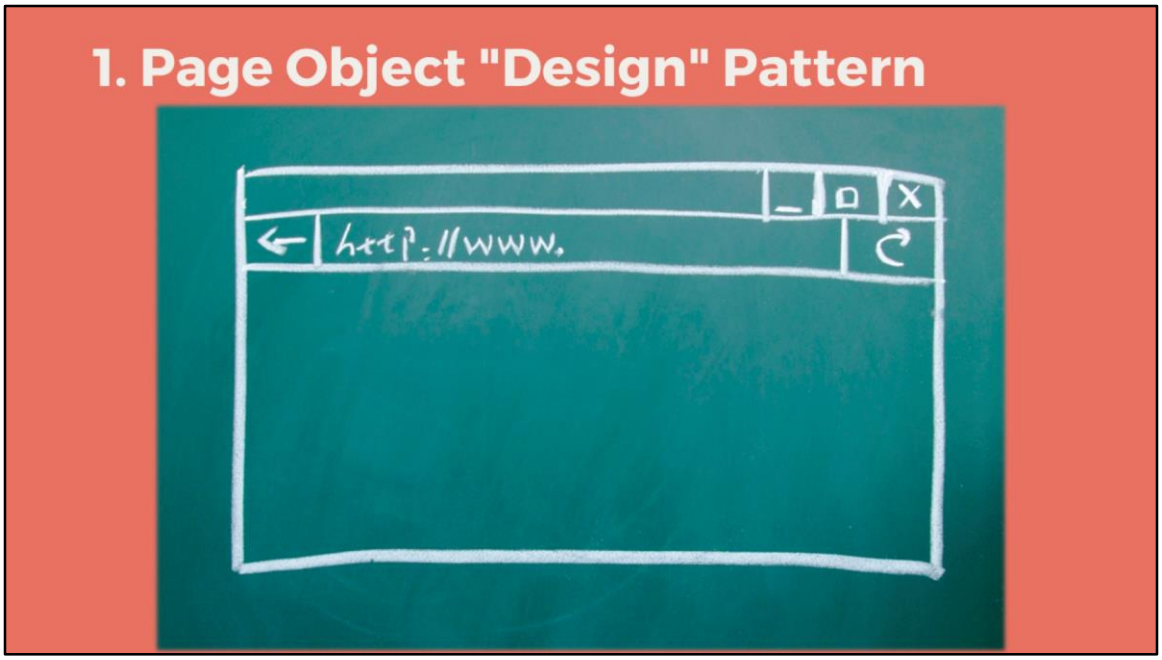
Half a year later, there was a huge rebranding of all Telerik products. Our team had the job to create one master site for all of them. This was one of the biggest projects of which I have been part. **We had to integrate several systems into a single one and further rewrite most of our existing logic to be service oriented.**



During the whole process, we read a lot of books and watched a lot of tutorials of C#, design patterns, and automation frameworks.

Elders Council

With so much new knowledge, we got together **with the rest of the most senior quality assurance guys to determine the future of our test framework**. The time looked like an excellent opportunity to rewrite our test framework and to start fresh.

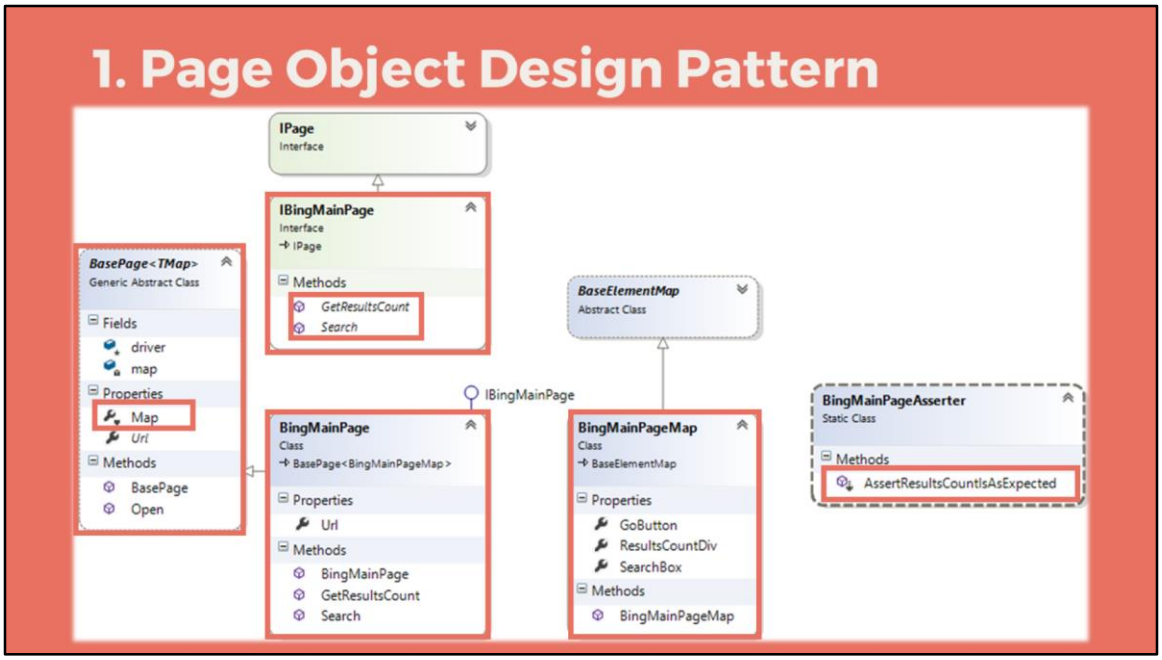


We decided that the best way to separate the different pieces of our broad facades was to move them to different Pages.
Until now I showed you almost only the darkest sides of the mentioned design patterns. From now on I'm going to show how the design patterns should be used correctly. Because of that I am restarting the design patterns count in the title.
The famous programmer Martin Fowler was one of the first that mentioned Page Object as a pattern and explained its usage. You may not be able to find it in the list of all "official" design patterns because the community started using it recently. **Most of the other design patterns have been about for more than 20 years.**

Selenium WebDriver Page Object

```
public class BingMainPage
{
    private readonly IWebDriver driver;
    private readonly string url = @"http://www.bing.com/";

    public BingMainPage(IWebDriver browser)
    {...}

    [FindsBy(How = How.Id, Using = "sb_form_q")]
    public IWebElement SearchBox { get; set; }

    [FindsBy(How = How.Id, Using = "sb_form_go")]
    public IWebElement GoButton { get; set; }

    [FindsBy(How = How.Id, Using = "b_tween")]
    public IWebElement ResultsCountDiv { get; set; }

    public void Navigate()
    {
        this.driver.Navigate().GoToUrl(this.url);
    }

    public void Search(string textToType)
    {
        this.SearchBox.Clear();
        this.SearchBox.SendKeys(textToType);
        this.GoButton.Click();
    }
}
```

I saw Page Objects for the first time in Selenium WebDriver. In general a page object abstracts **an HTML page**. It **provides an easy to use interface** for **manipulating the page elements** without searching for them in the HTML.
In the example the different properties represent the different elements of the pages. They are located through the help of the FindsBy attributes **that are holding the finding strategy**. Below them you can find the different actions that can be performed on the page.



Nonetheless, after our bloody experience with the thousands-code-lines facades, I suggested that we need to modify the pattern a little bit to fit our needs.
My proposal was to separate the elements, logic, and assertions. **As you can see in the class diagram,** the page holds a reference to the element map through the Map property, inherited from the base page class. The page implements a specific interface that defines what actions it should be able to do. The test classes know only about the page object but not about the element map. **The assert methods are implemented as extension methods of the interface of the page**. As a result, we can use them directly in the tests as normal methods provided by the concrete page.

Page Object Element Map

```
public class BingMainPageMap : BaseElementMap
{
    public BingMainPageMap(IWebDriver driver) : base(driver)
    {
    }

    public IWebElement SearchBox
    {
        get
        {
            return this.driver.FindElement(By.Id("sb_form_q"));
        }
    }

    public IWebElement GoButton
    {
        get
        {
            return this.driver.FindElement(By.Id("sb_form_go"));
        }
    }

    public IWebElement ResultsCountDiv
    {
        get
        {
            return this.driver.FindElement(By.Id("b_tween"));
        }
    }
}
```

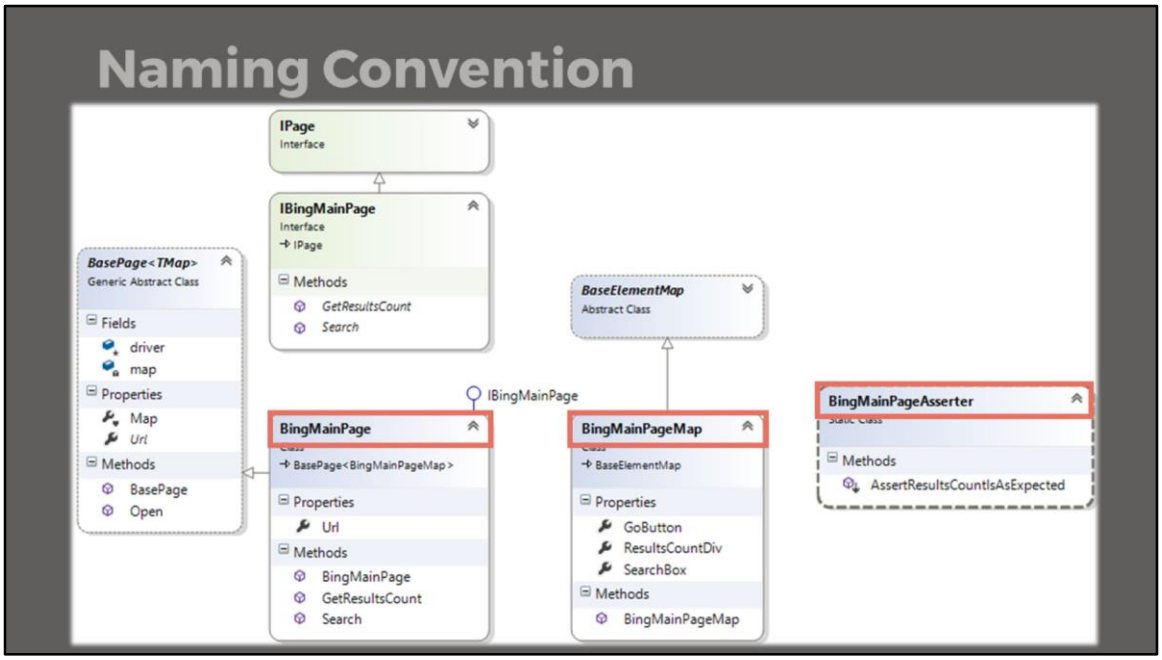
The Page Object Element Map contains all elements and their location logic.
What is the benefit? **The benefit is that if you have to fix some localization logic it is present only in a single place**. Second the page map hides the nitty-gritty details of the **FindElement** strategy, like Java Script or chained method calls. Thanks to all of that you **gain** better **maintainability**, a more **readable code** and **smaller files because the code is separated in three different logical units.**

Page Object Asserter

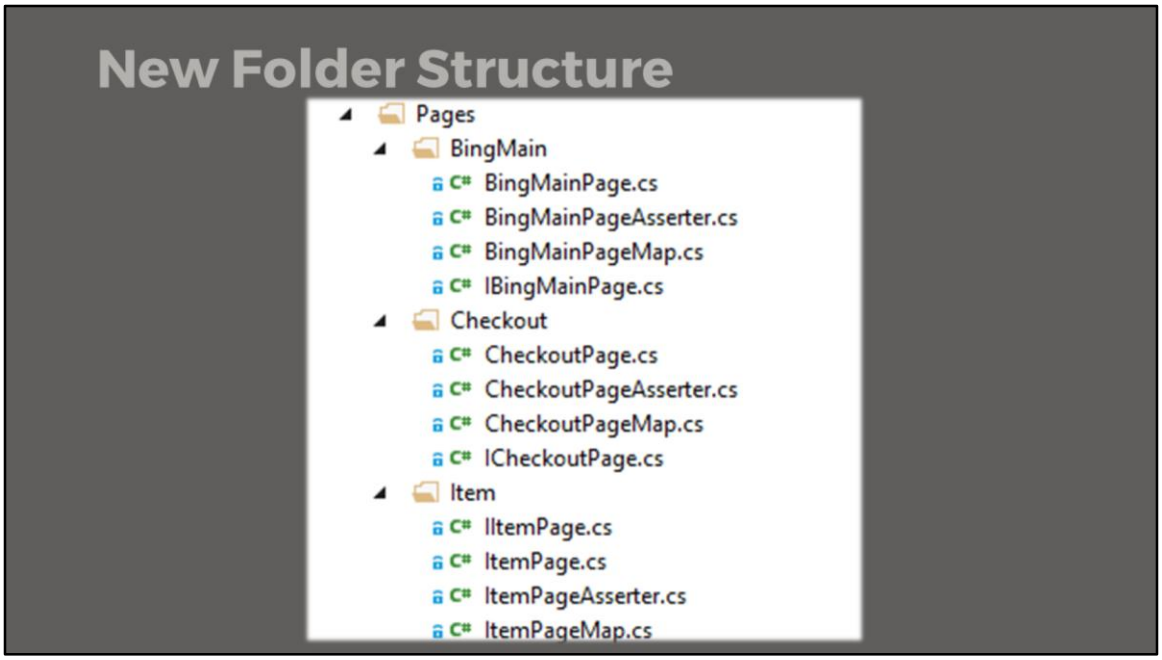
```
public static class BingMainPageAsserter
{
    public static void AssertResultsCountIsAsExpected(this IBingMainPage page, int expectedCount)
    {
        Assert.AreEqual(page.GetResultsCount(), expectedCount, "Custom Exception Message.");
    }
}
```

The Page Object Asserter consists of **the assertions that can be performed on the page**. The Page Object **Asserter increases the code reuse by holding the assertions that are used many times in one place.** You do not need to fix them in multiple spots throughout your code base. Further, the specified exception messages are reused.
These methods extend the interface of the page and can be used as normal methods directly in the tests. If the programming language you are using is not having a feature like the extension methods you can always pass the interface as a normal parameter and remove the static modifiers.

The Page Object holds the actions that can be carried out on the page, like Search and Navigate. The best implementations of the pattern hide the usage of the Element Map. **This makes the actual tests more readable and more understandable** from business perspective as the complex framework logic is **hidden behind the page objects.** By complex framework logic I mean actions like clear the text input before real typing, wait for page to load, execute a Java Script code and so forth.



I want to draw your attention to the names of the participant classes. All of their names are containing the name of the web page, in this case BingMain, followed by the word Page. Element maps' names are ending with the suffix **Map** while the asserter's classes are ending with the word **Asserter.** Of course, this is our convention; you can change it as you like. The main point here is that it is important to have a naming convention**. This way, your teammates know, in each particular moment of time**, at which file they are looking. It makes the searching through files a lot easier.



Another handy thing that we introduced for better coordination was the new folder structure. We placed all pages under **Pages** folder plus every particular page had its own where its files were held. These decisions boosted our development performance to the maximum extent.

Usage in Tests Code Example

```
[TestClass]
public class BingTests
{
    private IBingMainPage bingMainPage;
    private IWebDriver driver;

    [TestInitialize]
    public void SetupTest()
    {
        driver = new FirefoxDriver();
        bingMainPage = new BingMainPage(driver);
    }

    [TestCleanup]
    public void TeardownTest()
    {
        driver.Quit();
    }

    [TestMethod]
    public void SearchForAutomateThePlanet()
    {
        this.bingMainPage.Open();
        this.bingMainPage.Search("Automate The Planet");
        this.bingMainPage.AssertResultsCountIsAsExpected(264);
    }
}
```

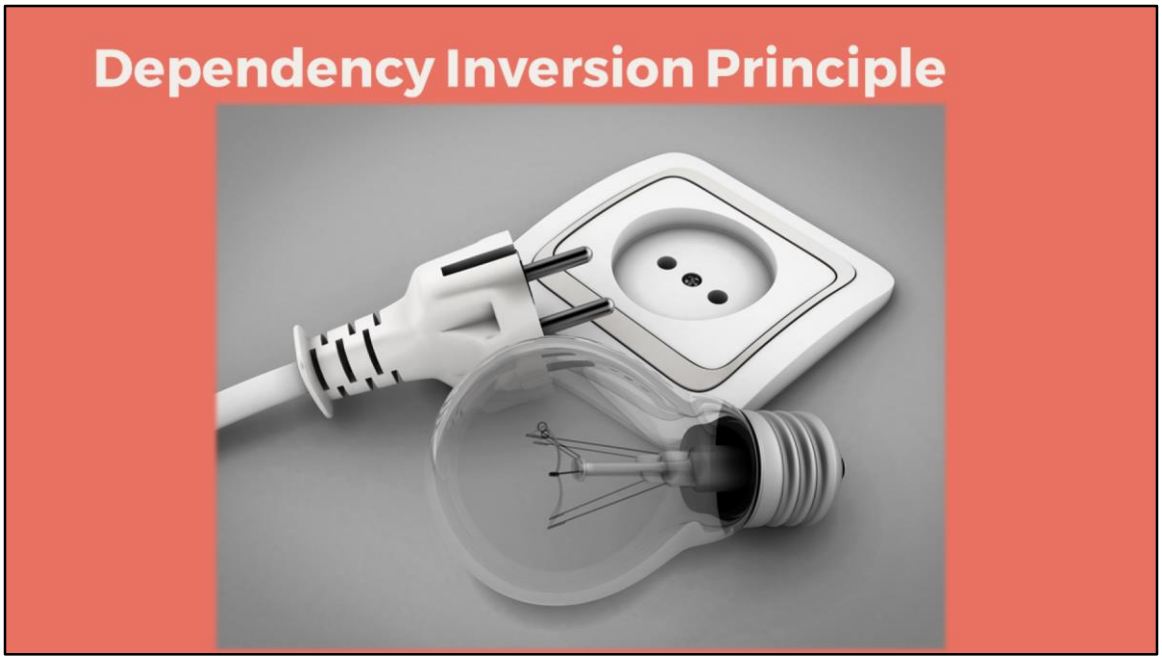
The usage of page objects in tests is straightforward. Create a new instance of the desired page. Then use the provided methods as appropriate. At the end of the test-call, one of the methods of the page asserter. **Also, if you don't like the idea of putting your assertions in a dedicated class** because they are going to be used only once, you are free not to. Just place them directly in your tests. **The suggested approach is appropriate if your assertions are going to be used multiple times.** Of course, you can always mix both approaches.

One of the biggest advantages *of the proposed ideas* is the single responsibility of the page's classes.
In object-oriented programming, the **single responsibility principle states that every class should have responsibility over a single part of the functionality**, and that responsibility should be entirely encapsulated by the class.
In the presented implementation, the map class is responsible only for locating elements, the asserter class for asserting things and the page itself for providing service methods.

2. Facade Design Pattern v.2.0

```
public class ShoppingCart
{
    private readonly IItemPage itemPage;

    private readonly ISignInPage signInPage;

    private readonly ICheckoutPage checkoutPage;

    private readonly IShippingAddressPage shippingAddressPage;

    public ShoppingCart(IItemPage itemPage, ISignInPage signInPage,
                        ICheckoutPage checkoutPage, IShippingAddressPage shippingAddressPage)
    {
        this.itemPage = itemPage;
        this.signInPage = signInPage;
        this.checkoutPage = checkoutPage;
        this.shippingAddressPage = shippingAddressPage;
    }

    public void PurchaseItemAndVerifyWorkflow(string item, double expectedItemPrice,
                                                                    ClientInfo clientInfo)
    {
        this.itemPage.Open(item);
        this.itemPage.AssertPrice(itemPrice);
        this.itemPage.ClickBuyNowButton();
        this.signInPage.ClickContinueAsGuestButton();
        this.shippingAddressPage.FillShippingInfo(clientInfo);
        this.shippingAddressPage.AssertSubtotalAmount(itemPrice);
        this.shippingAddressPage.ClickContinueButton();
        this.checkoutPage.AssertSubtotal(itemPrice);
    }
}
```

This is the code of our shopping cart facade responsible for creating purchases. We decided to use the façade design pattern in a little different way.
It combines the different pages' methods to complete the wizard of the order. If there is a change in the order of the executed actions I can edit it only here. It will apply to tests that are using the façade. The different test cases are accomplished through the different parameters passed to the façade's methods.
These types of facades contain a much less code because most of the logic is held by the pages instead of the façade itself.
Finally, I want to briefly allude to the name of the façade. It doesn't contain the word Façade in its name because it should not suggest that it is hiding a complex logic.



Through the usage of the pages' interfaces the façade follows one of the most superb programming principles, called Dependency Inversion Principle. It suggests that our high-level components (the facades) should not depend on our low-level components (the pages); rather, they should both depend on abstractions. **You can replace the version of some of the pages without changing even a single line of code in the façades.**

The instantiation of the pages is straightforward if you need only one or two. However, different facades can depend on four as in the example or even more pages. This means that their instantiation process can become tedious. **The factory design pattern comes to the rescue.** The factory **encapsulates object creation** and **reduces the dependencies of the application on concrete classes.**

Factory Facade

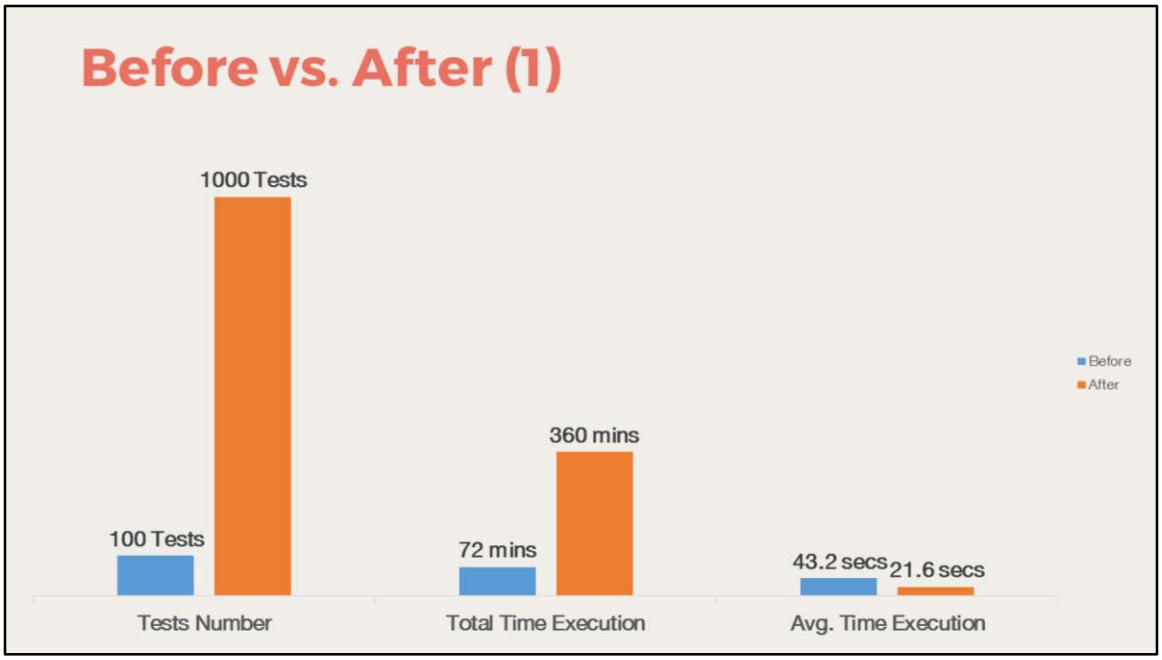
```
public class ShoppingCartFactory : IFactory<ShoppingCart>
{
    private readonly IWebDriver driver;

    public ShoppingCartFactory(IWebDriver driver)
    {
        this.driver = driver;
    }

    public ShoppingCart Create()
    {
        var itemPage = new ItemPage(this.driver);
        var signInPage = new SignInPage(this.driver);
        var checkoutPage = new CheckoutPage(this.driver);
        var shippingAddressPage = new ShippingAddressPage(this.driver);
        var purchaseFacade = new ShoppingCart(itemPage, signInPage,
                                              checkoutPage, shippingAddressPage);
        return purchaseFacade;
    }
}
```

The shopping cart façade factory contains a single method responsible for the instantiation of all dependent pages. Here, also all pages **depend on the IWebDriver interface** and **this is another benefit of constructing them all at once because we can pass the already instantiated driver's instance to all of them.**

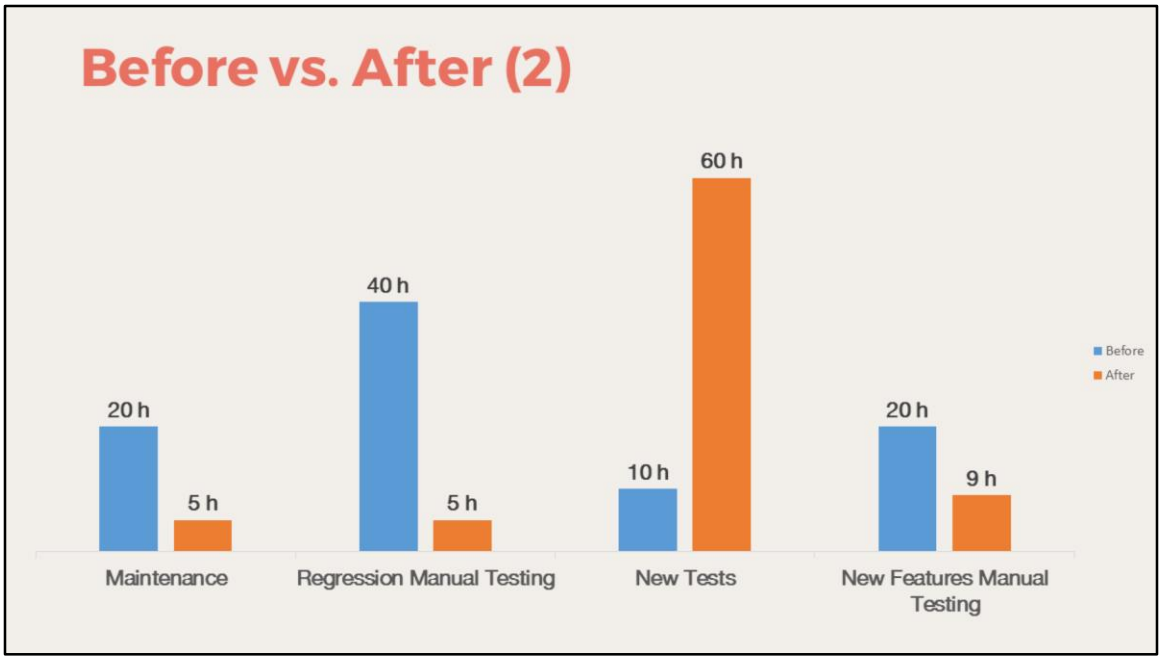
Facade Factory in Tests

```
[TestClass]
public class ShoppingCartTests
{
    private IFactory<ShoppingCart> shoppingCartFactory;
    private ShoppingCart shoppingCart;
    private IWebDriver driver;

    [TestInitialize]
    public void SetupTest()
    {
        driver = new FirefoxDriver();
        shoppingCartFactory = new ShoppingCartFactory(driver);
    }

    [TestCleanup]
    public void TeardownTest()
    {
        driver.Quit();
    }

    [TestMethod]
    public void Purchase_Book_Discounts()
    {

        shoppingCart = shoppingCartFactory.Create();
        shoppingCart.PurchaseItem("The Hitchhiker's Guide to the Galaxy",
                                  22.2, new ClientInfo());
    }
}
```

If you need to use the shopping cart façade in tests, you can get an instance from the shopping cart factory. You are not more obligated to create the dependent pages yourself. As pointed before if you need to replace the implementation of some of the pages used in the shopping cart facade, you need to do it only in a single place, inside your factory class. This allows us to create tests with less code and lowers the maintainability costs associated with them.



At the beginning of our journey, we had 100 tests. A couple of years later with the help of all mentioned practices we managed to increase the total number of tests to over 1000. The 100 tests ran for 72 minutes. **As the tests became coded and more maintainable thanks to the applied design practices, we had more spare time to optimize them**. As result now these 1000 tests run for only 6 hours on a single machine or for 90 minutes using four machines in parallel which means that they are two times faster than before. The previous Avg. Execution Time for a single test was 43.2 seconds, now this time is decreased to 21.6 seconds.



Our team uses the Scrum development methodology, and **we are working with two-week cycles**. Two weeks are equal to 80 working hours in Bulgaria. In the beginning, we needed to spend approximately 20 hours per cycle to maintain our automation. Now the tests are so stable that we need only 5 hours. We can conclude that the tests are four times more maintainable than before. Now all of the manual regression testing is done by our automation and thanks to that we can spend almost 6 times more time writing new tests. That also decreases the required time for manual testing the new features. When the new feature is ready for testing, we are also ready with most of its automation.

To sum up. Design **patterns provide different type of reuse**. A type of reuse that I would like to call "concept reuse" or "idea reuse." Design **patterns provide a general solution to a common problem** so that people with less experience can utilize the know-how of the more adept developer and hopefully save a lot of time, agony and trial & error in the process. Also, the design patterns help to improve the developers' communication because of the usage of a shared language.
Patterns allow you to say more with less. **When you use a pattern in a description of a class, other developers quickly know precisely the design you have in mind.**

However, sometimes **extensive use of design patterns is overkill**. If a developer is very fluent with using design patterns, he or she can start treating them like a golden hammer. But when the requirements and future changes are very predictable, **design patterns can make the code unnecessarily complex**. A simple solution **will be more appropriate**. So design patterns should be used wisely and where needed, and like any other tool that we have at our disposal, if you try to fix everything using it you are going to end up in a big mess.

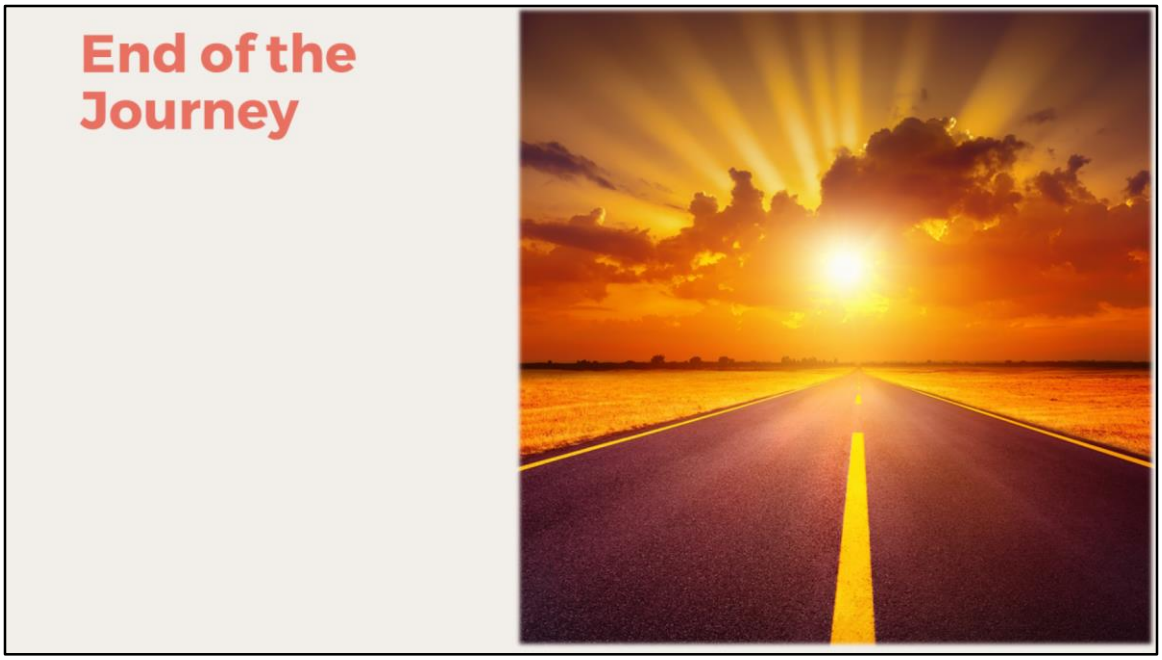
Do a Pilot

If you like the presented ideas and want to apply them in your projects, don't rush it. Do a pilot. Do a small pilot project **to test your assumptions and prove the concept**. Ideally, a pilot should involve a **representative subset of your application** and **have a narrow enough scope that it can be completed in 1-2 weeks.**
Since you cannot be sure what you don't know, it is better to learn your lessons on a small scale.

Going Deeper

- Design Patterns: Elements of Reusable Object-Oriented Software, By The Gang of Four
- Head First Design Patterns, By Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
- Design Patterns In Automation Testing, Automate The Planet
- Code Project
- DZone
- Twitter

Here you can find a few resources that helped me to learn more about design patterns. The first two bullets are the books that helped to extend my knowledge of design principles and patterns. The next bullets are online resources that you can use to expand your know-how. You can find more information in the biggest programming sites like Code Project or DZone. They have dedicated design patterns sections. Further, you can check my site- http://automatetheplanet.com where I write about design patterns in relation to automation testing. Finally, you can follow more adept developers in Twitter and ask them to help you.



These were the most prominent things, in my opinion, that changed the way our tests were working.

There are tons of other things that need to be done before you can accomplish your dream. A few of them are a proper logging, good locators, CI tools, good requirements and so forth. But I believe that these three patterns (Page Object, Façade and Factory) were my team's starting points to success.

There is time for one last design principle.

Reinventing the Wheel

In my opinion, if you need to walk from this lecture with only one thing. It should be that you don't need to reinvent the wheel.
Because you're not alone. At any given moment, somewhere in the world , someone struggles with the same software design problems you have. You know you don't want to reinvent the wheel, so you turn to Design Patterns **– the lessons learned by those who have faced the same problems**. With Design Patterns, you can benefit from the best practices and experience of others, so you can spend your time on … something else. Something more challenging . Something more complex. Something more fun.

Long Term Planning

Keep in mind on the fact that the test automation project will last as long as the application under test is being maintained. Achieving automation is not a sprint; it is a long distance run.

I wish you lots of success in your marathon to accomplish your own dreams.


Thank You!
?questions any
@SEETESTConf
SEETEST
2015 SOUTH EAST EUROPEAN
SOFTWARE TESTING CONFERENCE SOFIA

Thank you!