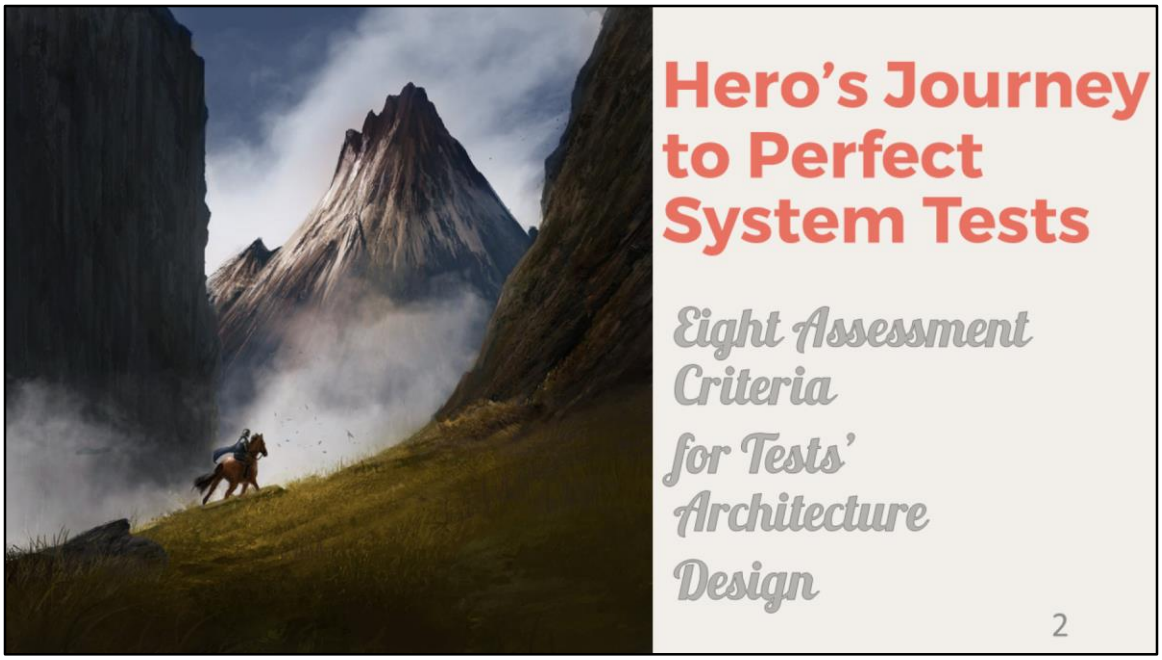
Today I am going to tell you a story. A story about heroes, lots of obstacles and the holy grail of the test automation- the perfect system tests' design recipe. With a mix of personal and mythological stories I am going to present to you eight criteria for system tests design assessment. You can find some of them in different books and blog posts but this list is unique.



My teammates and I created it specifically for our system tests design improvements. Today you will hear the whole story. What problems we had initially and how we developed a system to solve them. My inspiration about this talk came from the so called 'Hero's Journey'.

The Lecturer

- Anton Angelov
- Quality Assurance Architect
- Progress
- Site: http://automatetheplanet.com
- @angelovstanton
- angelov.st.anton@gmail.com

HEISENBUG

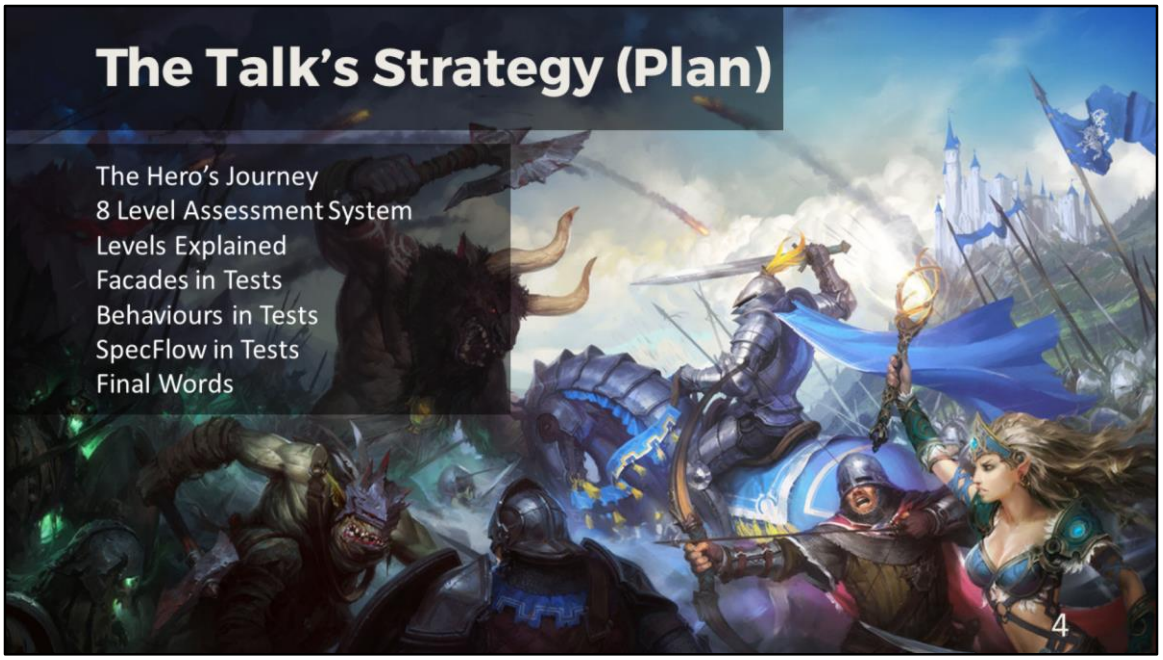
Progress

CODE PROJECT For those who code

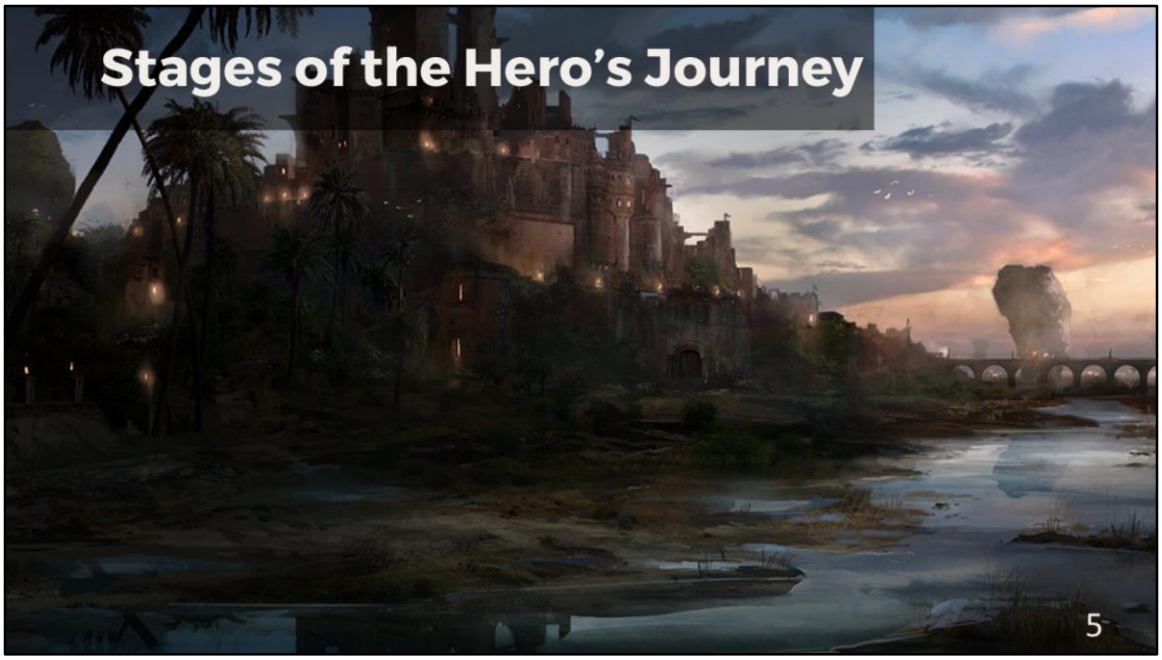
DZone MVB Most Valuable Blogger

3

But first who am I. I'm Anton Angelov, a Quality Assurance Architect in Progress. I'm a proud owner of automatetheplanet.com where I share all my ideas about code and tests. Also, I'm a most valuable Blogger at **DZone** and a MPV at **Code Project**.

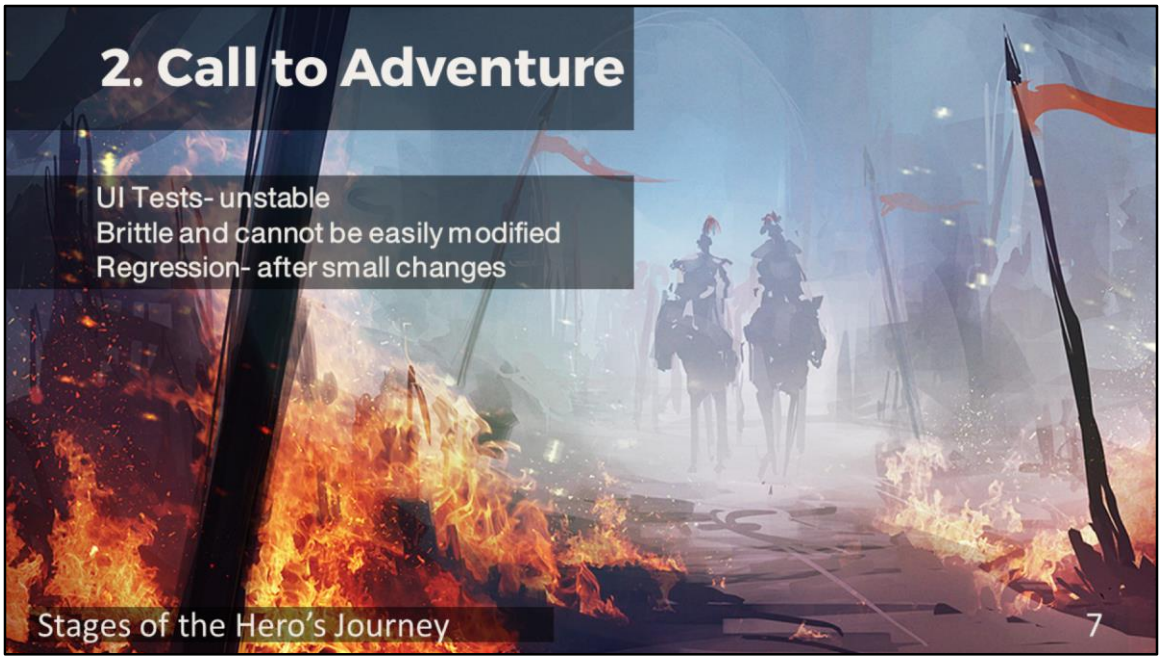


Before we begin, just let me quickly introduce to you the plan of the talk. First I am going to tell you what our problems were at the beginning through- storytelling and analogies about the hero's journey. Then I am going to present to you the core idea about the 8 level assessment system and what for we use it. After that, I will explain the concept behind each level. The rest of the presentation will be about giving examples how to apply the proposed system in the real world, comparing different test designs and chose the best one. By tests design, I mean different methods for writing end to end tests rather than boundary value analysis and so on. What classes we use, how we arrange the tests.
The first example will be about facade based tests. I will quickly explain what the facades are and then use the system to evaluate the design. The second one will be about behaviours in tests. We will assess them as well. Lastly, we will talk about Specflow. I will again shortly tell you more about the design and then compare it using the system. At the end, we will sum up everything learned.



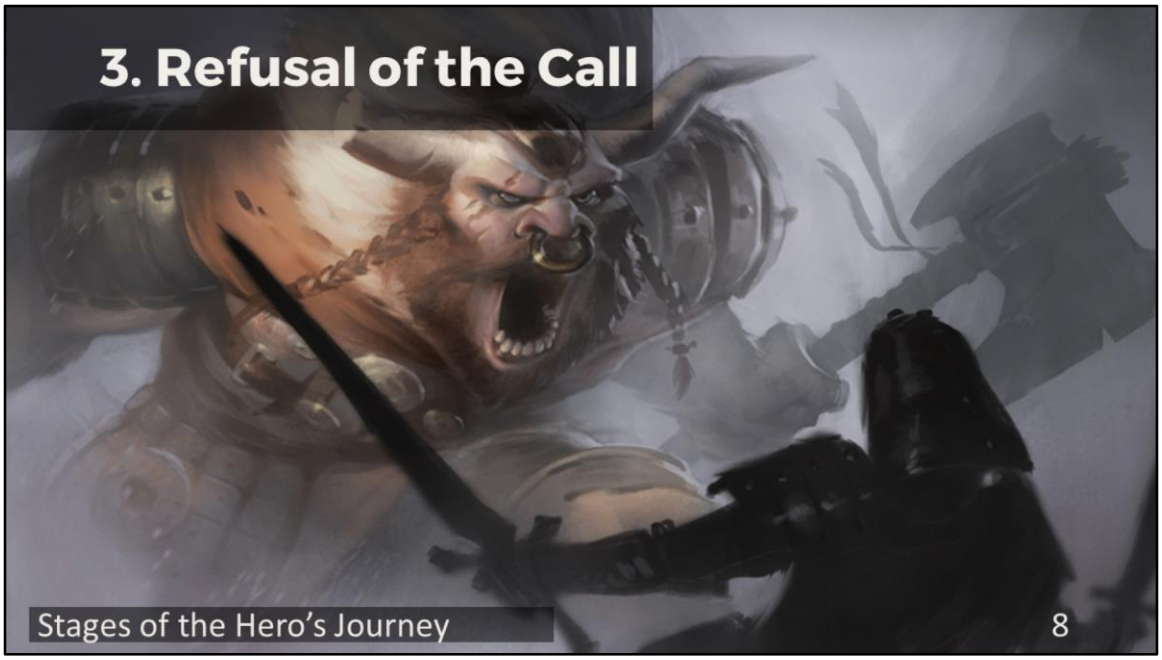
Intrigued by mythology, author Joseph Campbell studied the myth and made the famous claim that nearly all myths, and some other story types, share common ideas and format. The different adventure stages are called the "hero's journey". I am going to tell you my "hero's story" or at least one of them. There are twelve steps to the hero's journey.

This step refers to the hero's normal life before the adventure begins.
We had over 1000 tests that ran for over 6 hours on a single machine. Sometimes all of them were green but sometimes they were problems so we needed to troubleshoot them over and over again.



The hero faces something that makes him begin his adventure. This might be a problem or a challenge he needs to overcome.

The biggest problem for us was that we couldn't trust our UI tests. They verified big part of the system but were so brittle because they weren't designed in a way that can be easily modified. Small changes in the main workflow caused almost always regression in a random group of tests. Our challenge was to find a better design so that we can refactor them and make them more maintainable, more readable and always green.

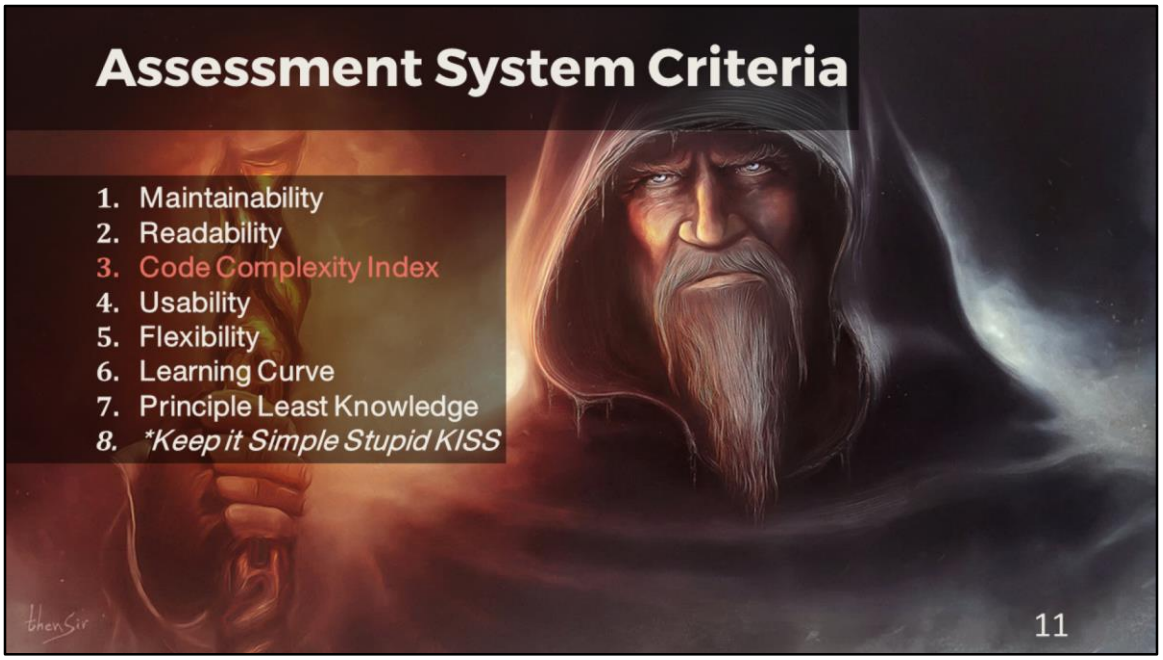


The hero attempts to refuse the adventure because he is afraid.
Before we came up with the system, we tried to patch up the tests and find quick solutions, hoping that this way we can fix the regression problems and simultaneously be able to add new tests. However, our problem here was that for quite some time we didn't have the whole picture. As you will see through the analysis and comparing of the different ideas, we can achieve much better results.

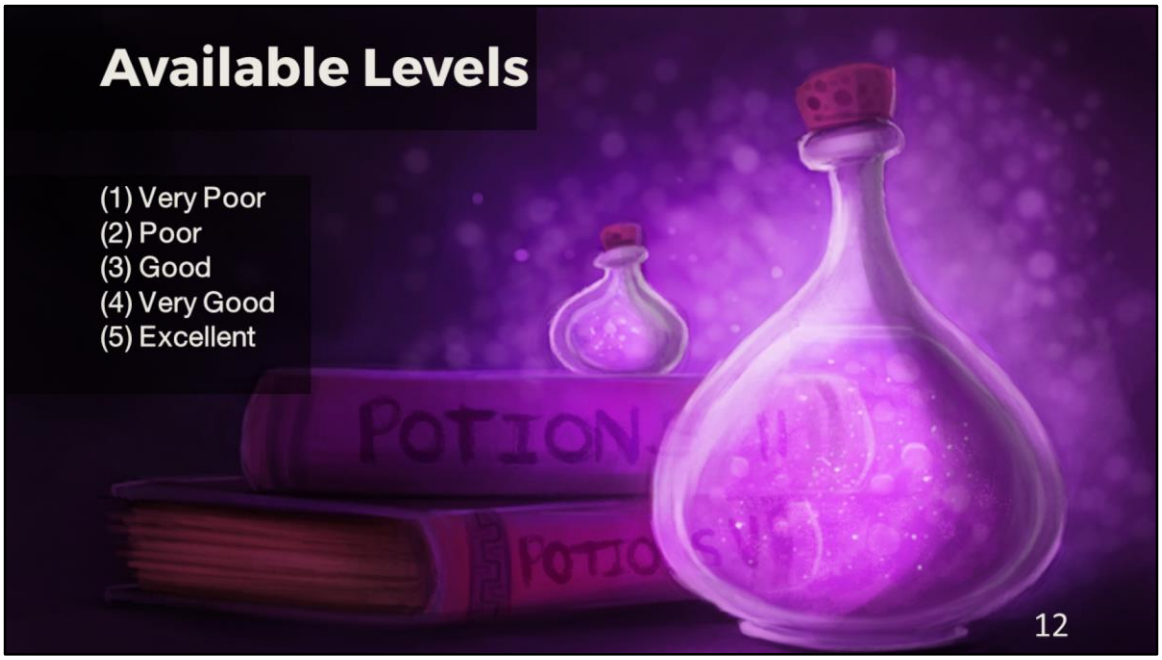
The hero finds someone who can give him advice and ready him for the journey ahead.

We had this issue that the developers didn't know what the QAs are doing and usually the design and architecture of the tests was a responsibility of single man that couldn't know everything. I think one of the best things that the system gave us was that the final decision for the most appropriate design of the tests was result of a team effort.

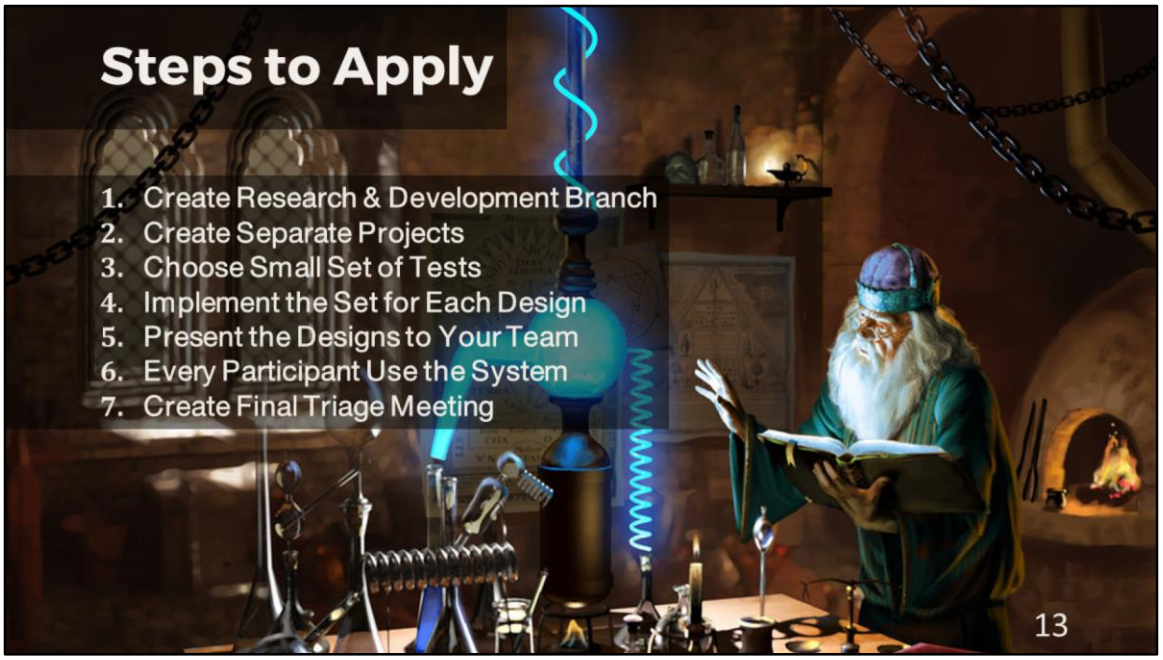
The hero leaves his ordinary world and crosses his first threshold.
My team is using the Scrum agile methodology. As you may know in Scrum at the end of every sprint there is the so called retrospective meeting. As a team we decided that we need to revise our current approach for writing system tests. So my hero's journey started.



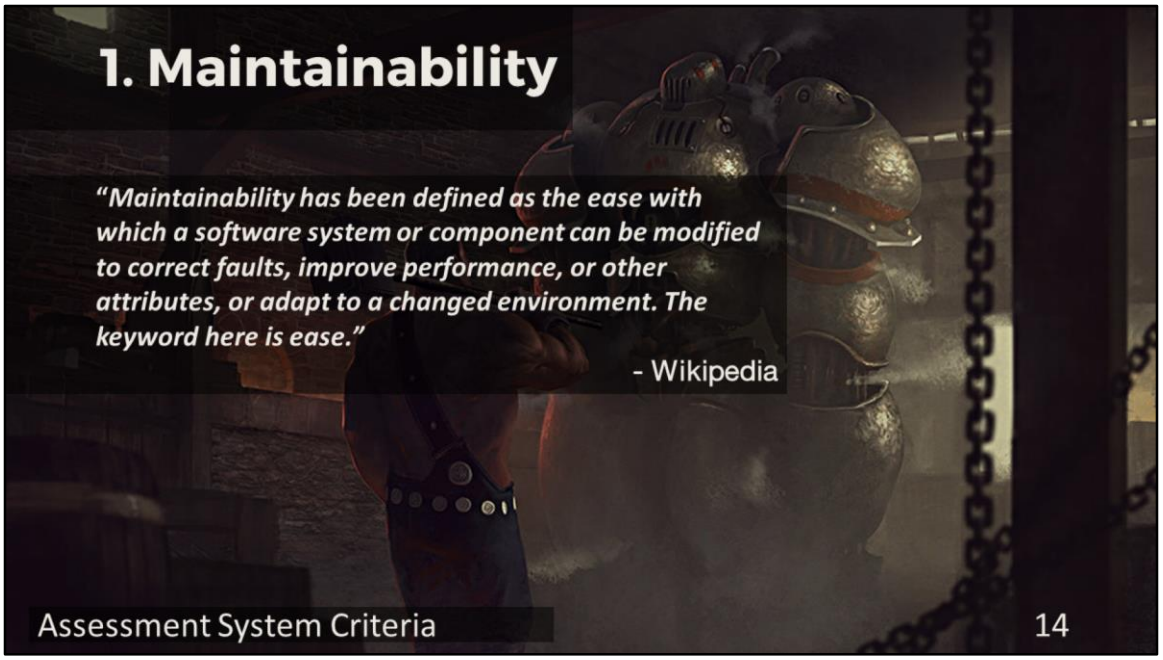
So now it's time to present to you the different levels of the system. Each level represents a characteristic of the tests. As you will see, they are listed in a numbered order which means that they are ordered by importance. However, I think this order depends highly on the context of your team and the skill of its members. So you can reorder the criteria if you want. Our team is responsible for a complex legacy licensing system we need to have lots of regression tests and be able to extend and modify them easily because of that the maintainability holds the first spot. Since we have lots of tests, they need to be readable because sometimes the tests are documentation too. The third one is CCI, I will tell you more about it a little bit later, but it represents how complex our code is, we want our code to be simple. Also, it is the only tool calculated metric. We don't want to reinvent the wheel, so the usability is important. The next one is flexibility. How easy is to learn to write tests. The seven is connected with the maintainability, I will tell you more about it in a bit. Our last resort of comparing is that the simplest design wins if all other criteria are equal. It is not a metric but a principle.



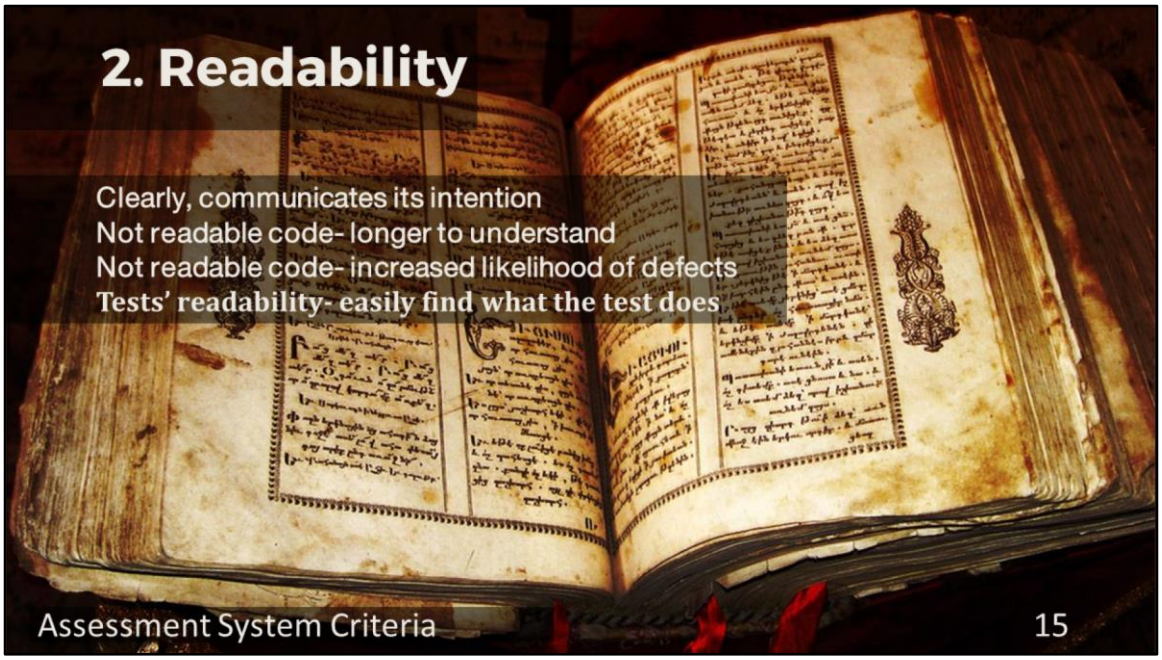
For every criterion, there will be a rating assigned. You can find the possible ratings on the slide. And of course, they have a number representation.



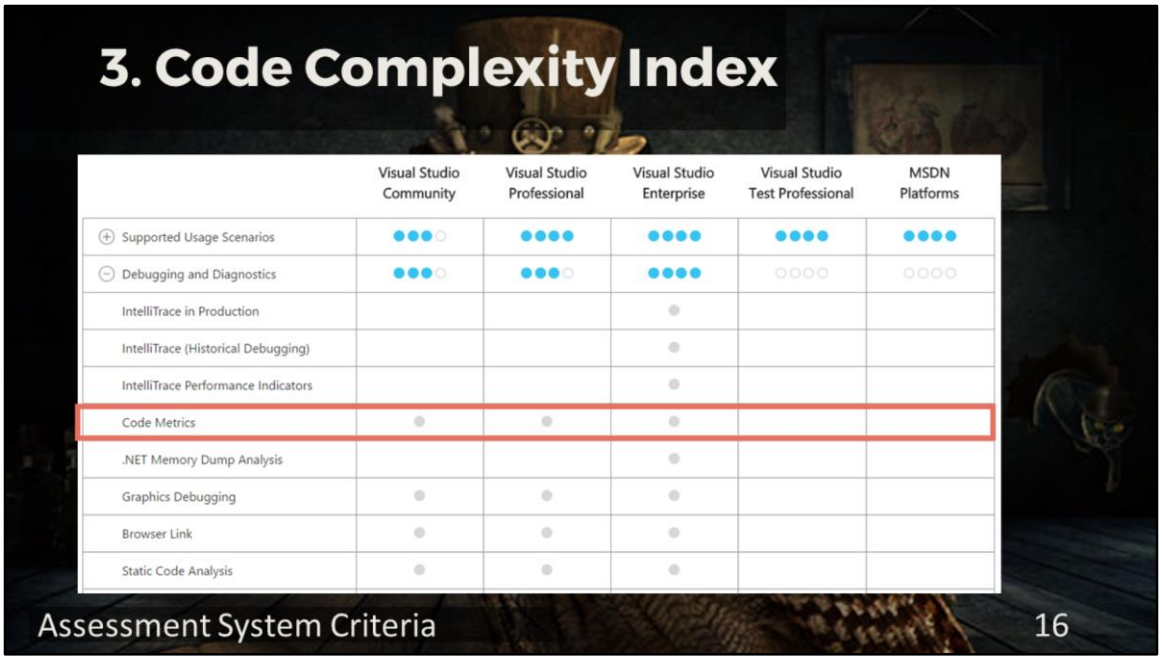
Here are some pragmatic steps to apply it.
1. First, create whole new “Research & Development” branch.
2. Then create separate projects to test your new ideas (). Do not refactor your existing test framework’s code before you are completely sure which idea is the best for your case.
3. To be able to evaluate effectively and assess the different ideas it is best if you choose a small set of identical tests to implement.
4. Create different folders for each idea. Choose a same small set of identical tests to implement for each design. If the tests you create are different, how do you expect the assessment to be accurate? Usually, we don’t refactor directly all of our tests because it costs a lot of time that we usually don’t have. Anyway, the system will work for any number of tests.
5. Present the design to your team.
6. Use the provided eight-level evaluation system to assess the different solutions. It is best if a couple of people participate in the process because some of the points are personally subjective (like what is readable test or which design is more easy to learn)
7. Create a final triage meeting with your whole team and decide which idea to implement based on the results of the assessment. Before we proceed with the examples how we use the system, I am going to explain what every criterion means.



The official definition by Wikipedia is the following: **Maintainability has been defined as "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment". The keyword here is ease.**
The most important part for me is the troubleshooting. How much time do you need to find out if there is a bug in the functionality that the test is asserting or it is a problem with the test itself? When there is some issue in the code- you are looking into the logs. You are all sweaty, looking and looking, unable to locate it. And debug deeper and deeper, and deeper to find out the root cause. I am sure you have experienced it more than once. This is the maintainability what I mean.



Readable code is code that clearly communicates its intention to the reader. Code that is not readable takes longer to understand and increases the likelihood of defects. There is a tendency for some programmers to use comments as a substitute for readable code or to simply comment for the sake of commenting. I believe tests' readability means how easy is to find what the test does without the need of huge comments or large tests' descriptions. I am sure all of you at least once in your lifetime have seen a test's name that is two rows long.

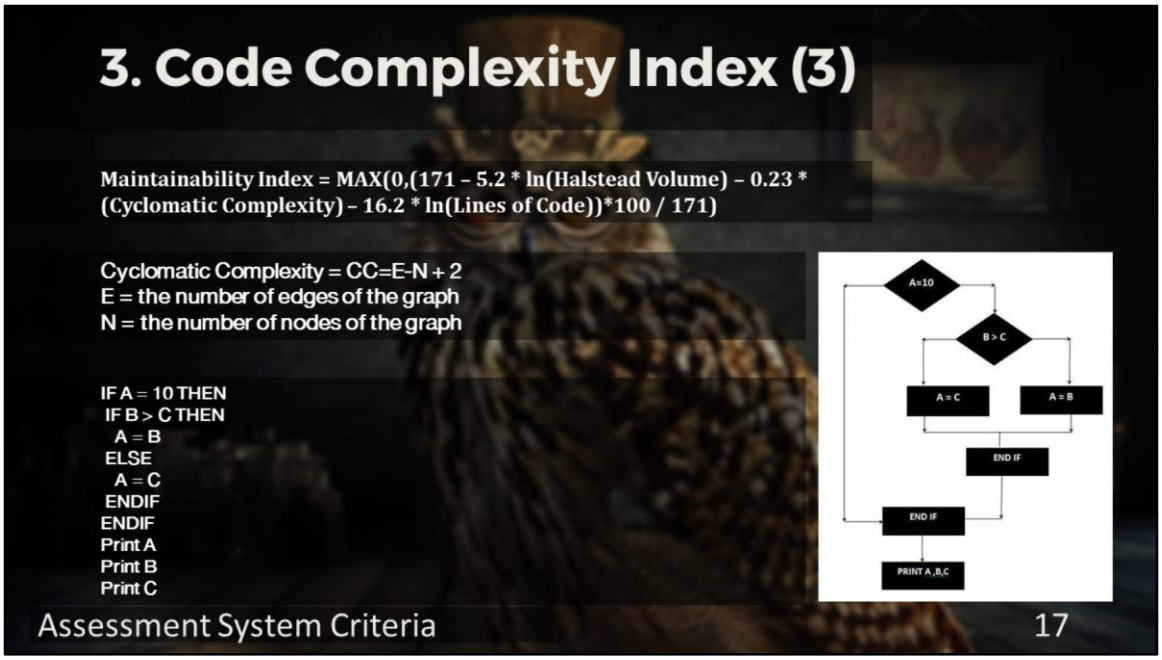


The code complexity index is our custom-made metric. We created a formula for it. It contains four important parts that can be calculated with tools such as Microsoft Visual Studio IDE. This is the only metric from the system that is tool calculated. All others are based on the participants' opinion.

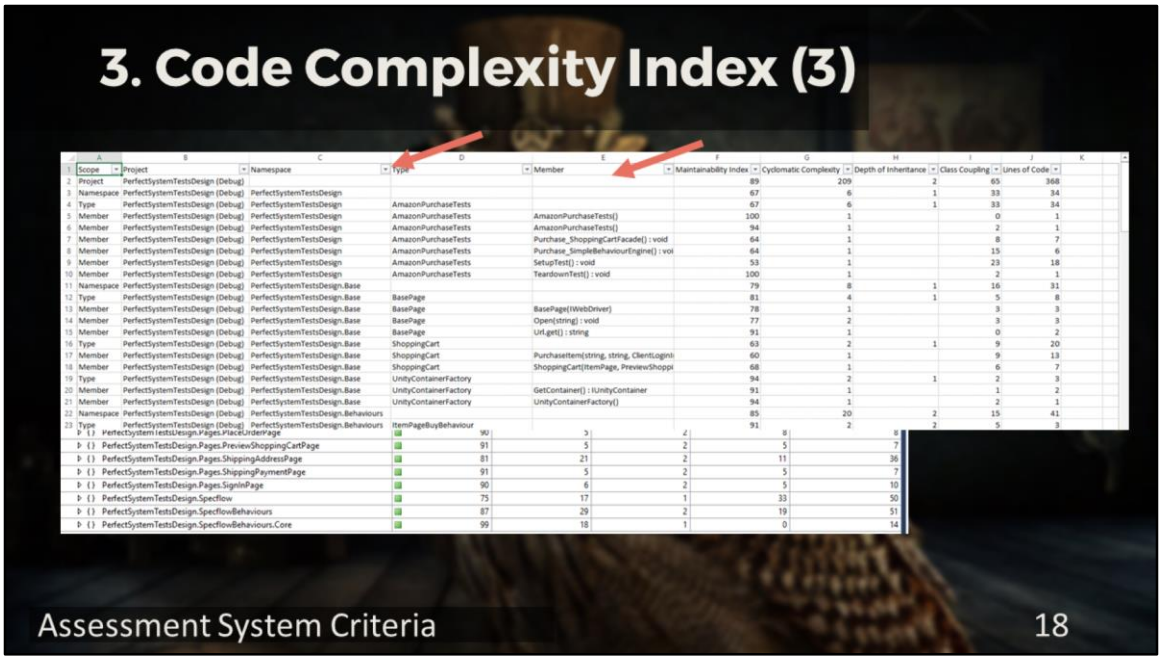
Depth of Inheritance –The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are.

Class Coupling – Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of it's interdependent on other types.

These metrics' calculations are available in the development editions of the application, even in the free one- the community edition.

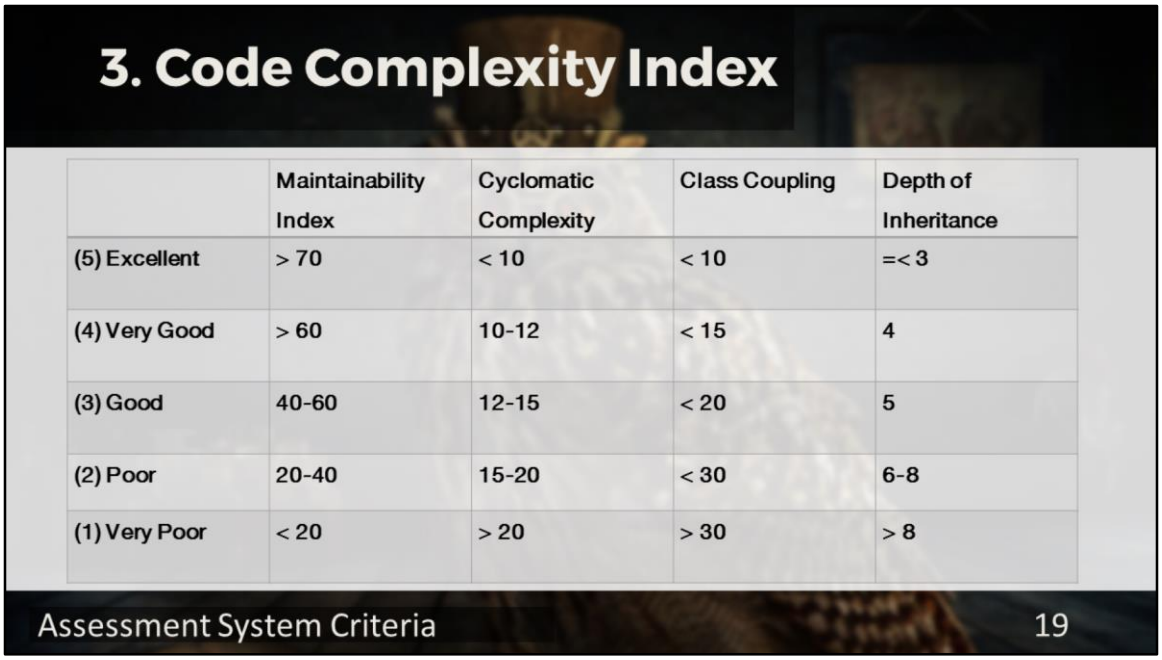


Maintainability Index – Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Most of the formulas used to calculate the metrics are not public. However, I found an unofficial one for the maintainability index. I am not going to decipher it. I wanted to emphasise that real mathematics stays behind this metrics.
Cyclomatic Complexity – Below you can find the formula for Cyclomatic complexity. The Cyclomatic complexity is based on the number of decisions in a program. The control flow shows seven nodes (shapes) and eight edged (lines), thus using the formal formula the Cyclomatic complexity is 8 – 7 + 2 equal to 3.

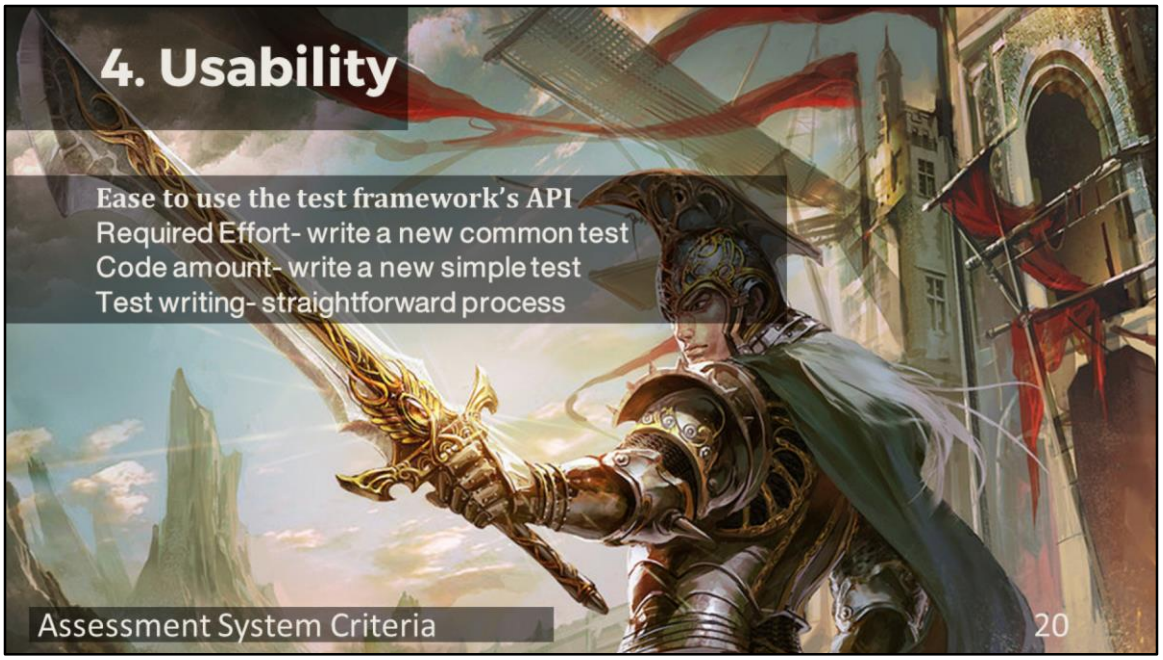


Scope	Project	Namespace	Type	Member	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Project	PerfectSystemTestsDesign (Debug)				89	209	2	65	368
Namespace	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign			67	6	1	33	34
Type	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests		67	6	1	33	34
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	AmazonPurchaseTests()	100	1		0	1
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	AmazonPurchaseTests()	94	1		2	1
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	Purchase_ShoppingCartFacade() : void	64	1		8	7
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	Purchase_SimpleBehaviourEngine() : voi	64	1		15	6
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	SetupTest() : void	53	1		23	18
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign	AmazonPurchaseTests	TeardownTest() : void	100	1		2	1
Namespace	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base			79	8	1	16	31
Type	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	BasePage		81	4	1	5	8
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	BasePage	BasePage(IWebDriver)	78	1		3	3
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	BasePage	Open(string) : void	77	2		3	3
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	BasePage	Url.get() : string	91	1		0	2
Type	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	ShoppingCart		63	2	1	9	20
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	ShoppingCart	PurchaseItem(string, string, ClientLogin	60	1		9	13
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	ShoppingCart	ShoppingCart(ItemPage, PreviewShoppi	68	1		6	7
Type	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	UnityContainerFactory		94	2	1	2	3
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	UnityContainerFactory	GetContainer() : IUnityContainer	91	1		1	2
Member	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Base	UnityContainerFactory	UnityContainerFactory()	94	1		2	1
Namespace	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Behaviours			85	20	2	15	41
Type	PerfectSystemTestsDesign (Debug)	PerfectSystemTestsDesign.Behaviours	ItemPageBuyBehaviour		91	2	2	5	3
{}	PerfectSystemTestsDesign.Pages.PlaceOrderPage		90	3	2	8	8		
{}	PerfectSystemTestsDesign.Pages.PreviewShoppingCartPage		91	5	2	5	7		
{}	PerfectSystemTestsDesign.Pages.ShippingAddressPage		81	21	2	11	36		
{}	PerfectSystemTestsDesign.Pages.ShippingPaymentPage		91	5	2	5	7		
{}	PerfectSystemTestsDesign.Pages.SignInPage		90	6	2	5	10		
{}	PerfectSystemTestsDesign.Specflow		75	17	1	33	50		
{}	PerfectSystemTestsDesign.SpecflowBehaviours		87	29	2	19	51		
{}	PerfectSystemTestsDesign.SpecflowBehaviours.Core		99	18	1	0	14		

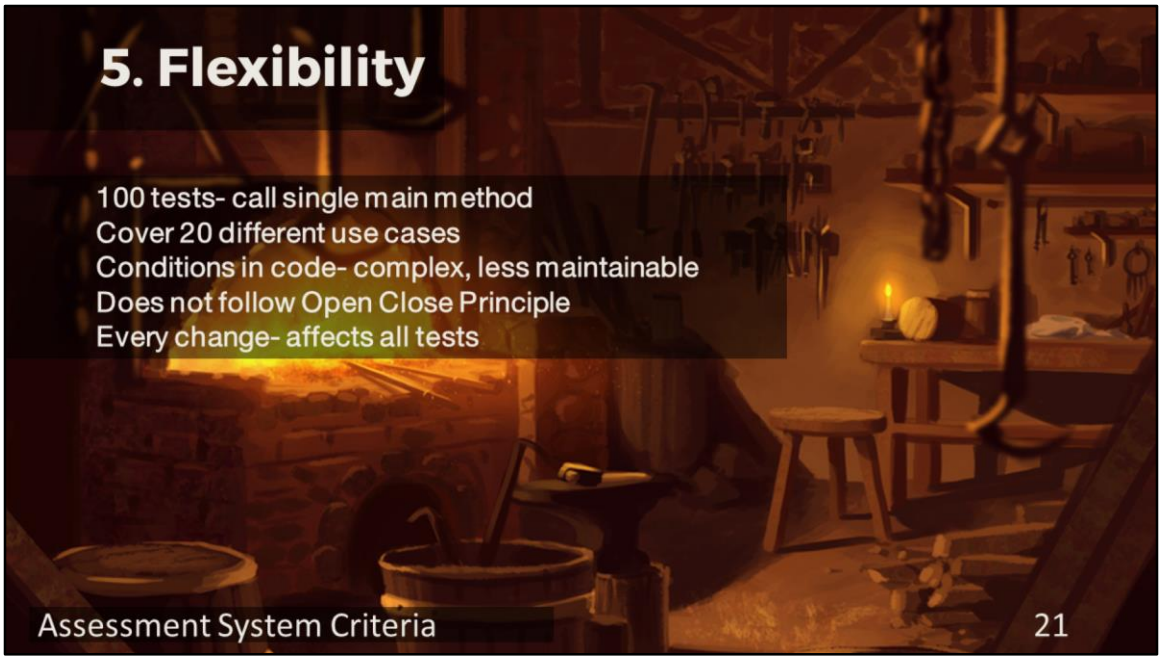
Here I added a few images how to calculate the mentioned metrics using Visual Studio. First, you need to select your project and find the Analyse menu item in the context menu and click Run Code Analysis. After few seconds the code metrics results window will show up. There you will find this green Excel icon. Once you click it, you will be able to export the results to an Excel sheet. Since you cannot calculate the metrics only for a couple of classes only for the whole project. To be able to calculate the parameters only for a few chosen classes you can filter them by full namespace which is available on the sheet.



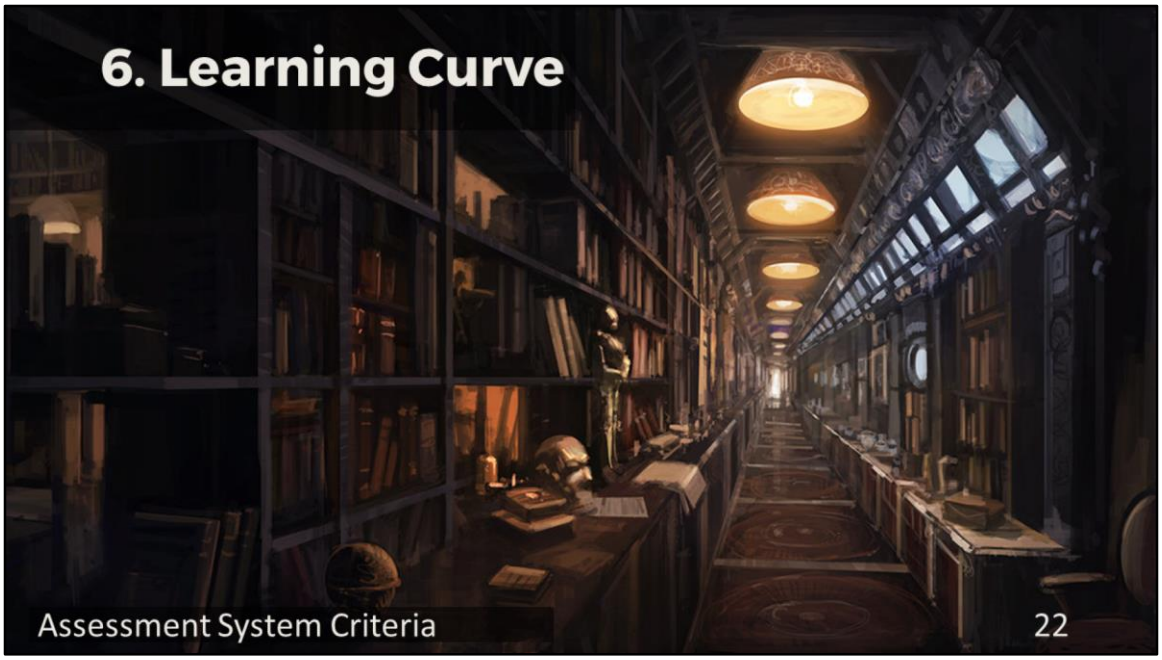
I could not find any official values published by Microsoft for assessment of these criteria. So I did some research and read blog posts of Microsoft MVPs that suggested a sample assessment system. I modified it a little bit to fit our needs. You can observe the result in the presented table. We use the table to calculate the rating for the different parts of the formula.



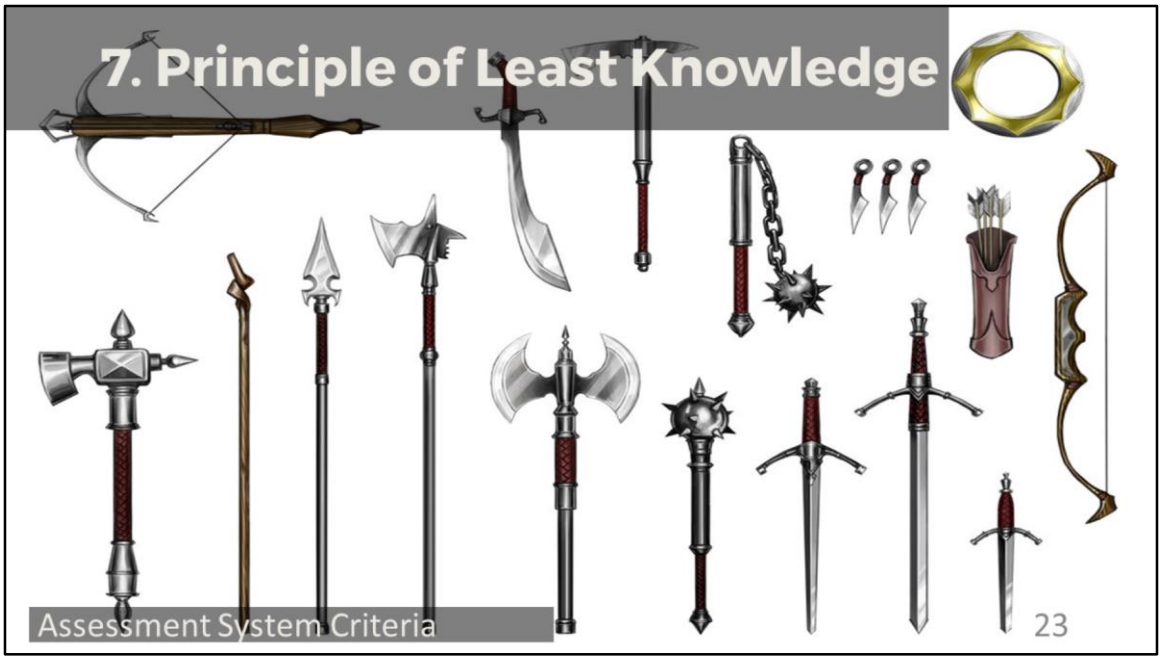
By usability, I mean how easy is to use the test framework API. How much effort is required to write a new common test leveraging on the existing test API? How much code do you need to write a single simple test? If you use complex design patterns and lots of classes, your tests may become really complex. The tests writing should be a straightforward process, should bring joy and pleasure to the writer. My dream is to be able to write tests through 3 4 touches of the keyboard. This is something like the tests' variation of iPhone.



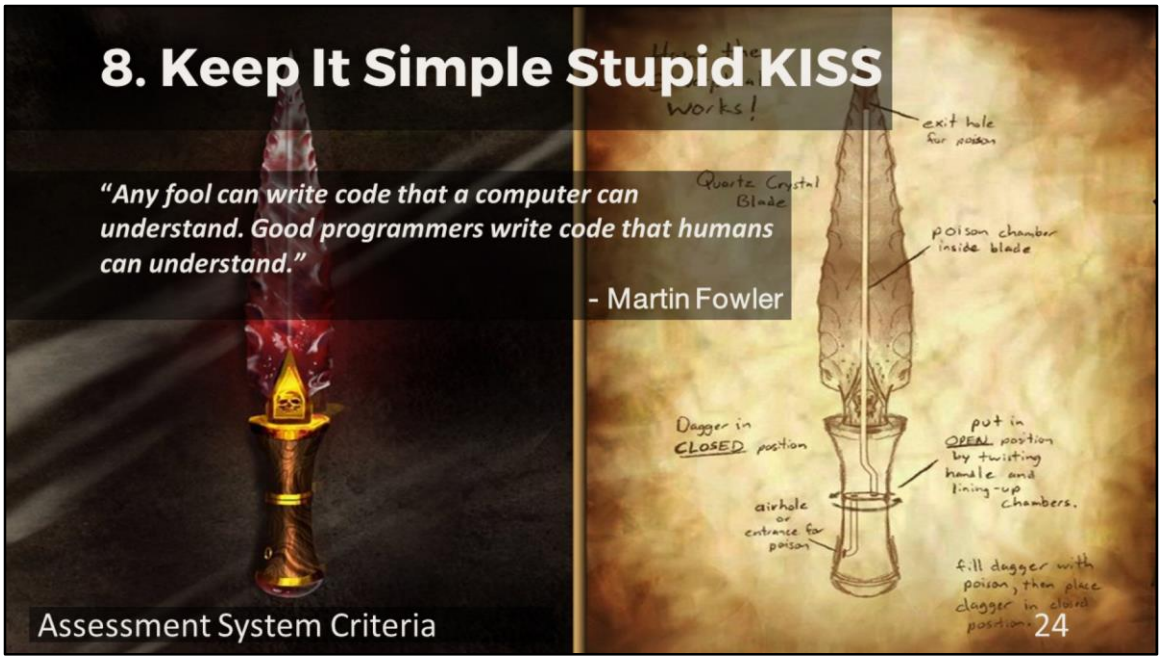
By flexibility, I mean how easy is to add a new step to the existing workflow. If you have 100 tests that use one primary method and the whole process is described there that means that if you want to support 20 different use cases, you need to have lots of conditions in your code. Usually, the conditions tend to make the code more complex and less maintainable. Also, design as previously described will not follow the Open Closed Principle that states that software entities should be open for extension, but closed for modification. Every change in this imaginary method can affect all of the tests that use it. Best tests framework's designs should allow you to add new steps quickly without the possibility to affect all other tests.
For example, you want to add a new assert for the new tax applied on the last step of the shopping cart. You want to add it just for two tests and not affect the other 20.



If a new member joins your team and he needs to read 100 pages long documentation before he is ready to write his first test. Or even worse if you don't have any documentation and you need to spend countless hours teaching each new member how to start writing. This means you have a poor test framework API learning curve.



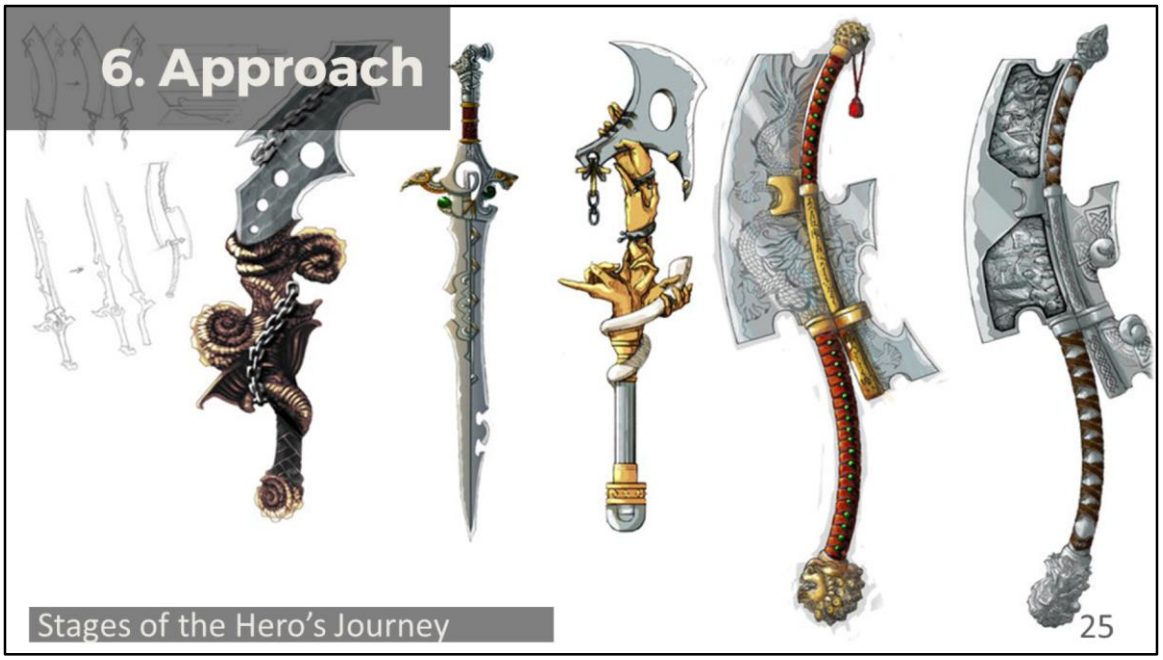
When the assessment system was designed most of our tests shared the currently executed test's data through a static class. Most of the times the different components of the design did not need to use the whole information so we decided to include the principle to our list. For example if you have a client that have first, last name, email, country and so on. And you have a test for resetting a password. If you pass only the email everything is ok but if you need the whole object this is a problem.



Keeping things simple is, ironically, not simple! It requires abstract thinking. Let me quote Martin Fowler:

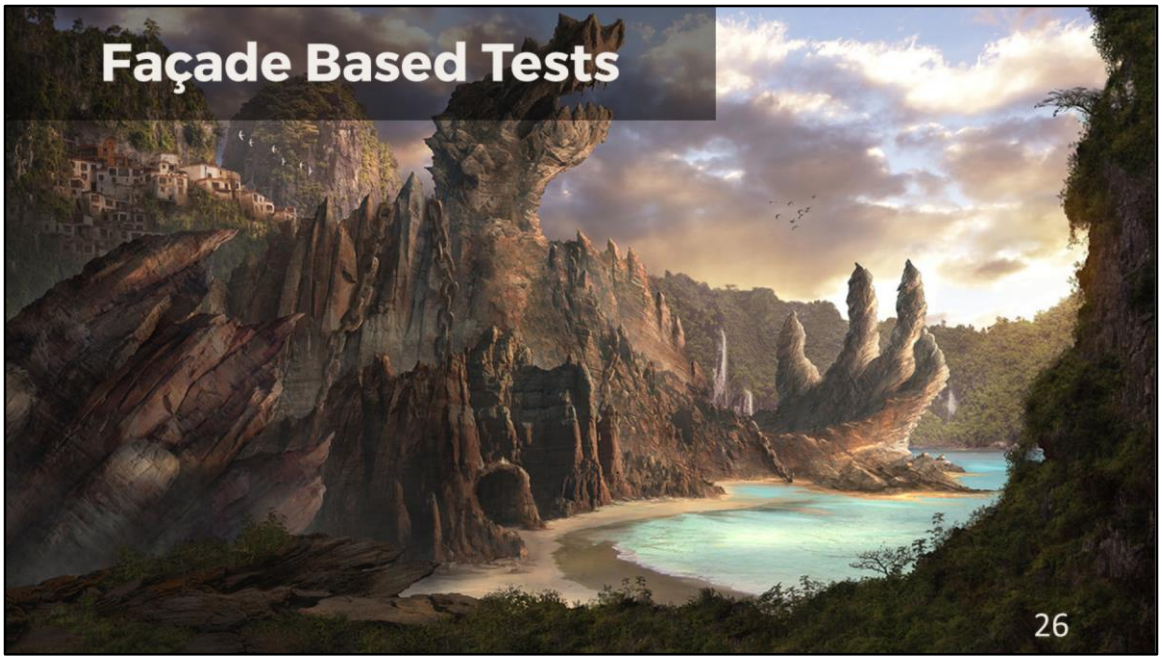
"*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*" Think about it for a second—how much code have you seen that was easy to read, that was simple enough to understand? Probably not a lot.

This is not a metric as the previous ones. And we don't assign a rating for it. We just apply this principle if all other criteria are equal but usually is not necessary. This is like ан election ballotage. But as you will see, we are not going to use it in the examples.

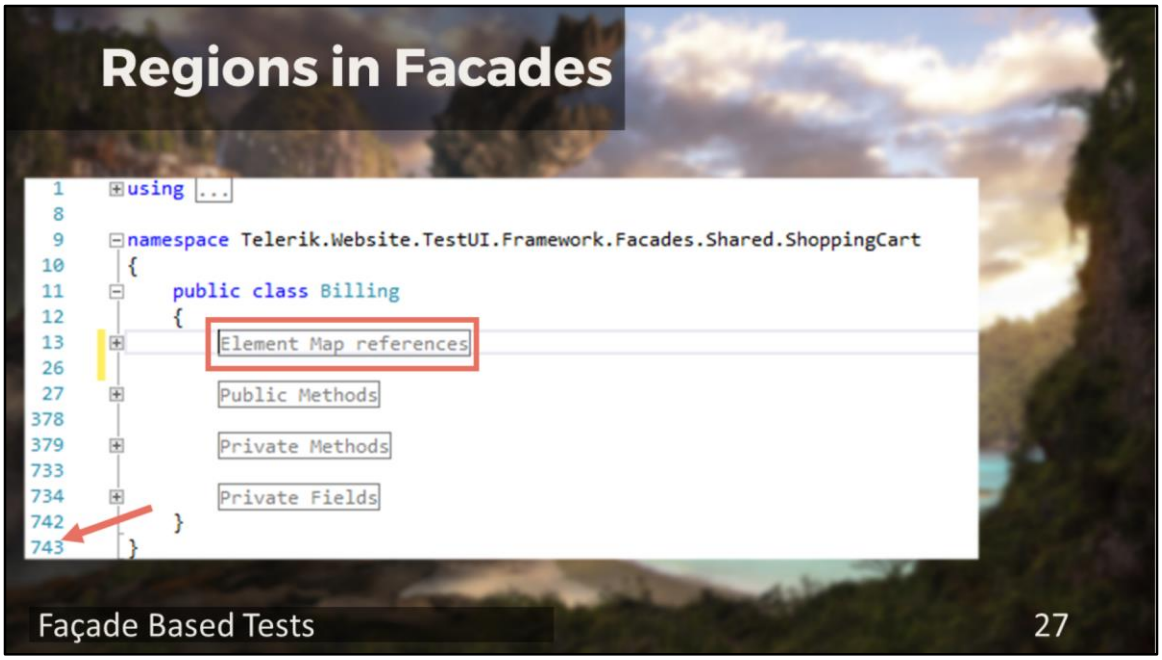


The sixth step of the hero's journey is the Approach step. Setbacks occur, sometimes causing the hero to try a new approach or adopt new ideas.

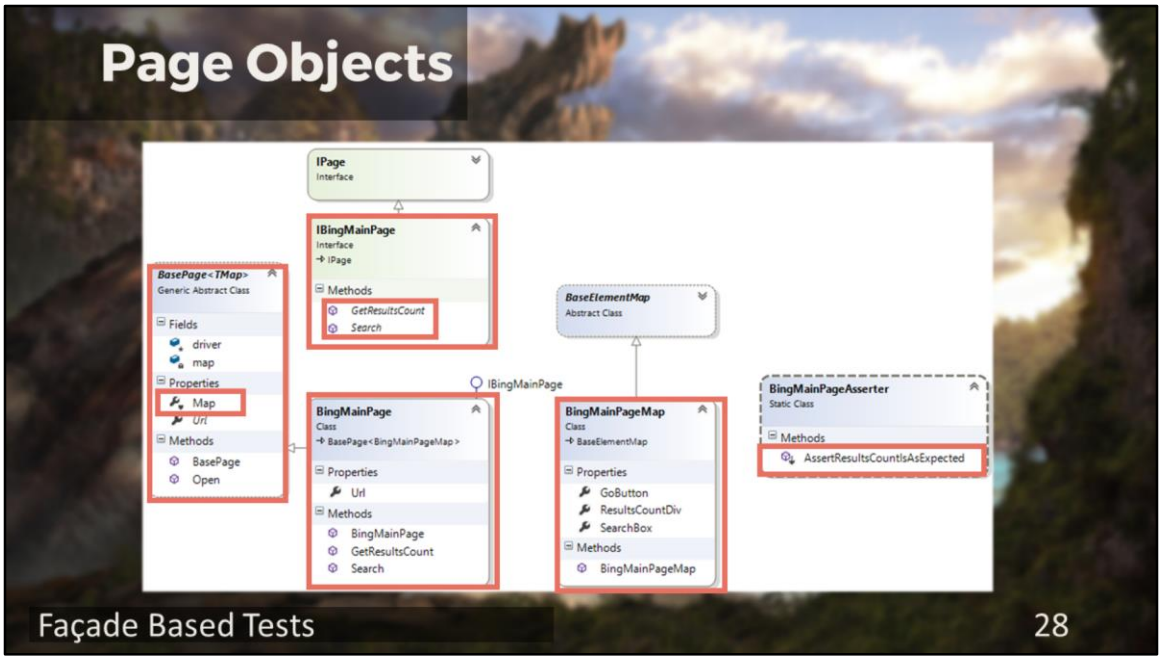
We can have lots of ideas and approaches, but we need to analyse them well and decide which one is the best. Because of that now its is time show you how to use our system in practice. I will use some of the real designs that we evaluated in the past. I will shortly explain the specifics of each one of them and then I will assign ratings for each level described in our assessment system. Further, I will clarify the reasoning behind my rating decisions.



The initial versions of our tests framework utilized the façade design pattern. This is the first design that we are going to evaluate through the proposed assessment system. **A facade is an object that provides a simplified interface to a larger body of code, such as a class library. It makes the software library easier to use and understand, is more readable, and reduces dependencies on external or other code.**



There are not 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. We had to use regions inside to separate the different parts of the code. Regions let you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. As depicted in the image, in the Billing façade, four different regions were used for separating the element map properties, the private fields, and the rest of the methods.

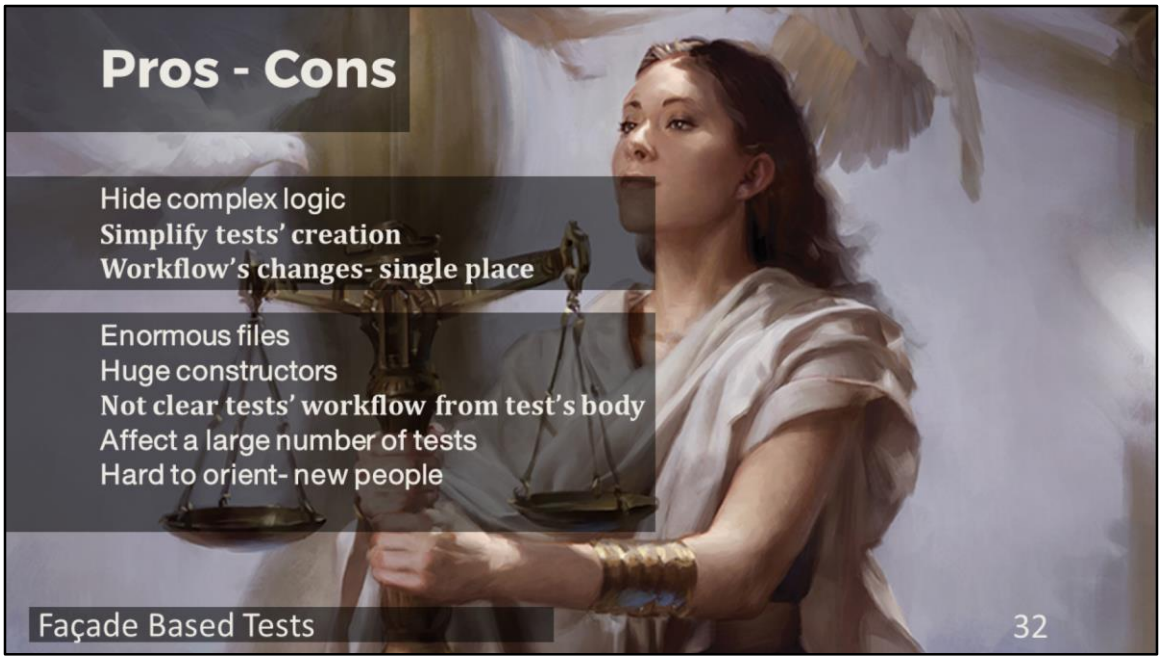


We use a slightly modified version of the pattern. As you can see in the class diagram, the page holds a reference to the element map through the Map property, inherited from the base page class. The page implements a specific interface that defines what actions it should be able to do. The assert methods 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.

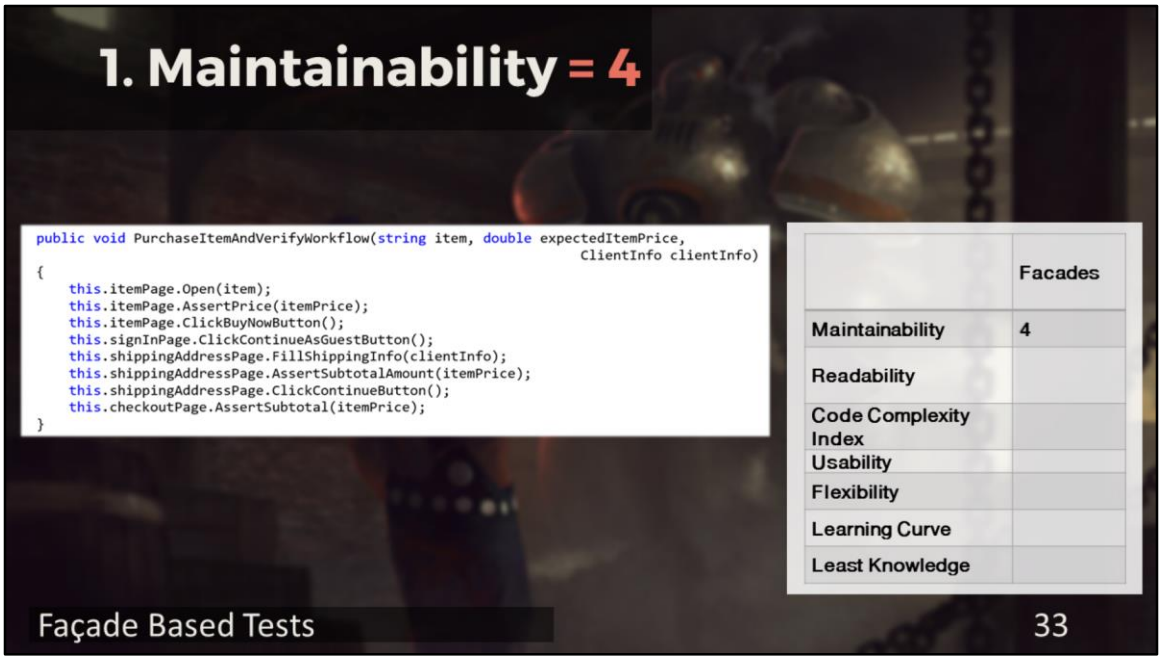
This is the code of our shopping cart facade responsible for creating purchases. We decided to use the façade design pattern in a little different way. It combines the different pages' methods to complete the wizard of the order. If there is a change in the order of the executed actions I can edit it 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.

This is a sample usage of the façade in tests. First we need to initialize all required parameters. After that you simply call the main workflow's method.

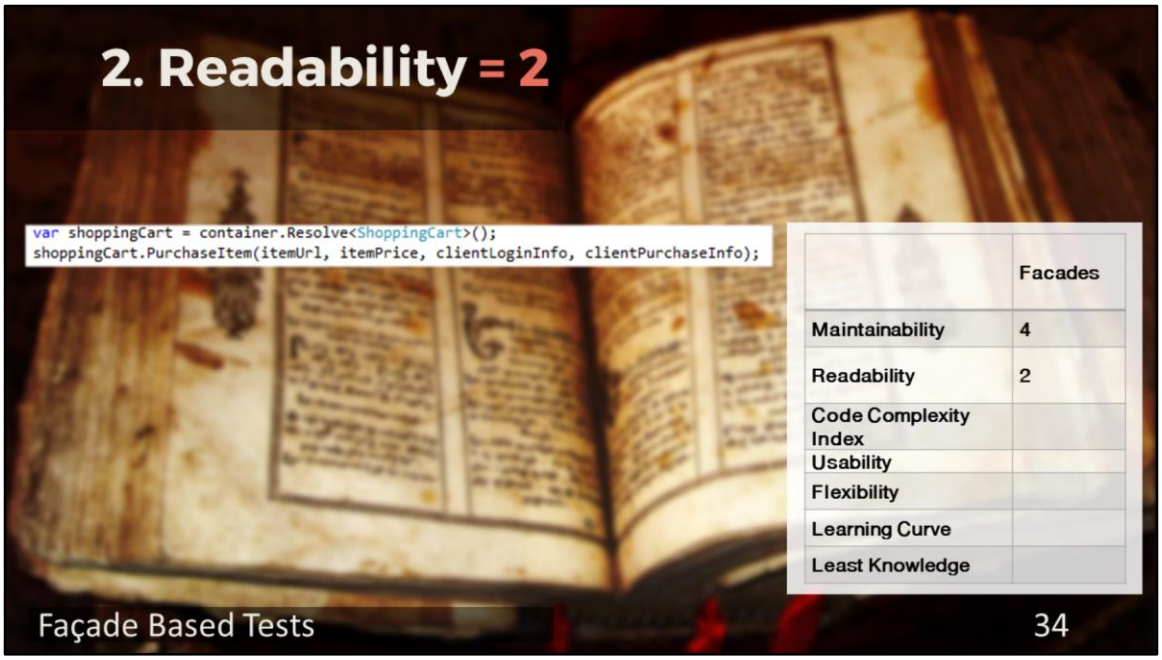
Here is the 7th step of the hero's journey. It is all about that the hero learns the rules of his new world. He meets friends and comes face to face with foes.
The analogy here with the talk is that after we came up with some new idea or design. We need to list all of the great things about it that can help us and all of the bad stuff (the foes) that can harm us during the time.



The Pros are that the facades hide the complex tests' logic. Simplify the tests creation and the workflow's changes happen in a single place.
The Cons are that their files are enormous in size and have huge constructors. The tests' workflow is not clear directly from the tests' body. A change in the façade can affect a large number of tests. Finally, it is hard for new people to orient themselves in the large files.

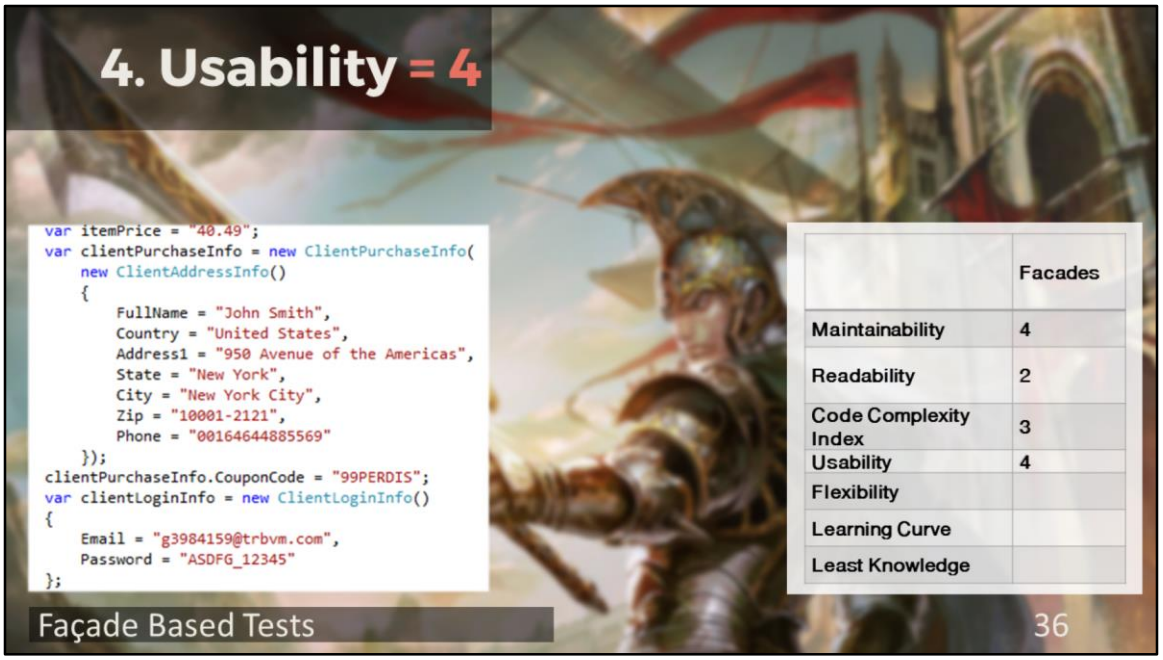


The maintainability is Very Good. The troubleshooting and adding new features to the facades is straightforward. However, the rating is not marked as Excellent because you can easily introduce a regression in the existing tests with small changes in the façade.

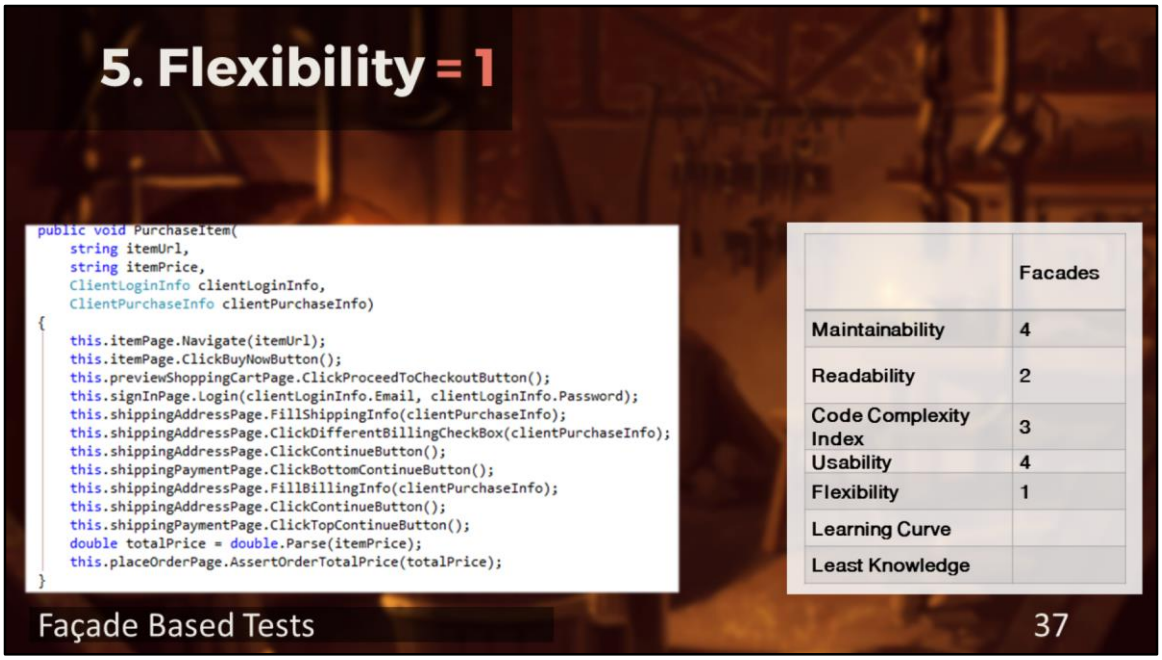


The readability is evaluated as Poor. The tests contain much less code compared to all other solutions. However, as you call only a single method from the façade in the tests, it is not clear to the user what this method is doing under the hood. Further, due to their large sizes, the facades are relatively unreadable and it is not an easy job to find something inside them.

The façade classes have a poor index because they are large in size and they depend on lots of other classes such as other facades and lots of pages. On the opposite, the tests' classes are fairly short in size and call only the façade itself.

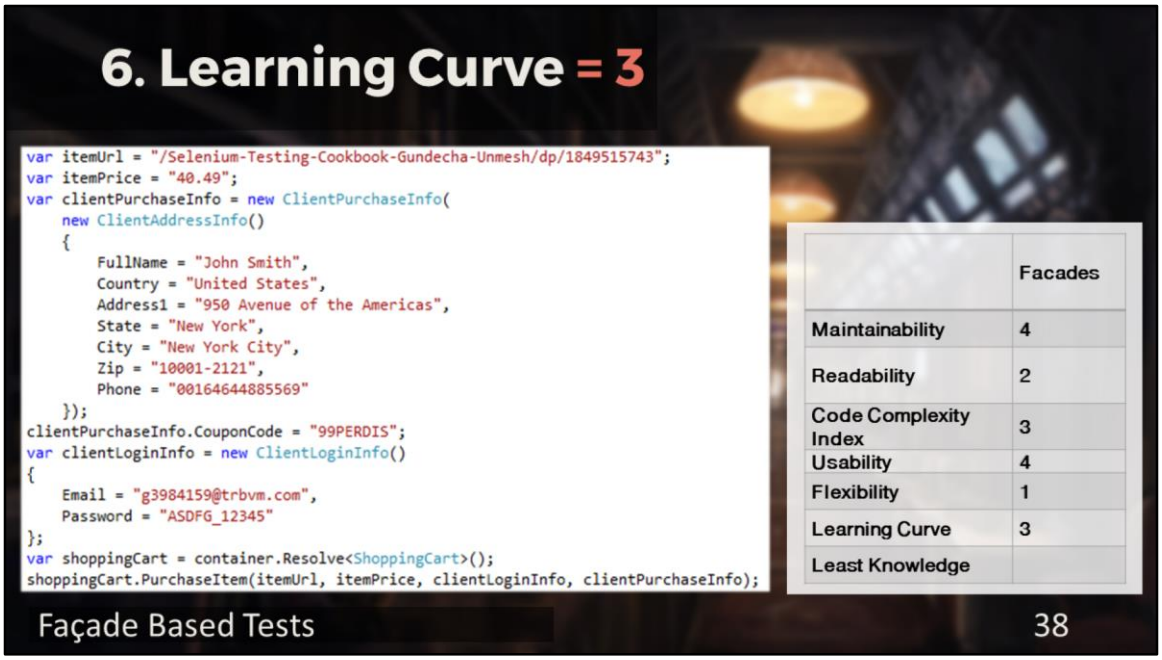


The usability is Very Good. The writing of new tests is straightforward. The rating is not Excellent just because the user has to initialize the tests context upfront.

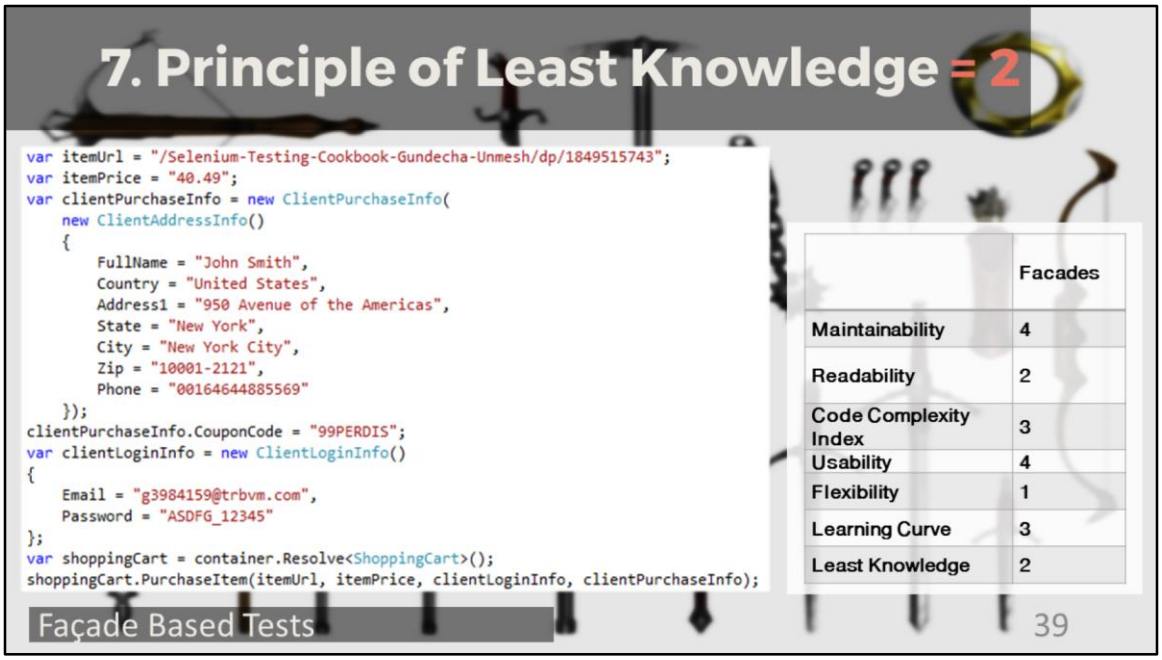


	Facades
Maintainability	4
Readability	2
Code Complexity Index	3
Usability	4
Flexibility	1
Learning Curve	
Least Knowledge	

The flexibility is Very Poor. If you change some of the existing workflows, you will affect all existing tests and possibly create regression issues. If you need to create custom workflow you have to add custom public workflow method which makes the already large façade even bigger. You cannot change or customise the already constructed workflows on a test level.

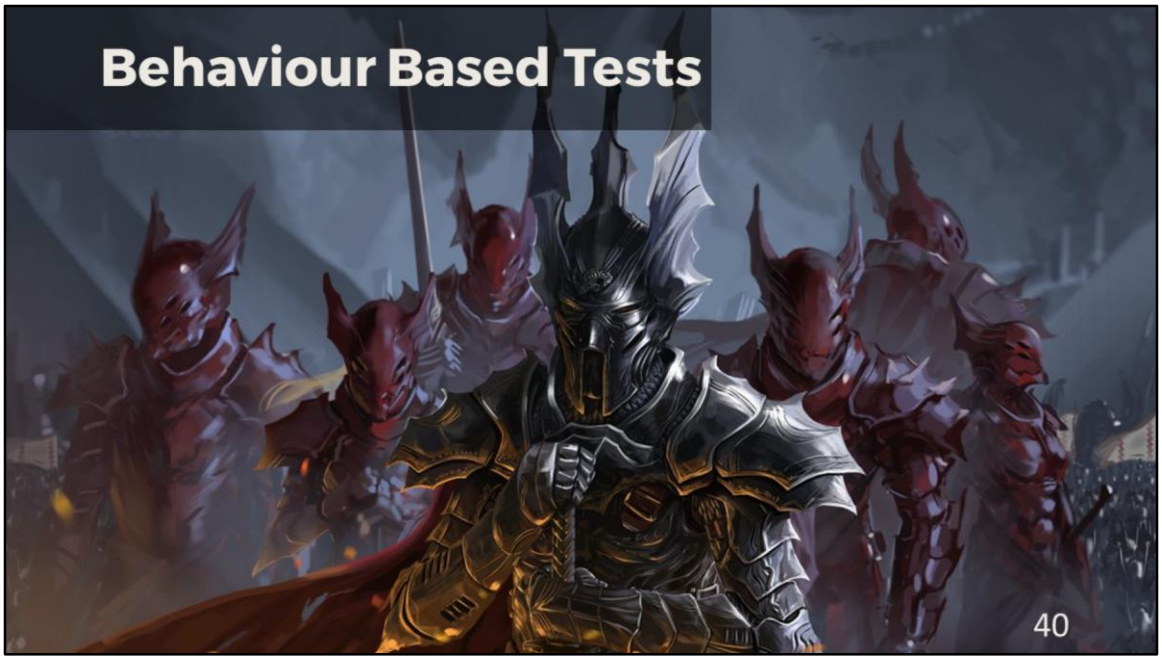


There are two tricky parts with the approach. First, you should know how to initialize the test context correctly. Secondly, if there are multiple public workflow methods, you should be aware which is the most appropriate to call.

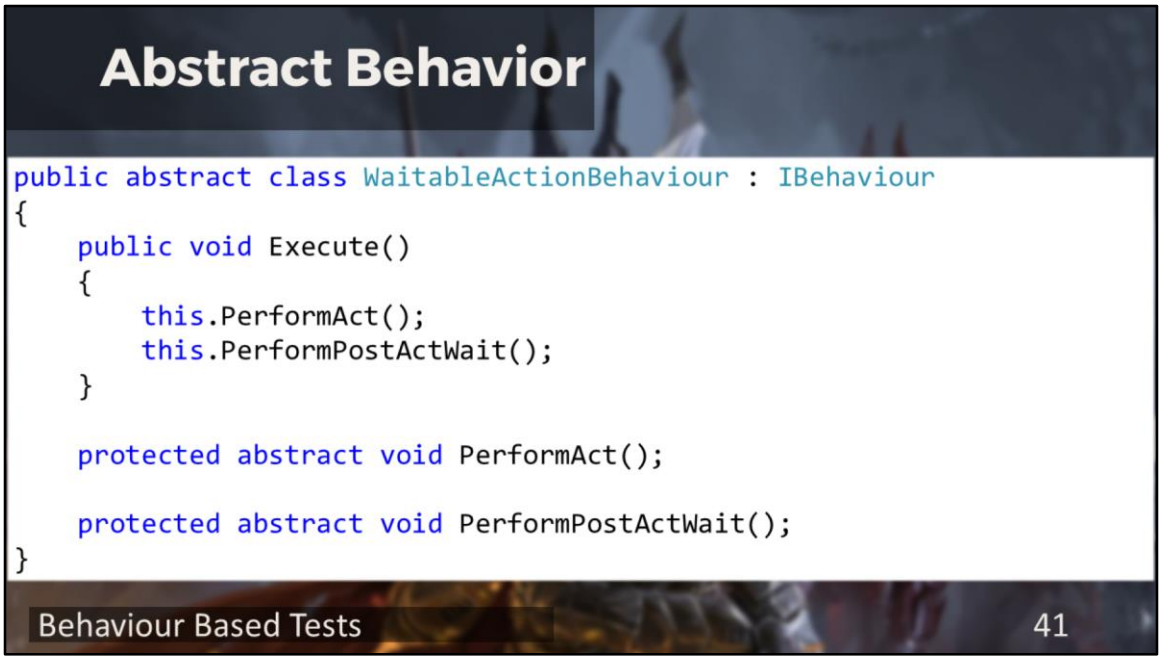


	Facades
Maintainability	4
Readability	2
Code Complexity Index	3
Usability	4
Flexibility	1
Learning Curve	3
Least Knowledge	2

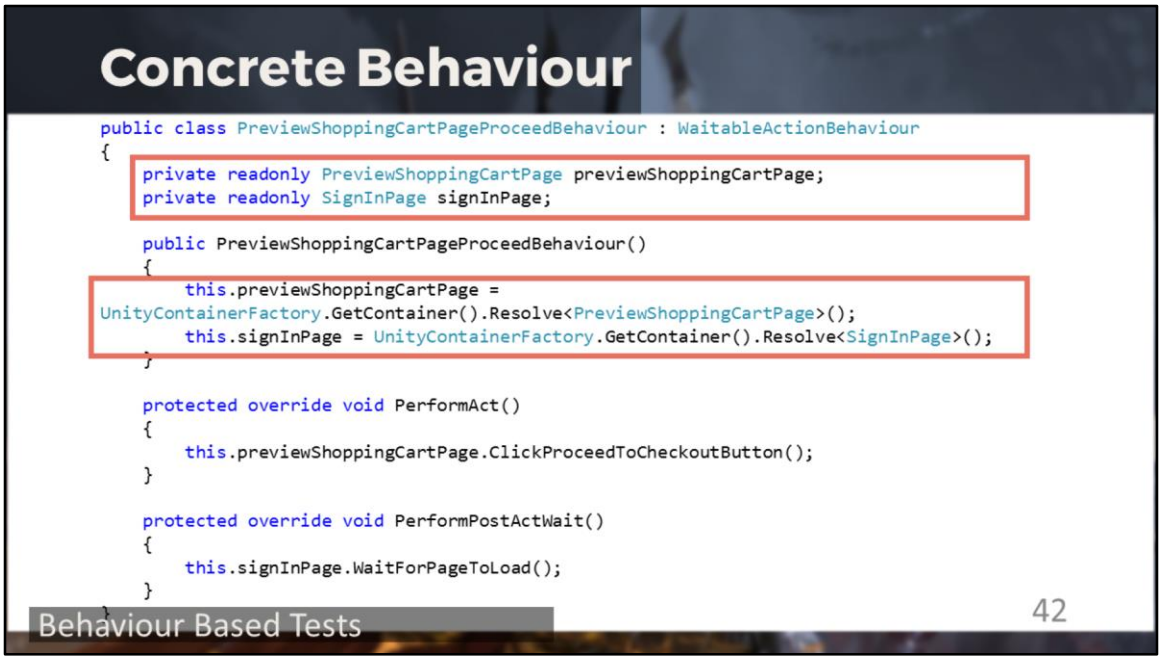
The façade has access to the whole test context which is usually enormous in size. Not all methods need all of the information present in the test context. Because of that the rating is marked as poor.



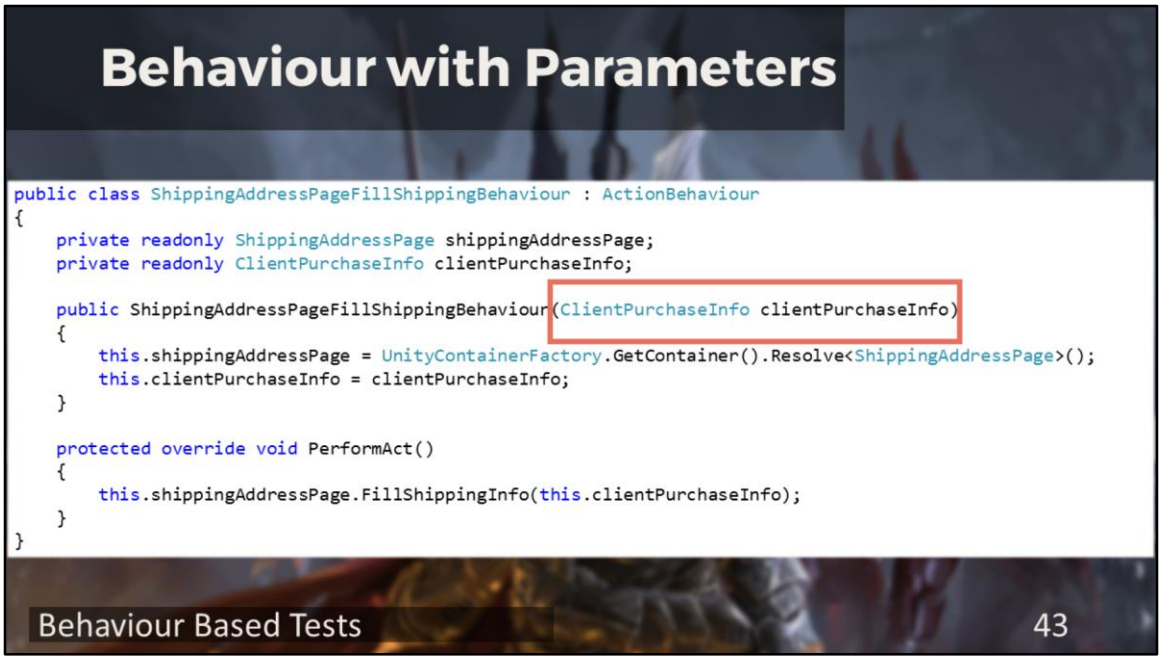
Now it is time to present to you the second design that I am going to evaluate through the assessment system. I named it Behaviour Based Tests.



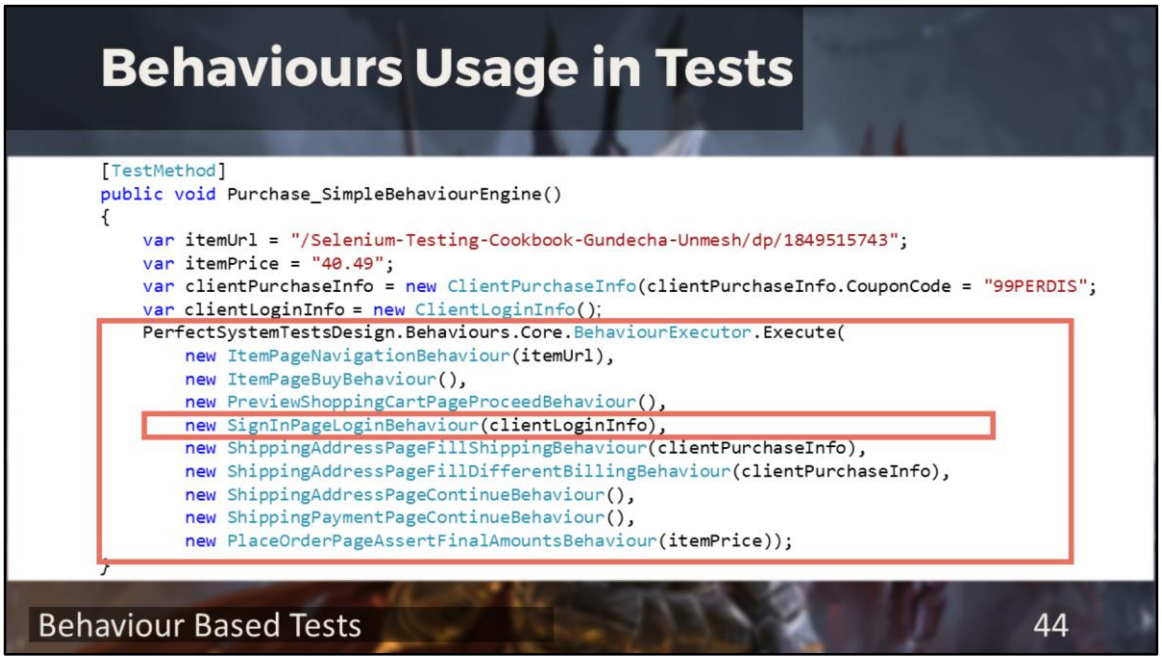
In general, one behaviour executes a page specific workflow- performs actions and waits for a condition. There are different types of behaviours- actions only, asserts only, combining both or adding additional pre/post-wait conditions (wait for the page to load or wait for an element to be visible). On the slide you can find the base class for all behaviours that first execute an action and then wait for something to happen.



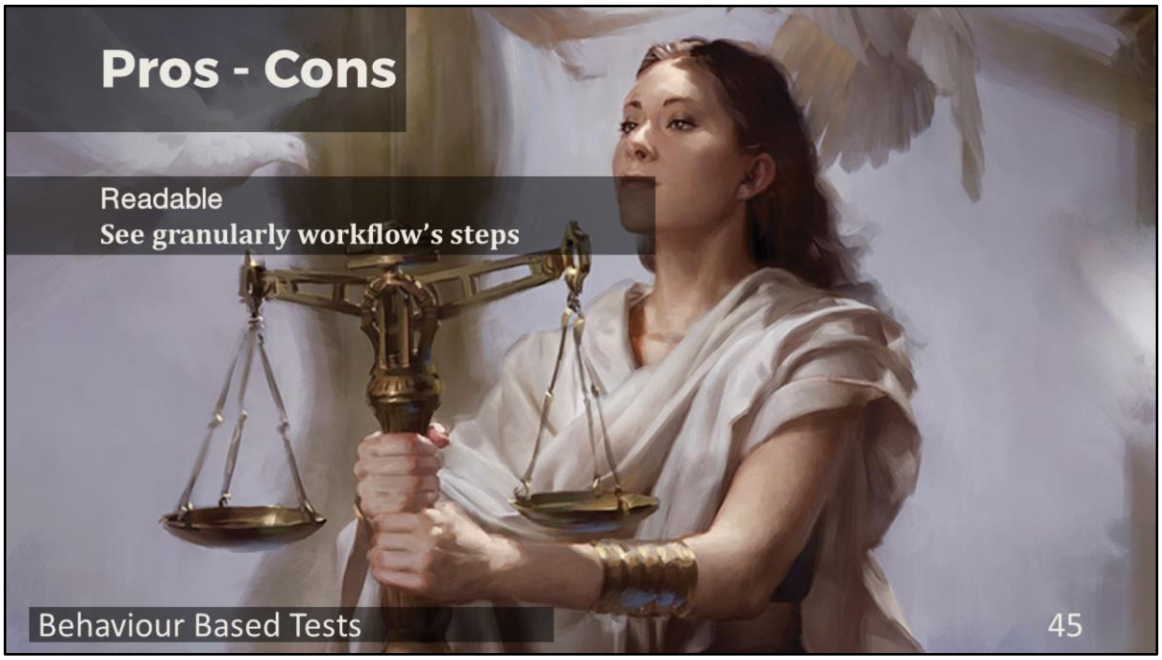
The concrete implementation of a particular behaviour does not have to override all base class' methods (if there are post/pre-wait and assert methods available, the class can override only one of them). The behaviours hold private instances of all dependent pages or other services. There are initialized in the behaviours constructors through Unity IoC container.



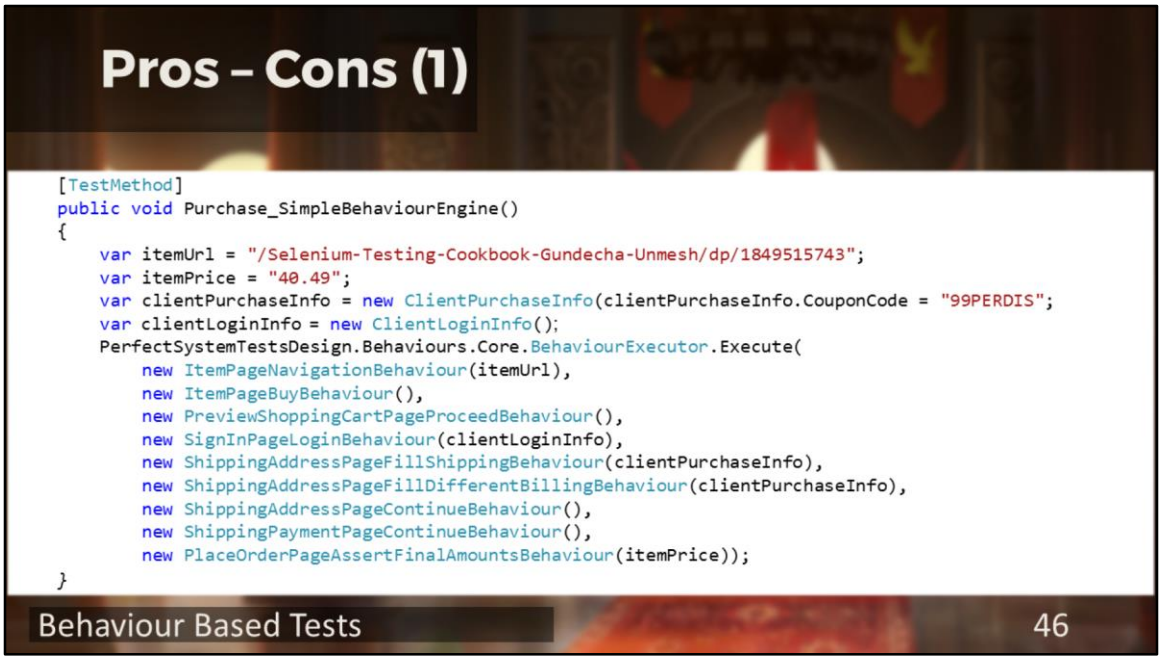
The specific behaviours can accept custom parameters if needed.



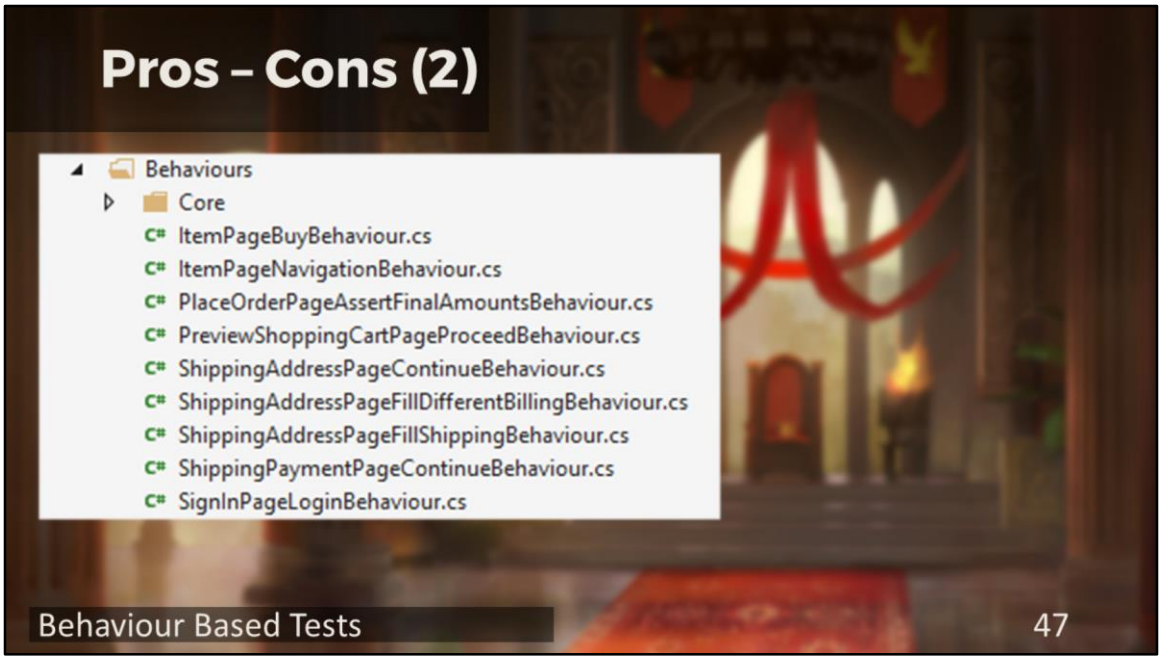
The behaviours are added as a list to a special behaviours executor that is responsible for executing them in the appropriate way. If the behaviour depends on any data, it is passed to its constructor.



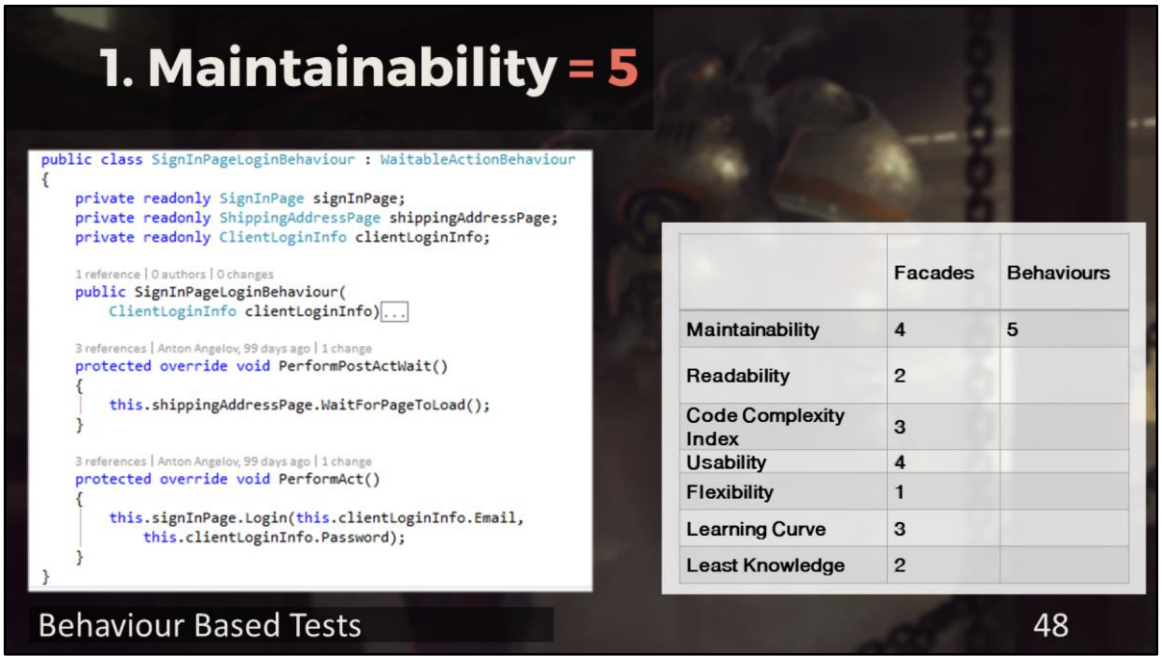
The behaviours are readable. Using them, you can see more granularly the different steps from the workflow.



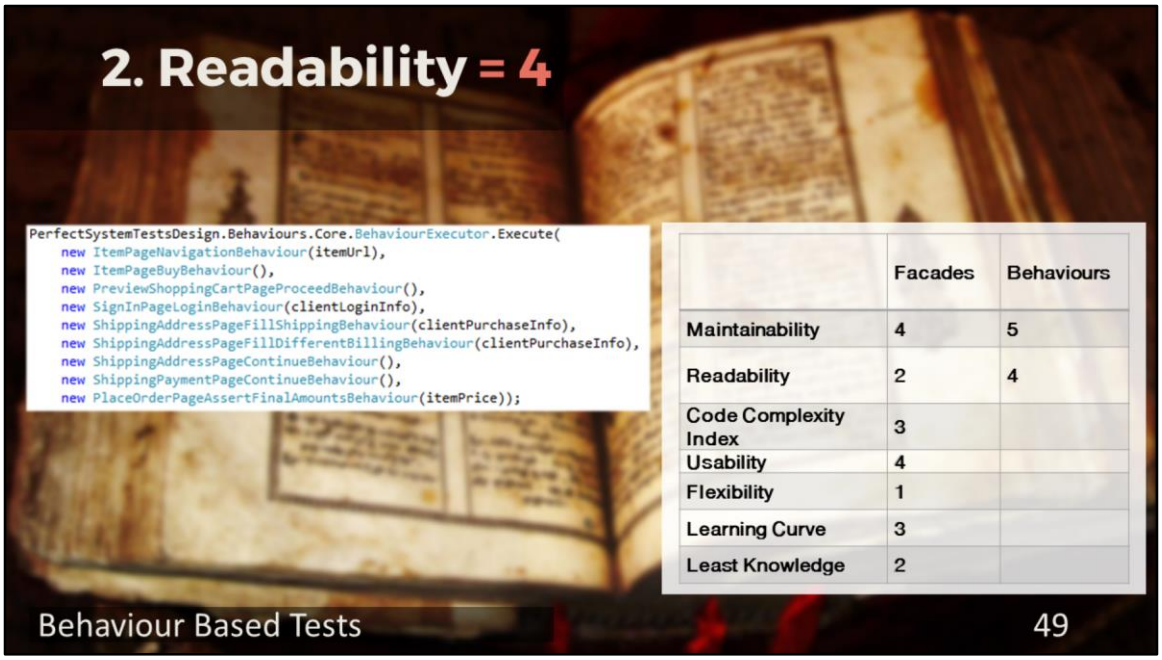
If you do not want to perform some of the asserts mini-workflows you just do not include its behaviour in the list.
The principle of least knowledge is followed because you pass only the required data to the behaviours' constructors.



However, a lot of new classes are introduced. The writing of new tests is slower because now you have to initialize all required behaviours. **The process of test writing is more error prompt because you can mistake the steps' order or assign wrong values to some of the behaviours' parameters.**

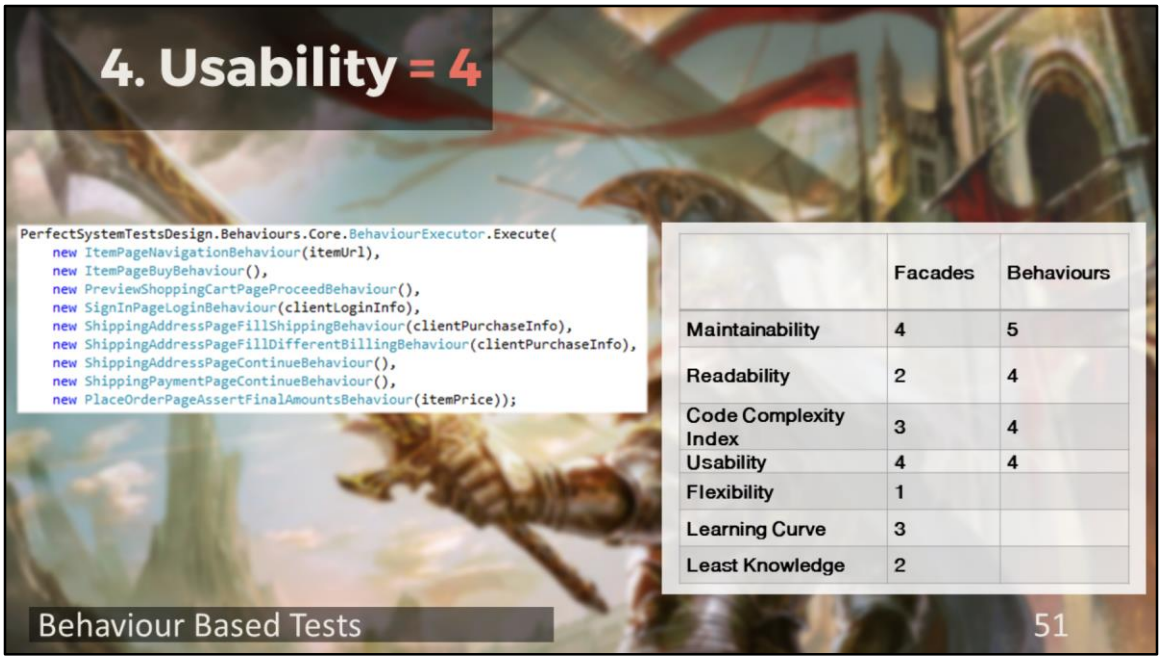


The maintainability is marked as excellent. The behaviours are mini-workflows for some use cases. If the use case should be changed, it is edited only here. As the behaviours are added only on demand, there are not executed for every case. If you fix one behaviour, the change will affect only the tests that are using it.

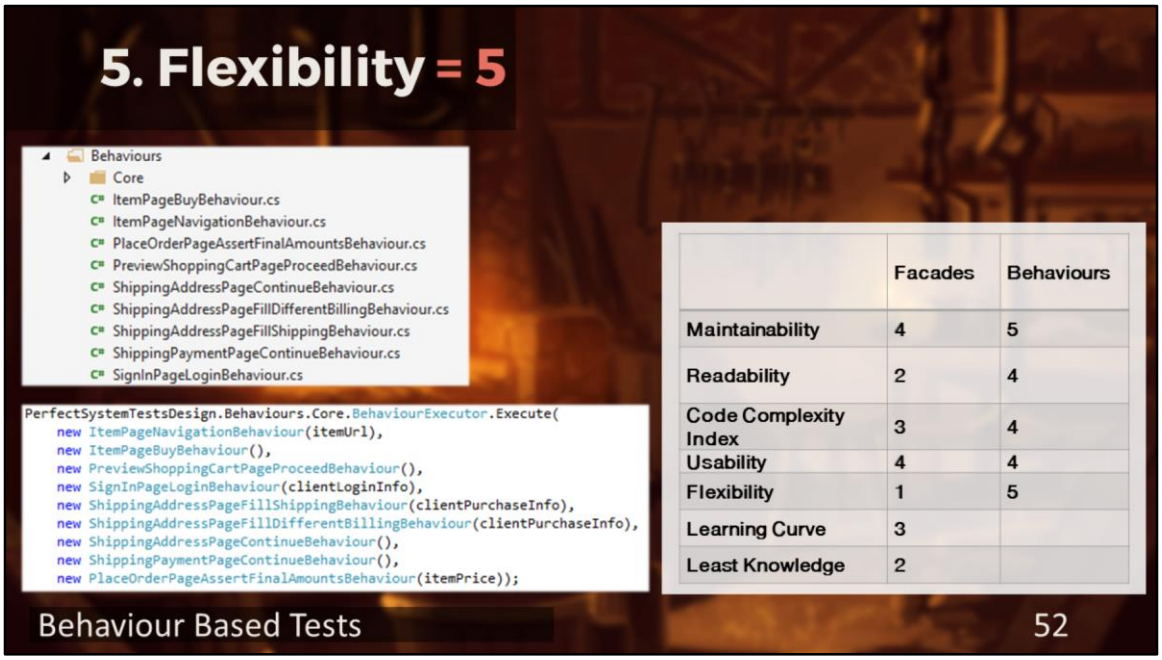


The readability is evaluated as Very Good. As the names of the behaviours are self-describing, you can guess their use case. **Moreover, as you list multiple behaviours to define the bigger workflow, the order of the steps is directly visible to the reader.**

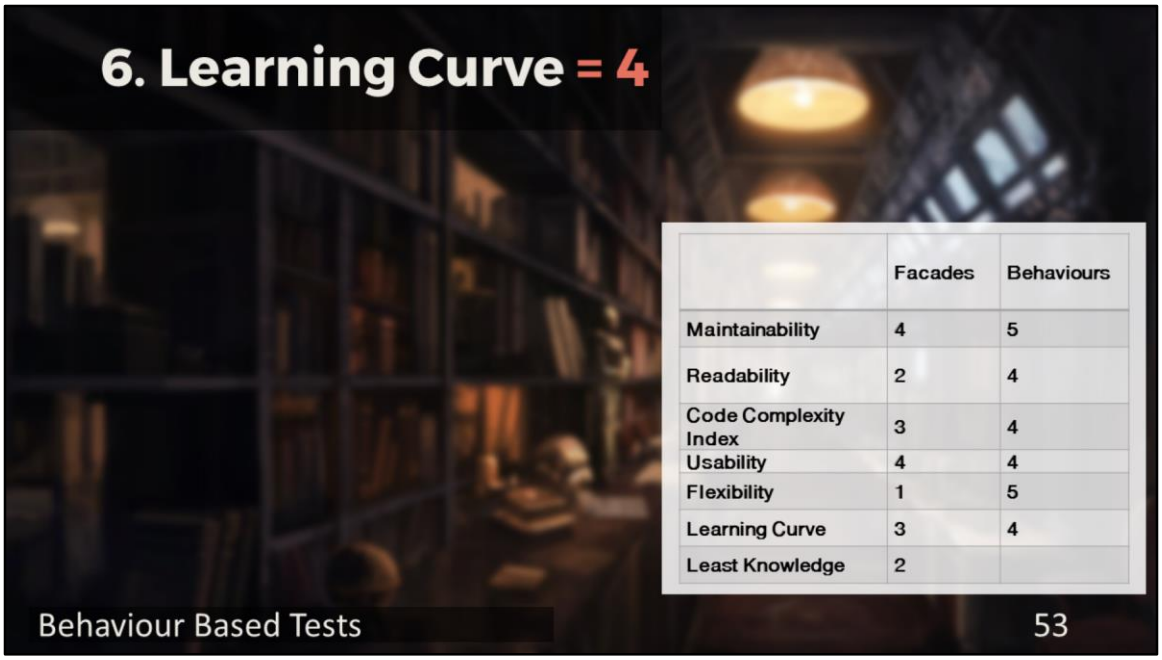
The behaviour classes are small and simple. However, the tests classes are more complex because you need to initialize the whole behaviours' execution chain.



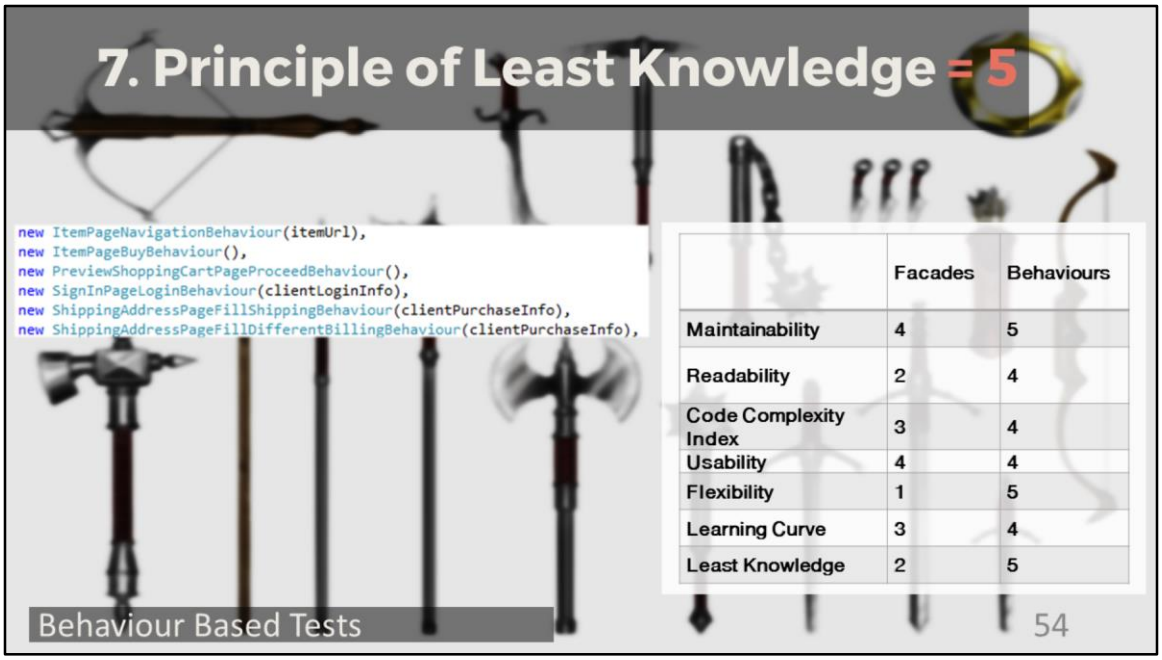
The test framework API is not so complicated to be used. However, you should know the exact name of the behaviours that you want to specify in the large workflow. Moreover, you should be aware of their correct order. Mistaking the order of some of the steps is possible. The writing effort is because here, you need to initialize multiple new classes.



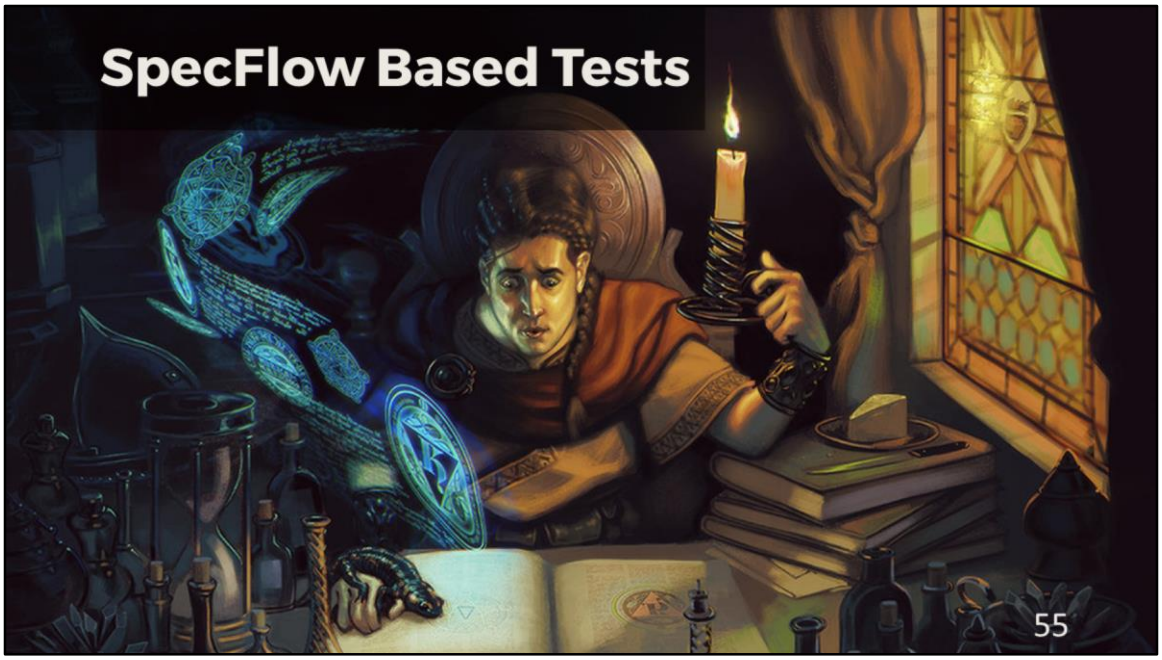
If you want to skip some of the optional mini-workflows, you just don't need to add them to the executer's chain. The same is valid if you have to add some custom mini-workflow that is valid only for a small limited amount of use cases. You just need to create the behaviour and add it to the list of behaviours for this particular case.



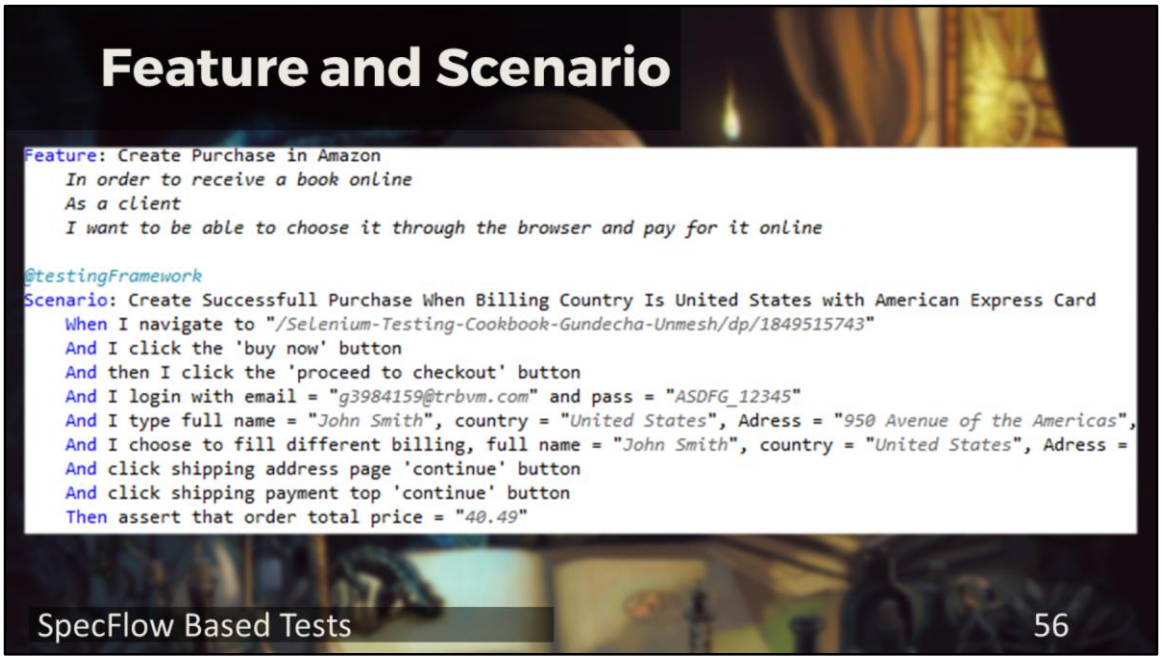
The learning curve for the test framework API is average because the user should know the exact order of the behaviours.
Moreover, should be familiar with the correct names of all behaviours and which exact implementation want to call. **There might be more than one implementation where it is slightly different.**



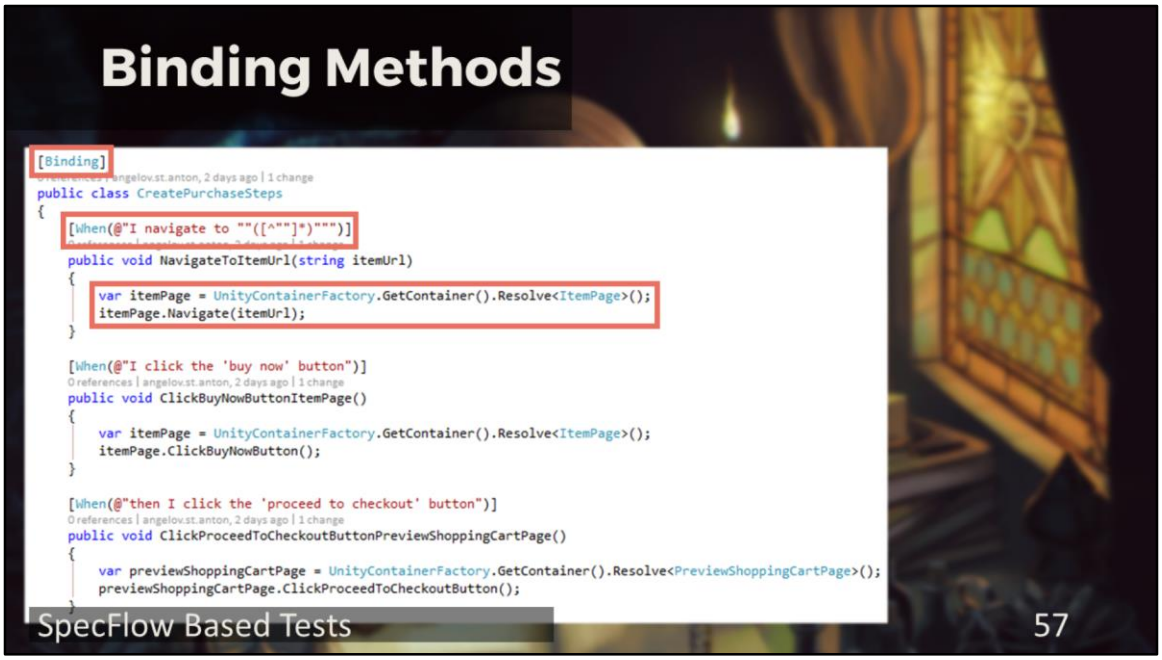
You pass only the required parameters to the concrete behaviours. So the rating is marked as excellent.



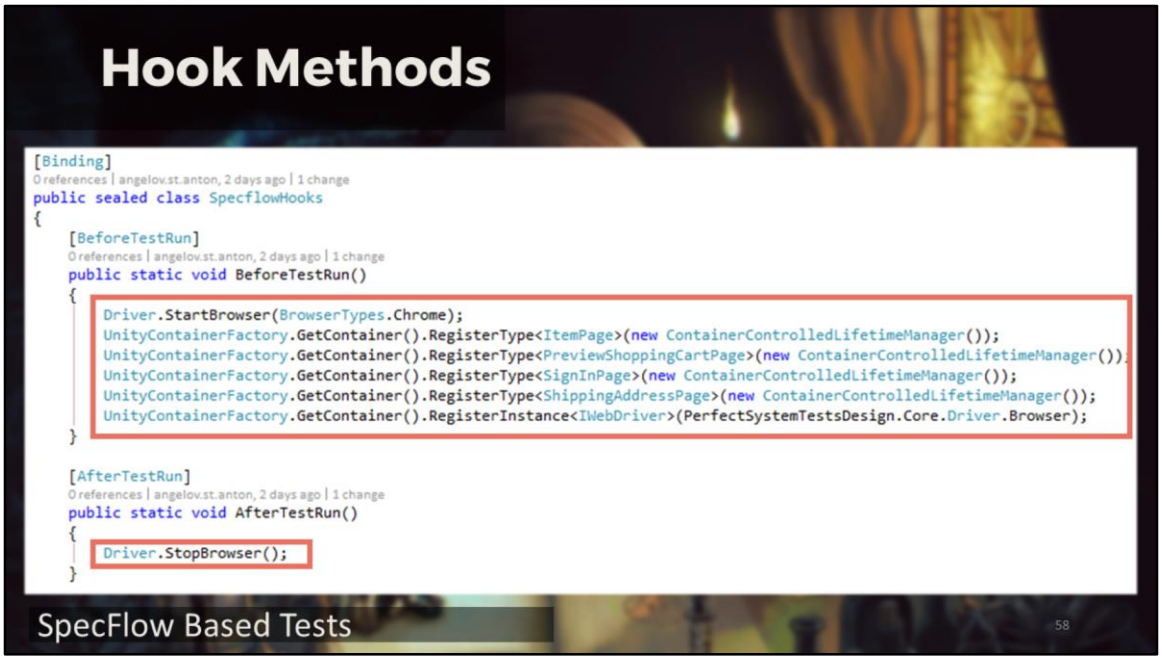
The next design that we will evaluate through the system is a design where the tests are written using SpecFlow.



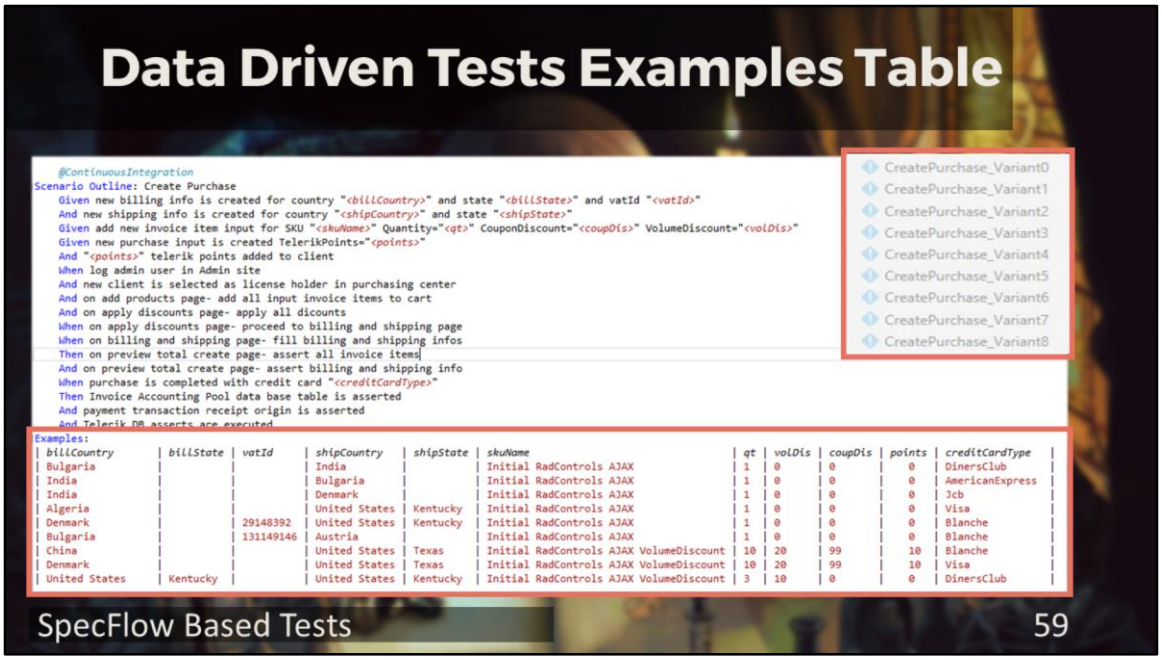
The SpecFlow uses the Gherkin DSL to describe the behaviour of the system using human-readable syntax. It uses the so-called specifications where the test scenarios are described through Gherkin sentences. On build, the DSL is compiled to MSTest tests.



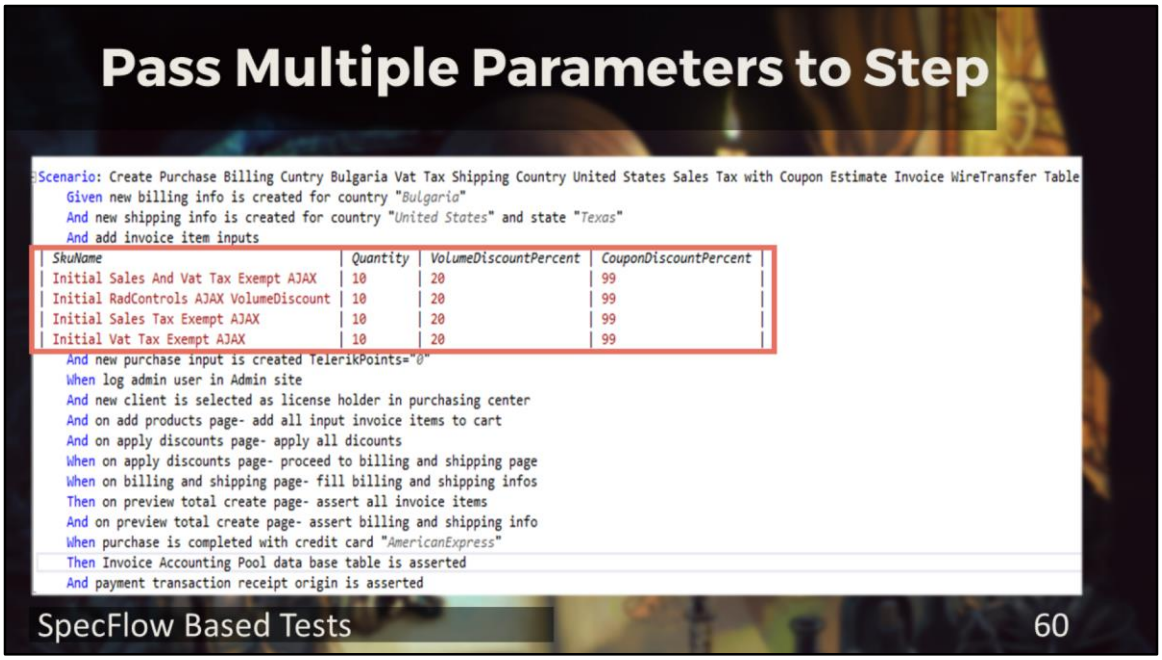
Another thing you have to do is to define binding methods for every sentence used in your scenarios. Otherwise, the tests will fail. These bindings are defined in standard C# classes marked with Bindings attribute. Each step method is marked with step type attribute containing the step's regex pattern. Inside the steps' methods, the pages' logic is called. This way every step defines and executes a small part of the test's workflow. With this approach of tests writing the standard MSTest classes don't exist.



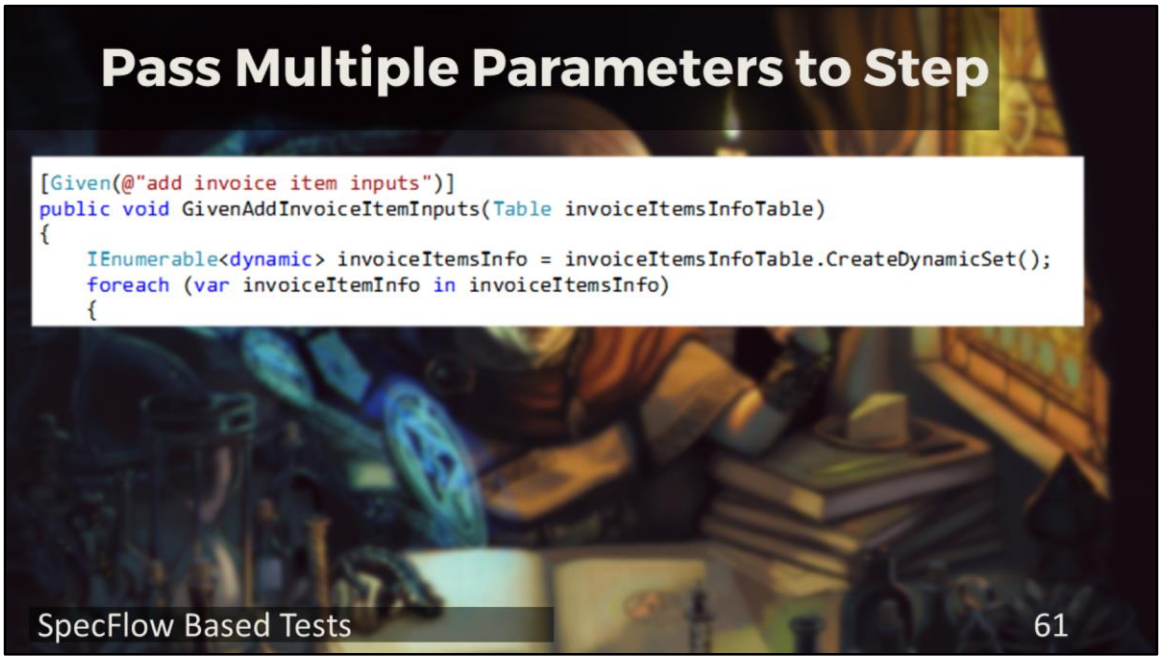
There are so called hooks classes where a different pre/post execution logic can be defined by a run, feature, step block or step level.
For example, here we start a browser before each test and register all needed pages as singletons for the test. After that, we close the browser.



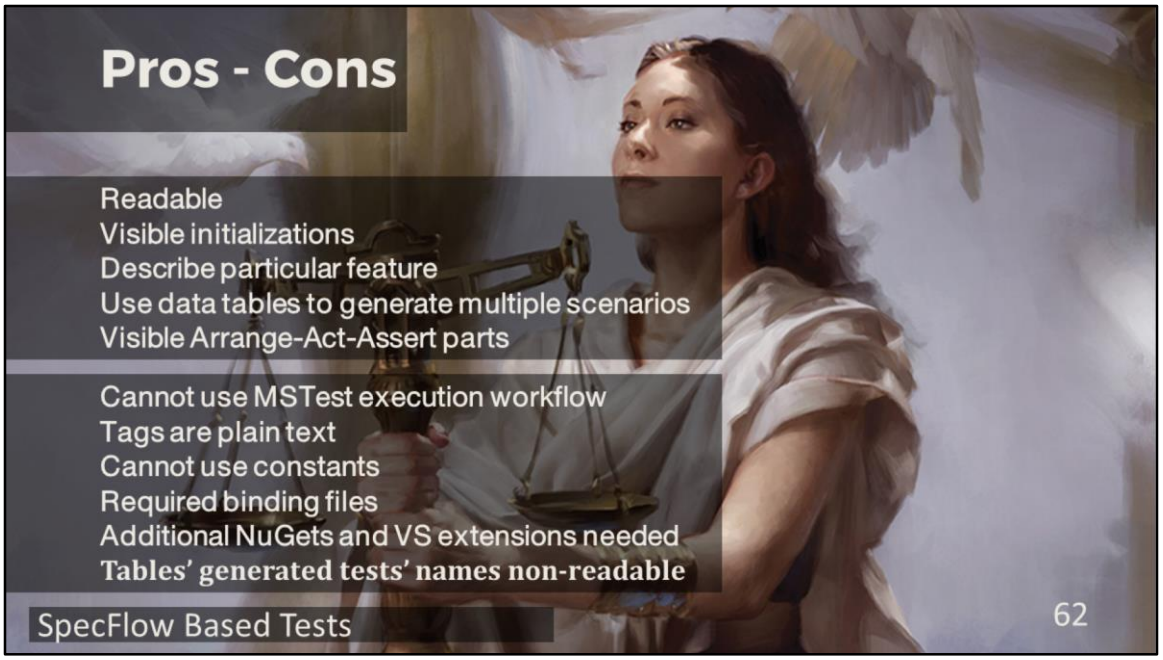
Specflow supports data driven tests through data tables- a new test is generated for every row in the table. Keep in mind that I formatted the table manually. The Visual Studio support for Gherkin is not on the needed level.



This table will pass something like a list of dynamic object to our binding method. By the way it is really implemented using dynamic C# objects.



Here is how we use the SpecFlow's parameters table in tests. As I pointed it creates a dynamic list of objects and we can iterate through them.



Pros

The test scenarios are readable.

The initializations are visible in the tests as they are described as a couple of sentences.

The scenarios describe a specific feature, and there is a description.

Multiple scenarios can be generated through examples data table. This way similar tests are generated where only the input data is different. The differences are more visible to the reader.

The different Arrange-Act-Assert parts of the test are more clear to the user.

Cons

The existing MSTest execution workflow cannot be used, should be rewritten with SpecFlow's hooks.

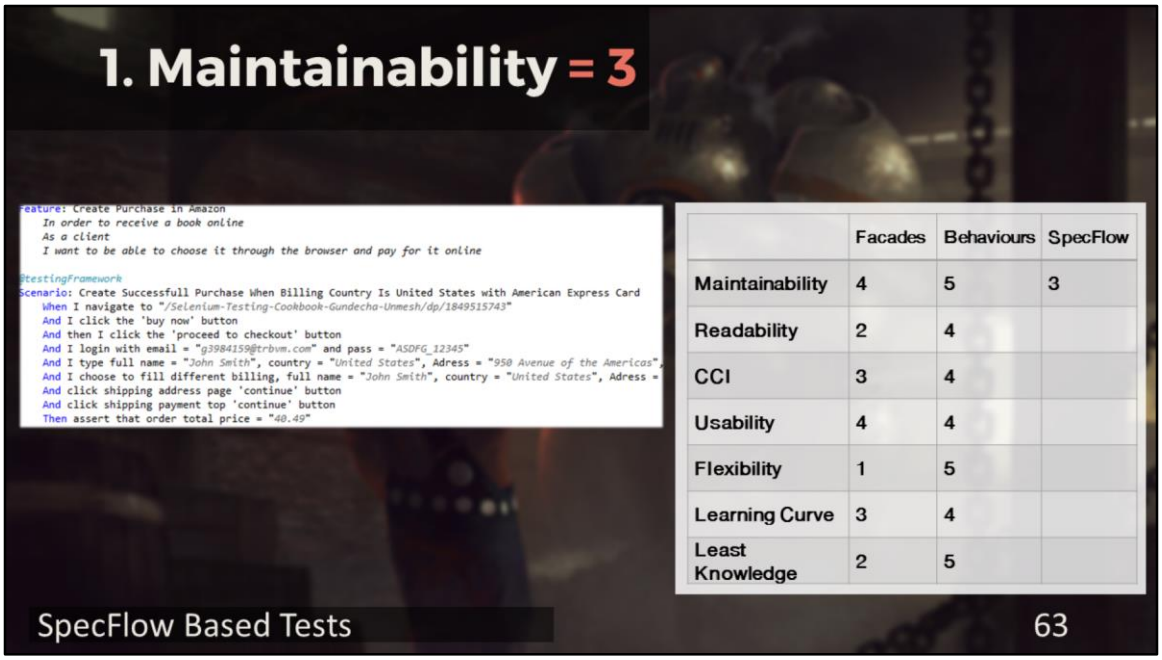
The tags used in the feature files are plain text.

Cannot use constants inside the feature files which means that all input data is hard-coded and cannot be reused.

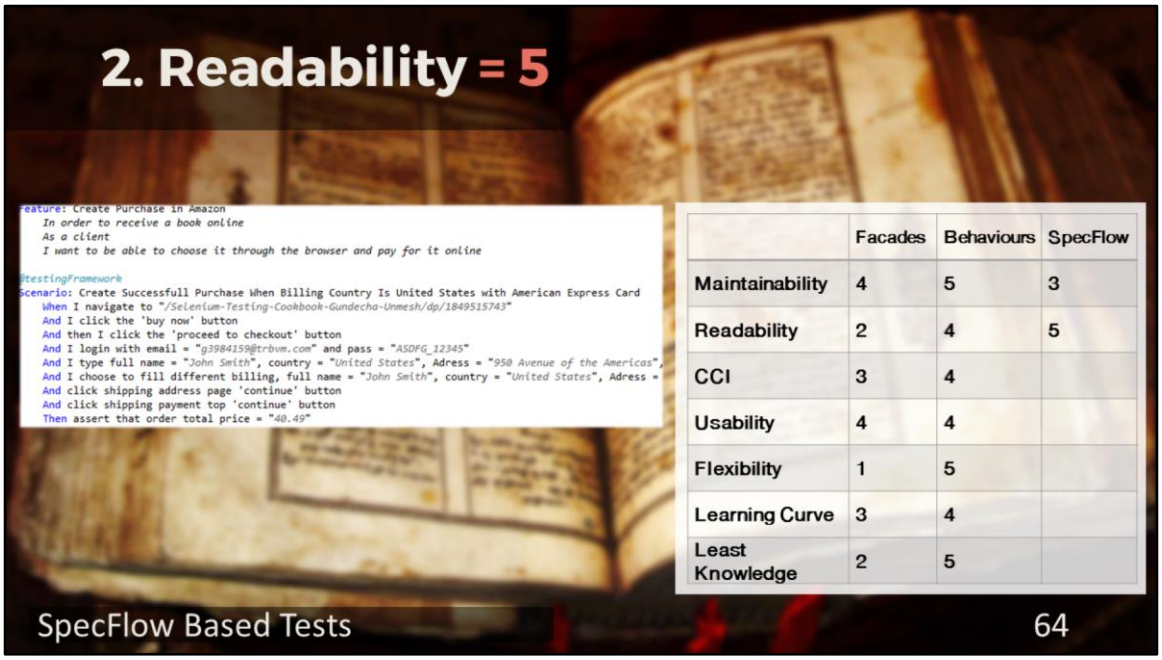
Additional binding files should be present in order SpecFlow to be able to work.

Additional NuGets and VS extensions should be installed.

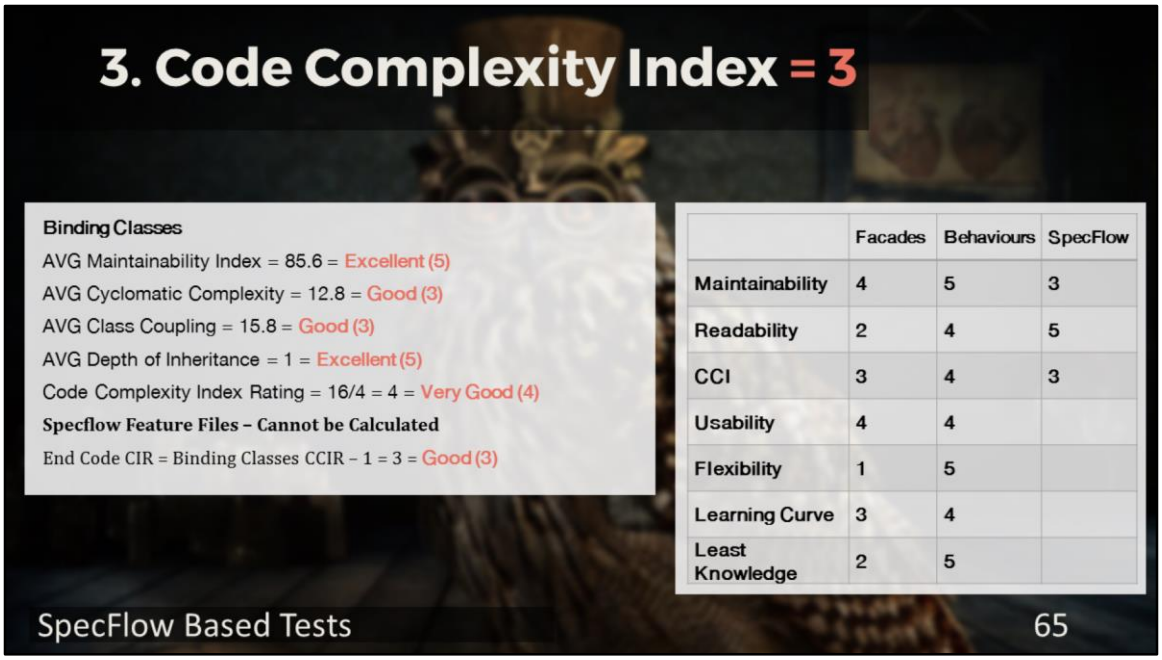
The tests' names generated from the examples' table are non-readable.



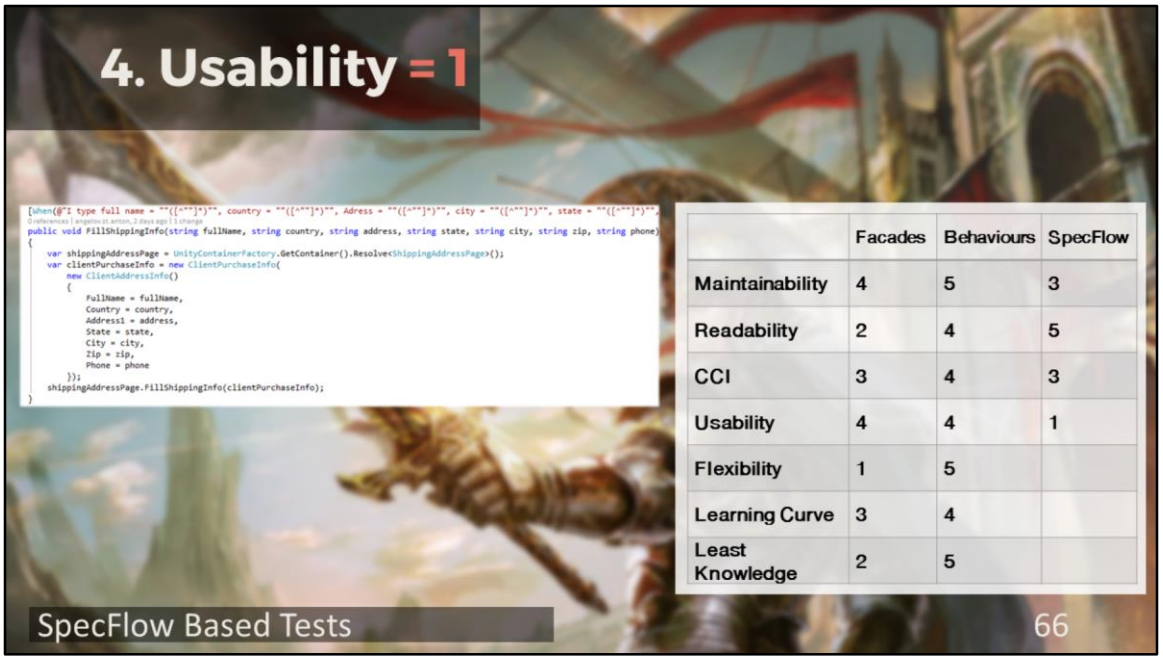
Everything mentioned for the behaviours is applicable for this approach too. However, the rating is decreased because additional binding files exist. Moreover, the user cannot use existing tags and constants in the feature files which leads to hard-coded data and copy-paste development.



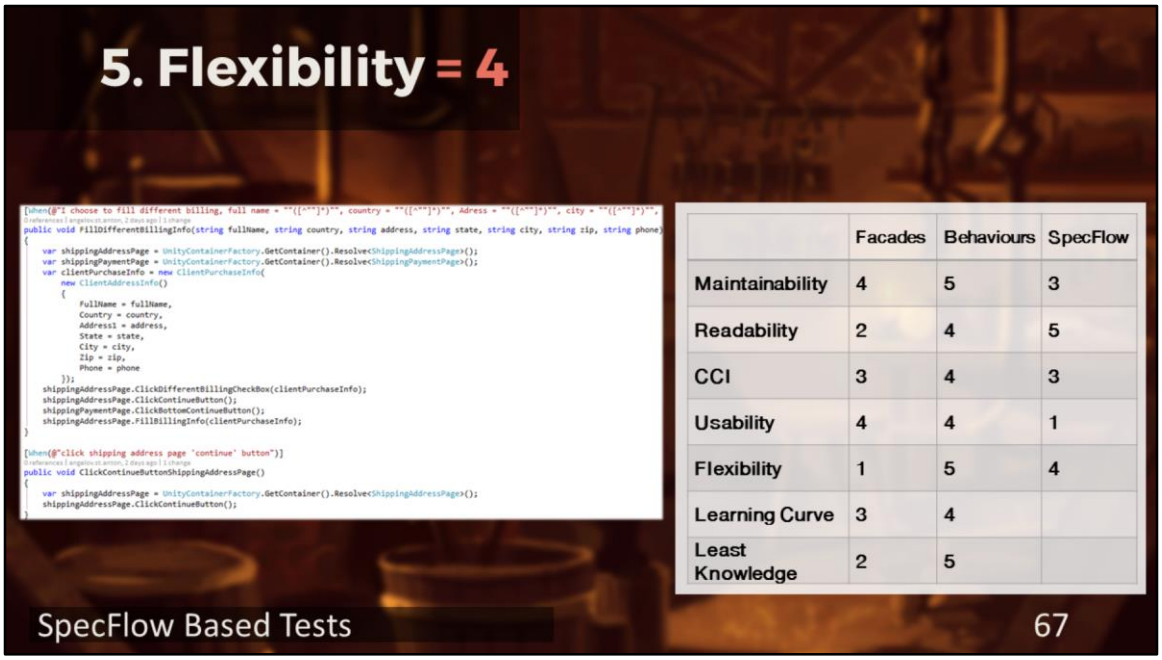
The main advantage of SpecFlow is the readability. All steps are described in human-readable syntax via the Gherkin DSL. Even the base initialize methods are described with a couple of sentences which makes them more meaningful to the user.



The code complexity index here is not entirely accurate because Visual Studio doesn't support the calculation of code metrics for Gherkin files. The index for the binding classes is marked as very good. Probably because there are not any base classes. However, they depend on multiple classes such as pages and behaviours or facades. I think you will agree with me that if we could calculate the metrics for the Gherkin files they weren't going to be very good because of that I decreased the overall rating with one and it is only marked as good.



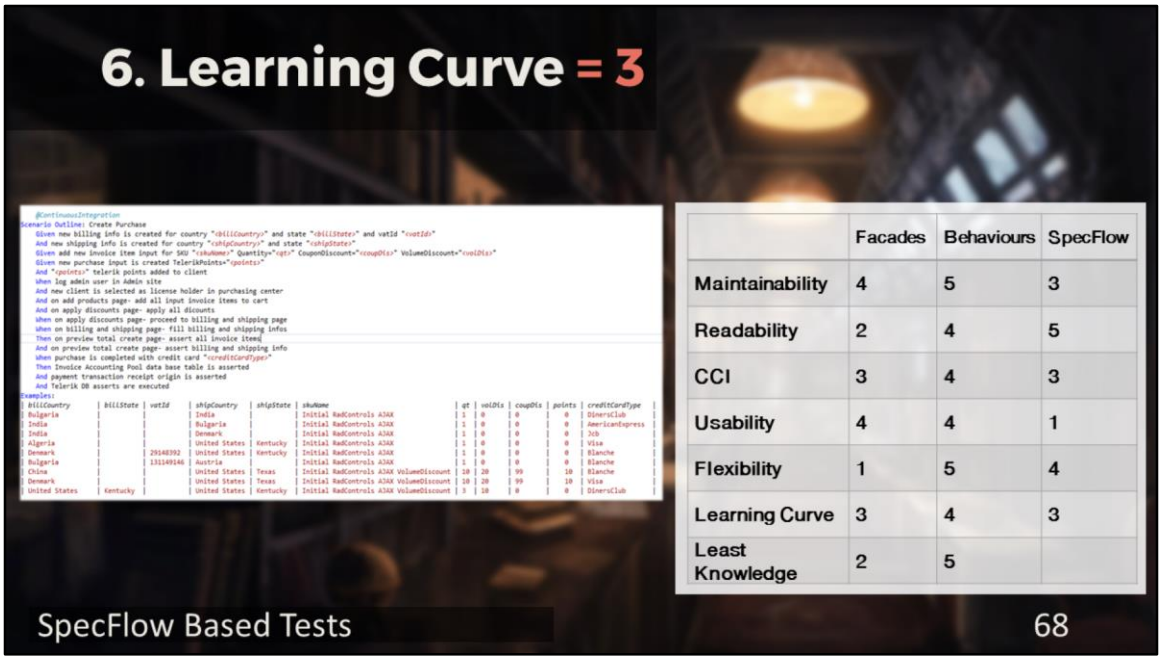
The rating for this parameter is calculated as Very Poor because there are a lot of new classes that should be created before the user can use any steps in his test (assuming that he/she is writing a new test from scratch for a new feature). The SpecFlow's integration with Visual Studio is poor and the suggested steps while writing are not very helpful (IntelliSense). It is a challenge if you need to define a couple of actions with common starting words (you should define different overridden methods in the bindings' classes + use custom regex patterns). If you use examples' table to generate tests you should format it manually if you want to be readable.



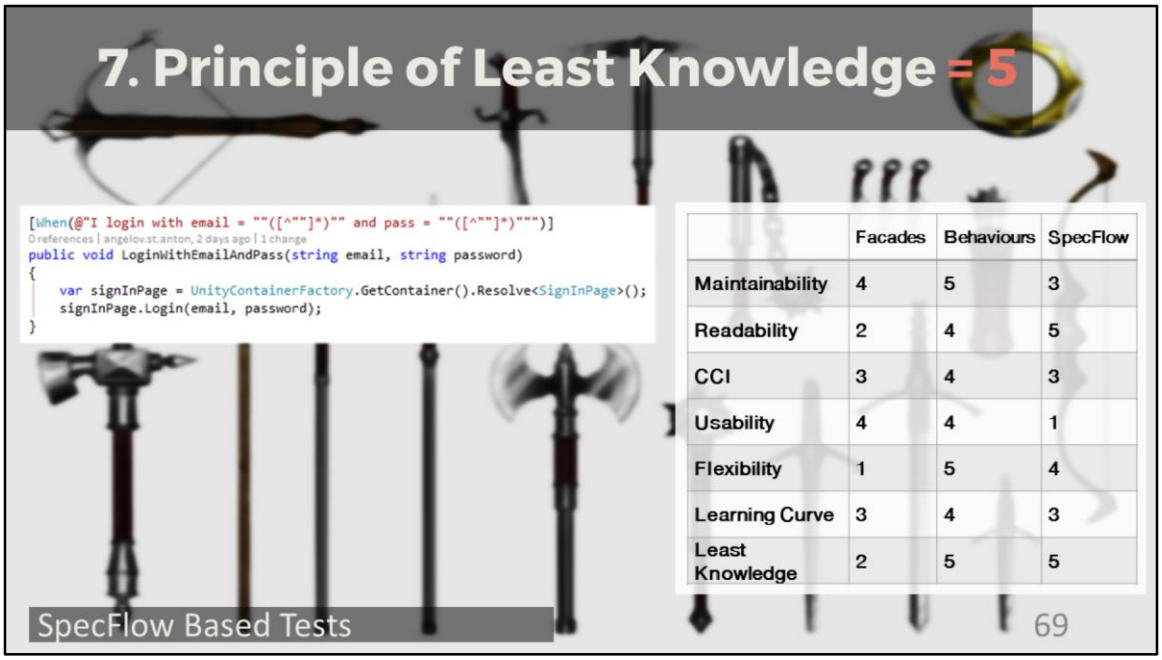

```
[When(@"I choose to fill different billing, full name = ""([^""]*)"", country = ""([^""]*)"", Adress = ""([^""]*)"", city = ""([^""]*)"",
0 references | angelov.st.anton, 2 days ago | 1 change
public void FillDifferentBillingInfo(string fullName, string country, string address, string state, string city, string zip, string phone)
{
    var shippingAddressPage = UnityContainerFactory.GetContainer().Resolve<ShippingAddressPage>();
    var shippingPaymentPage = UnityContainerFactory.GetContainer().Resolve<ShippingPaymentPage>();
    var clientPurchaseInfo = new ClientPurchaseInfo(
        new ClientAddressInfo()
        {
            FullName = fullName,
            Country = country,
            Address1 = address,
            State = state,
            City = city,
            Zip = zip,
            Phone = phone
        });
    shippingAddressPage.ClickDifferentBillingCheckBox(clientPurchaseInfo);
    shippingAddressPage.ClickContinueButton();
    shippingPaymentPage.ClickBottomContinueButton();
    shippingAddressPage.FillBillingInfo(clientPurchaseInfo);
}

[When(@"click shipping address page 'continue' button")]
0 references | angelov.st.anton, 2 days ago | 1 change
public void ClickContinueButtonShippingAddressPage()
{
    var shippingAddressPage = UnityContainerFactory.GetContainer().Resolve<ShippingAddressPage>();
    shippingAddressPage.ClickContinueButton();
}
```

	Facades	Behaviours	SpecFlow
Maintainability	4	5	3
Readability	2	4	5
CCI	3	4	3
Usability	4	4	1
Flexibility	1	5	4
Learning Curve	3	4	
Least Knowledge	2	5	

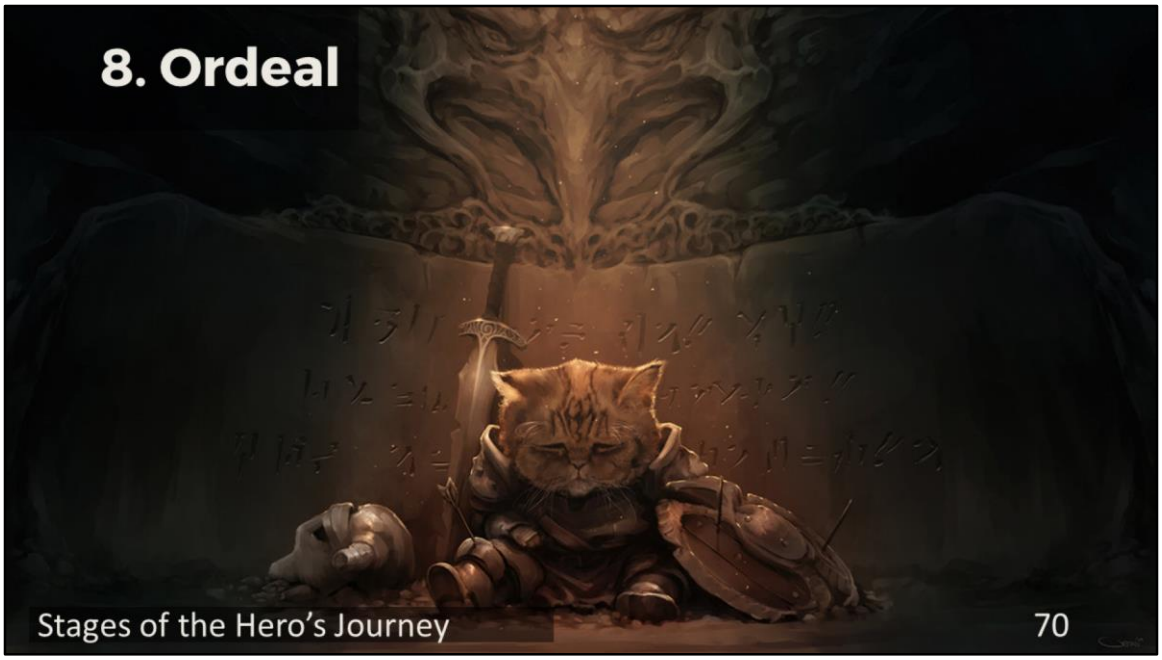
The rating is only good because in order the SpecFlow's API to support additional steps you need to create wrapper methods in the bindings' classes with custom regex expressions.



I guess it will be harder to write new tests compared to the approach of using only page objects, especially if there isn't existing tests using the SpecFlow's sentences' steps.

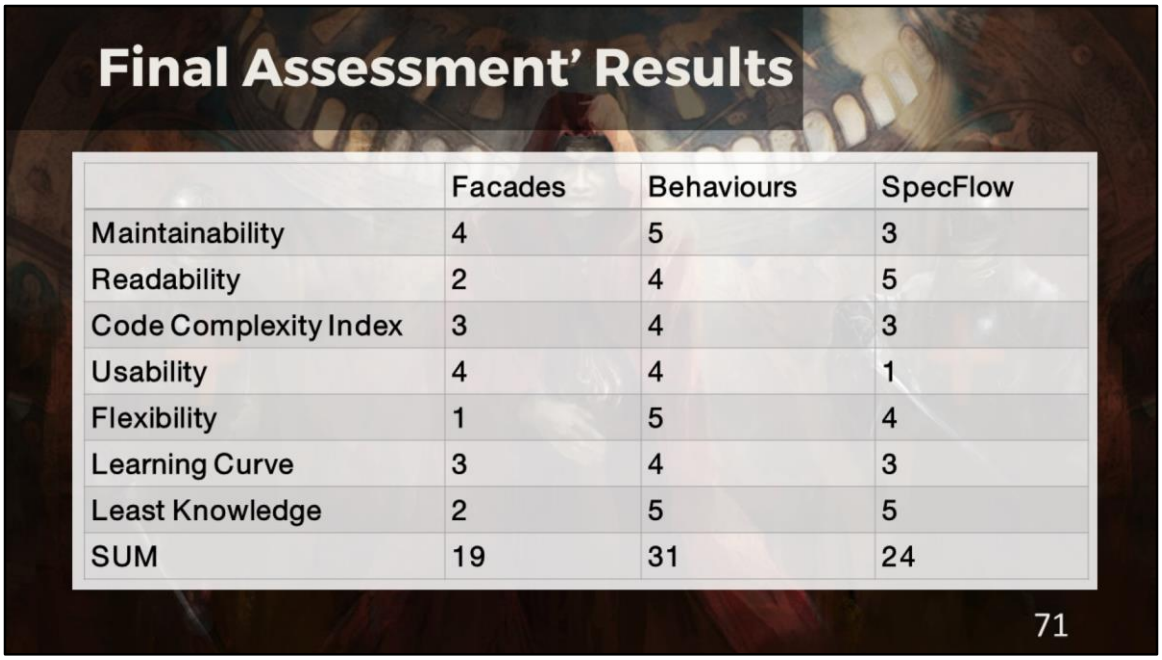


You pass only the required parameters to the concrete binding methods. So the rating is marked as excellent.



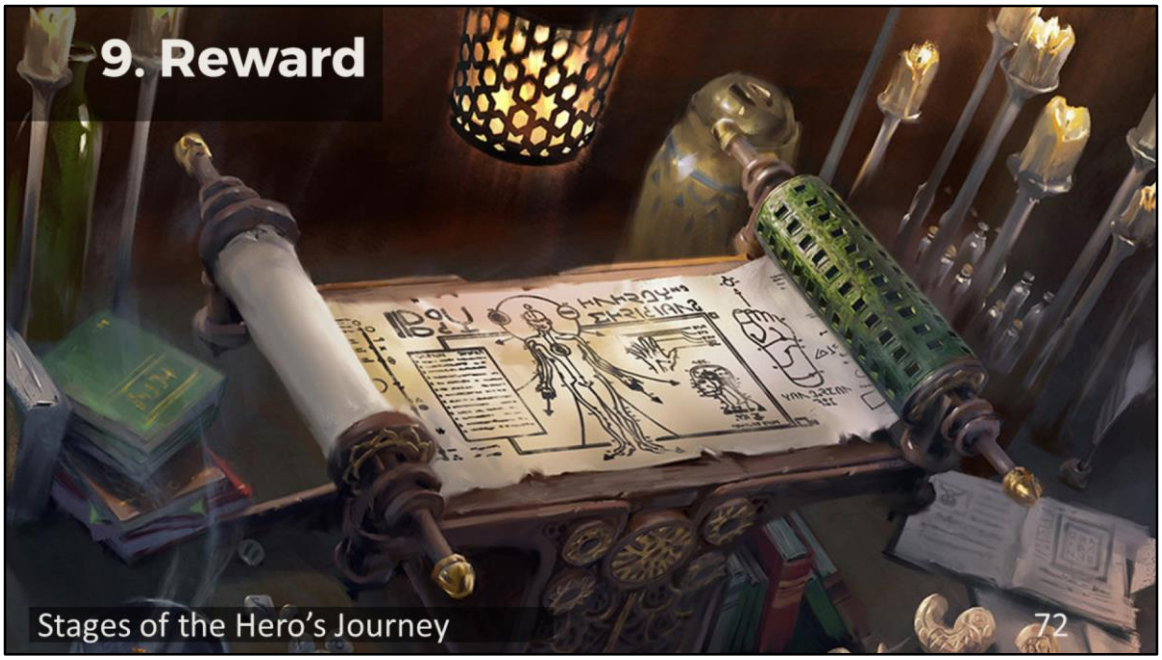
The 8th step from the hero's journey is called Ordeal. And the hero experiences a major obstacle.

Before the usage of the system, this cat was me. I struggled with the decision which of my ideas to apply next. I was always wondering is it going to help us or not? But through the aid of the assessment system, we have the final results' table, and it can give us all of the answers that we need to make our final decision. Without further ado here are the final results.

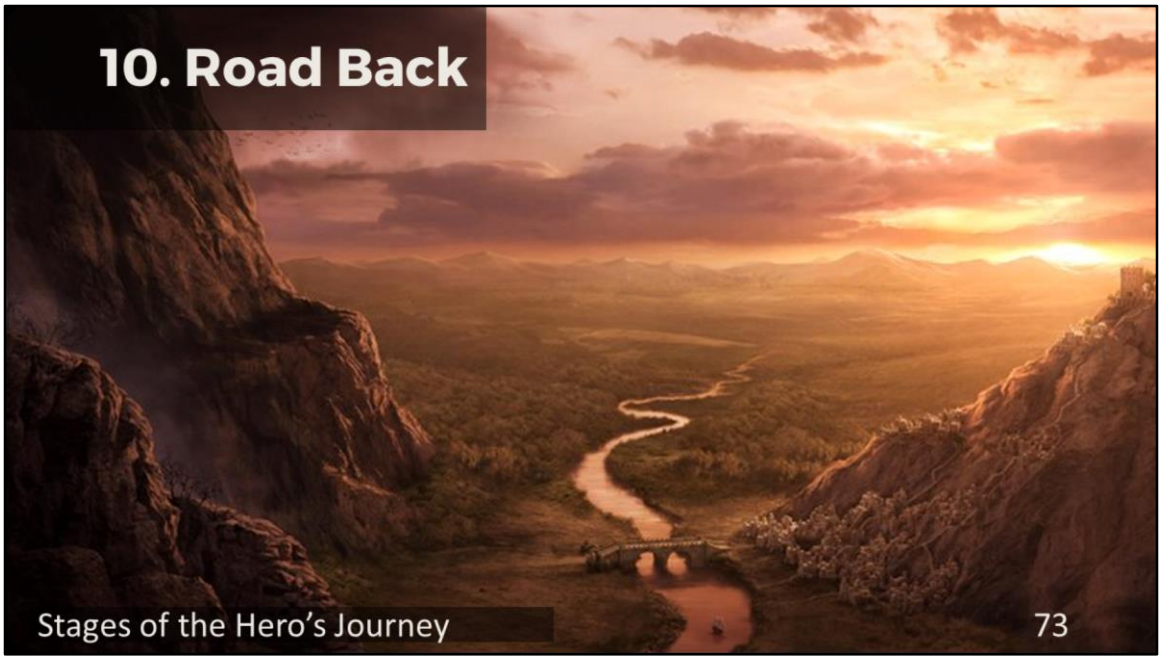


	Facades	Behaviours	SpecFlow
Maintainability	4	5	3
Readability	2	4	5
Code Complexity Index	3	4	3
Usability	4	4	1
Flexibility	1	5	4
Learning Curve	3	4	3
Least Knowledge	2	5	5
SUM	19	31	24

Here you can find the final assessments' results. Through the usage of our system we were able to find the definite winner- the Behaviour Based Tests approach. It outrun the facades in all aspects. Also, as I pointed at the beginning we didn't need the help of the last point of the system the Keep it simple principle.

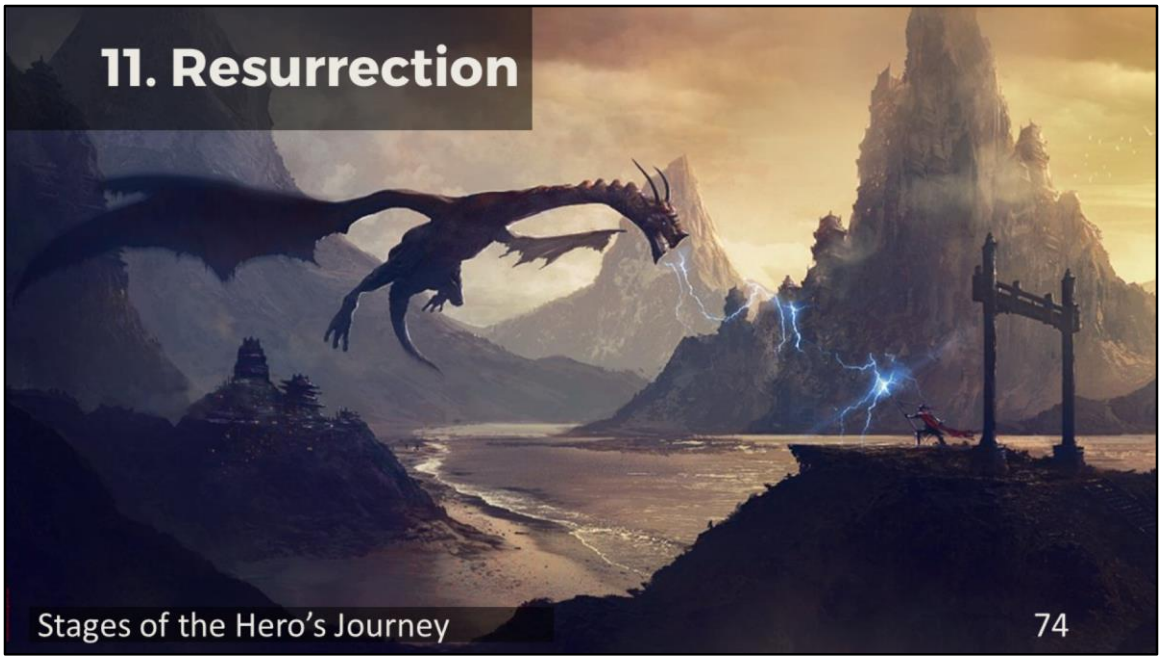


The 9th step of the hero's journey is the reward. After surviving death, the hero earns his reward or accomplishes his goal.
After lots of research, reading and mind struggling we came up with the best design between these three. Now for our end to end tests, we are using the behaviours design.

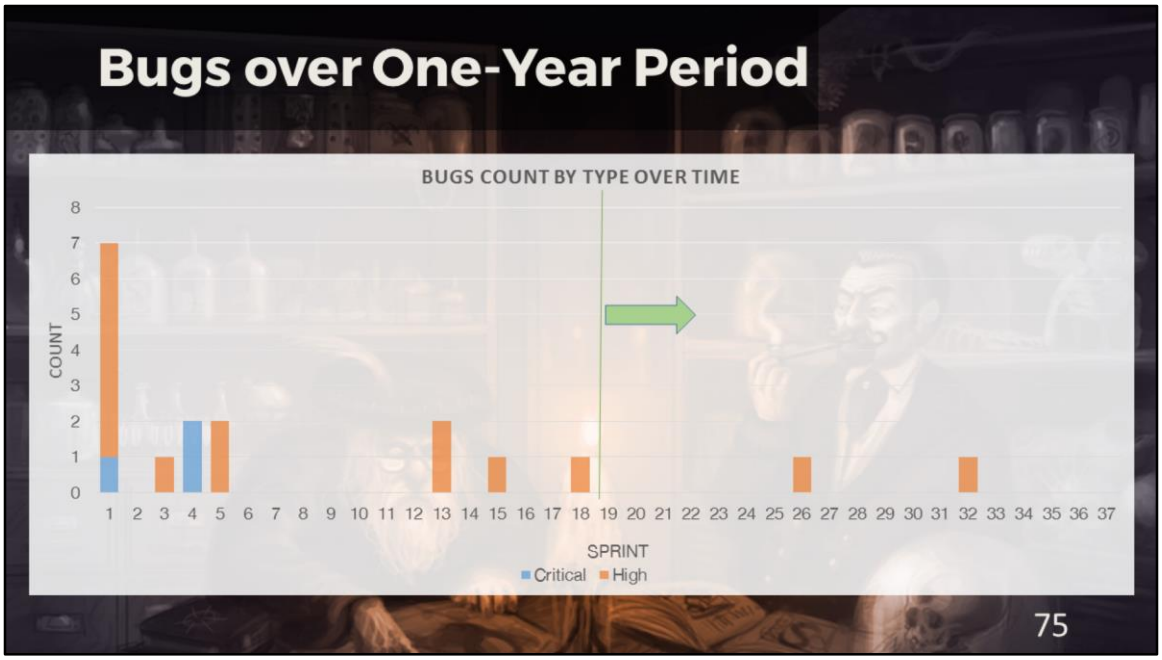


The next step is called "Road Back". The hero begins his journey back to his ordinary life.

We had this problem before that we didn't have information why the system is designed as is (usually, the people that made the decisions were not part of the team or the company anymore). What I mean here is that when we choose the preferred writing approach through our system is always good to write something like documentation or blog post in the team's portal/wiki so that if in the future the approach needs to be revised we can check fast our previous considerations why we chose it.

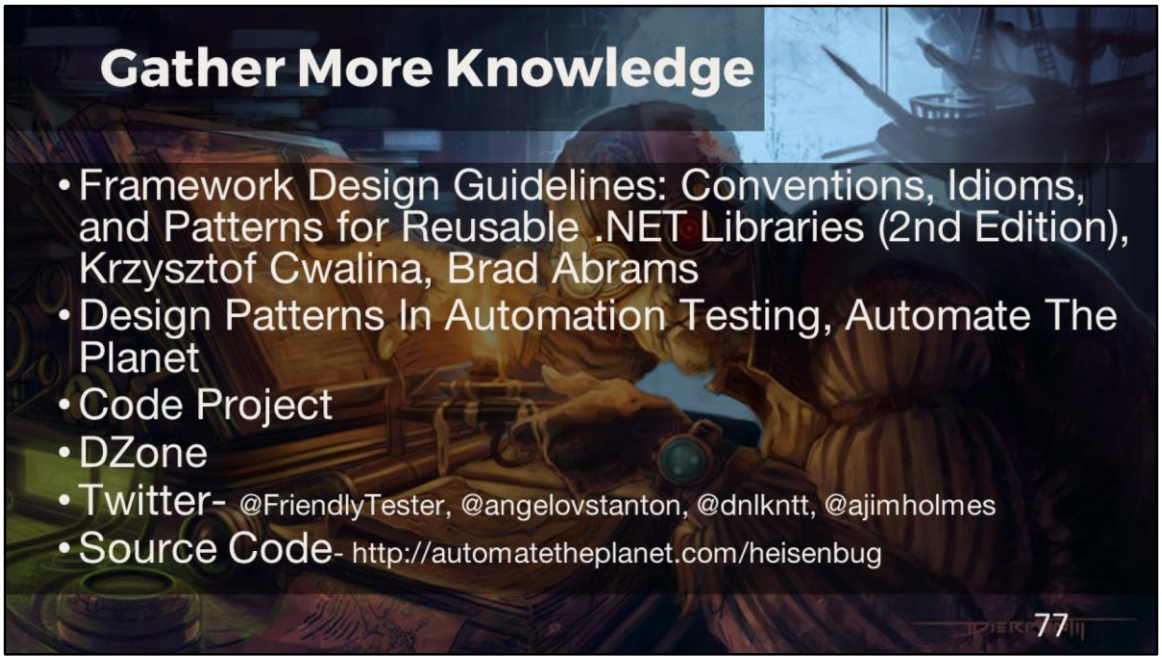


One of the most important steps from the hero's journey is the 11th one. The hero faces a final test where everything is at stake, and he must use everything he has learned. This is his final and most dangerous encounter with death.
I think this step refers to the moment where you apply the chosen design and see if it helps. I think in our case the tests became much more stable. To, believe me, I did a little research.

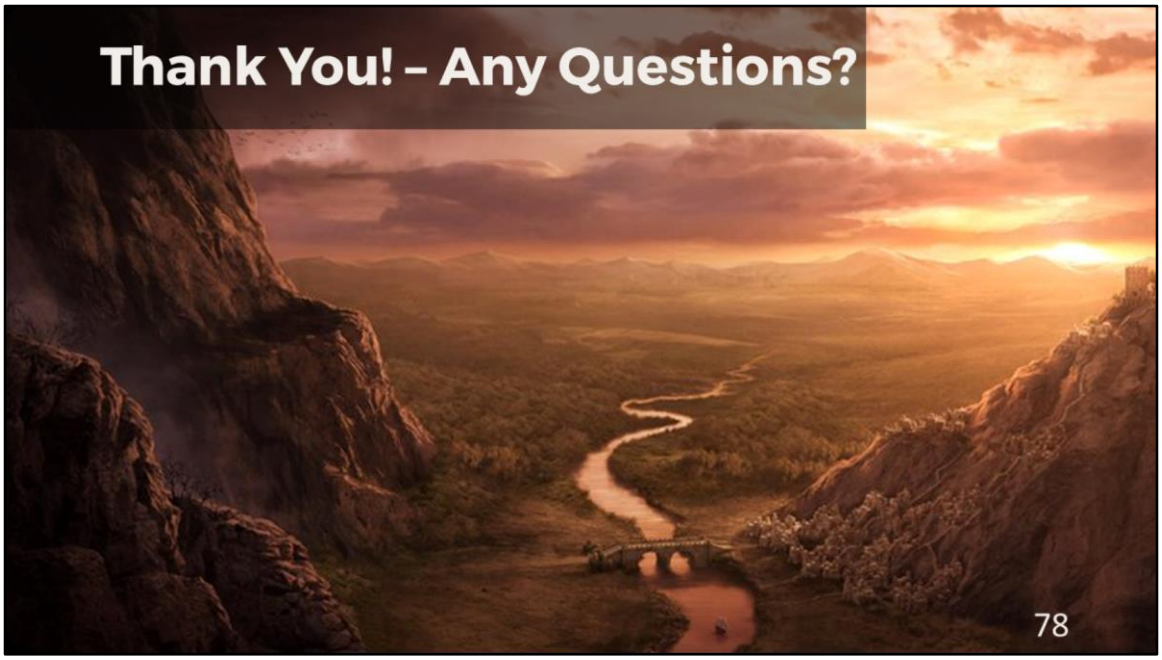


As you can observe from the chart, at the beginning we had more severe bugs compared the end of the period. Somewhere in the middle, after Sprint 19 we refactored our tests. These are the bugs found after the deployment of our applications. The bugs found on developers' machines are not logged. Most of the time our developers run some appropriate system tests along with the unit and integration tests on their machines before check-in. In the beginning, they didn't trust our tests because most of the times when the tests failed the problems were connected with regression in the tests. After we used our system and refactored the tests to use behaviours, our tests became more stable. Now our tests tend to have less regression (because the changes are scoped) and I can test and fix all bugs before check-in. I want to point out that we have CI and nightly runs, and these tests are run before that.

Here is the last step of the hero’s journey. The hero brings his knowledge or the "elixir" back to the ordinary world, and he starts to help others.
It was a long journey. I told you what problems we had at the beginning- not trustful tests. Then I told you the story behind the assessment system. We went through what each point means. Then I showed you the system in action, evaluating the facades, behaviours tests and SpecFlow tests.
It is hard to teach you to create good test framework designs, but at least you now have a powerful tool to evaluate your test framework design quality.



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.



Thank you!