**SOLID principles exercise**

**Best Practice Summary:**

* **STOP & EVALUATE**
  + **OOP objectives: Abstraction, Encapsulation, Inheritance, Polymorphism**
  + **SOLID principles:** 
    - **Single Responsibility,**
    - **Open/Closed,**
    - **Lyskov Substitution,**
    - **Interface Segregation**
    - **Dependency Inversion.**

**Best Practice Detail:**

**How one approaches developing a greenfield project of any size is up to each developer’s natural way of thinking. These approaches include:**

* **Working from the outside in:** 
  + **Coder starts with a console application and imagines what it would be like to use the application.**
  + **What would they like to see/do first if they were its user?**
  + **Then they write some code to get something basic working, like entering user details for a social network or loading media information for a collection organiser.**
* **Working from the inside out:**
  + **Coder again starts with a console application but then thinks about the elements of the problem.**
    - **They keep breaking the elements into subelements until they identify the smallest unit of operation.**
      * **In a blackjack game, this is drawing a card or placing a bet.**
      * **In a shop it’s paying for an order or adding an item to an order.**
    - **There may be multiple smallest units of operation. Which one they choose is entirely at their discretion and their choice can be completely arbitrary.**
  + **Then they write something basic to achieve the operation selected.**
* **Analysis-first: they make a list or draw a diagram of the problem’s elements and then take the inside/out or outside/in approach based on that.**
* **Interface-first: this is the approach you were shown when interface segregation was introduced.** 
  + **They create an interface named after the purpose of the application – e.g. IOnlineShop**
  + **If it’s possible to imagine oneself as the user of the application, they replay the experience of being the user of such a service in real life.** 
    - **If not, then they have to consider the core entity of the problem and anthropomorphosise it in order to visualise its journey thru the experience.**
  + **At each step of the journey thru that experience, they write a method signature representing the inputs, processing and outputs of that step. E.g. making a card payment: the method signature needs:**
    - **the card details (inputs)**
    - **the amount (inputs)**
    - **to reflect result of the transaction (outputs)**
    - **to reflect reason for any failure (outputs)**
    - **to give an indication of the processing detail – i.e. its purpose, which is of course reflected in the method’s name.**

**this would conceivably result in an interface entry like this:**

**PaymentResult CardPayment(CardDetails details, decimal amount);**

* **TDD: test-driven development – the coder takes any of the preceding approaches and writes a test to achieve each of the objectives in turn.**
  + **this approach is similar to “interface-first” in the sense that both involve using Types and Methods that don’t yet exist.**
  + **That’s the only similarity. Once the thought process has revealed a state to evaluate and the steps necessary to achieve that state, the coder can start following the TDD workflow.**

**In ALL of these cases, BEST PRACTICE is to STOP when you get the last step in any of the approaches above. Or, if you have a different approach, stop after writing the first method, class, etc and EVALUATE whether SOLID principles and/or OOP objectives have been violated.**

**RESOURCES**

**Use the SOLID checklist as documented in:**

**rlg-dotnet-fun/solid-checklist.docx**

**It’s not an exhaustive checklist by any means, and is intended as a guide to help you embed the habit of ongoing review and gradually acquire a perpetual awareness of the principles.**

**The solutions to all these parts can be found in:**

**rlg-dotnet-fun/Kata/BusinessObjectLayer/Progressive/OnlineShopV1/PartN**

**The changes to the main program are in:**

**rlg-dotnet-fun/Kata/PresentationLayer/OnlineShopDemoV1.cs**

**Each part’s functionality is in a method called PartN**

**Notes**

* **Each part’s code is in a separate folder to be able to show the development of the code over time. To avoid clashing names, I’ve put using statements in the top of the demo source file to distinguish between the folders.**

**E.g. Part1.OnlineShop refers to the type as it was defined in that part of the exercise.**

**This is a guided exercise to show how SOLID principles can be broken and how to address those sample violations.**

**Part 1**

**Objectives:**

* **Identify the basic functionality and data requirements**
* **Use the interface-first approach to reflect those in the code**

**Task:**

**Declare the minimum types to reflect an online shop that needs to:**

* + **load its product list from an inventory,**
  + **put selected products in a basket,**
  + **create an order from the basket**
  + **take payment.**

**The types we’re about to create are specific to the business needs and therefore should reside in the BusinessObjectLayer of the application architecture.**

**The Hard Way:**

* **Create an interface IOnlineShop and add methods**
  + **Online Inventory LoadInventory(string path);**
  + **OnlineBasket SelectProducts(OnlineInventory inventory)**
  + **OnlineOrder CreateOrder(OnlineBasket basket);**
  + **PaymentResult TakePayment(OnlineOrder order);**
* **Create a class OnlineShop to implement the interface**
* **Write a program to instantiate the class and call each method in turn**
* **Stop & Evaluate – these violations exist**
  + **SRP: OnlineShop is assuming responsibility for all the operations**
  + **ISP: IOnlineShop prototypes methods representing distinct functionality that is only loosely related. The implementations of these methods could differ considerably without affecting each other’s functionality.**

**The Easy Way**

* **Think about the outermost entity implied by the task, which is a shop. You could use the “online” part to derive the outermost entity is the internet, but redefining e-commerce as a whole isn’t the objective, here 🤣. So, we’re gonna need a class to represent that – OnlineShop.**
* **First, think SRP. What will this class’s single responsibility be? How do we decide that?** 
  + **What is a shop? It’s a place people go to find and purchase things they want. There’s a stock room, a till and a sample display of the items for sale, yes. But those things facilitate the disparate functions (or responsibilities) of the shop. As we saw in the Hard Way, if we go straight for coding that, we’ll hit a violation almost immediately.**

**The shop itself is a building – i.e. the shop building is a physical container for its operations.**

* **Now that we know what the shop is supposed to be doing, we can use the interface-first approach to define its operations.**
  + **Create an interface IShopContainer**
  + **Add methods to contain the main operations of the shop.**
    - **To identify what these are, we just have to think about being a customer. But if the concept wasn’t familiar to us, how would we identify what it is we’re containing?**
    - **In a word: granularity. Hierarchies are all over nature, so it follows that even the most abstract concepts will also break down into ever-decreasing units.**
      * **E.g. A Zargon needs to Jok Blokkers into Fnoods and then make Nombles and Quasks. Jokking is the process of separating Blokkers into their constituent parts: Fnoods and Fonods. Fnoods can be Karak or Rakar but only Rakar can be made into Nombles when added to an Aaarf. Quasks are what Fonods turn into if they’re placed in a Quiggle for exactly 12 seconds within 30 seconds of being extracted from their Blokker. There are limited Quiggles available.**
      * **Even this nonsensical description illustrates a hierarchy. The language tells us that this app’s primary function is to make Nombles and Quasks and a Zargon in charge of doing that.**
      * **We know that Blokkers are one of the source materials, so they’re subordinate to the Zargon as well.**
      * **Blokkers are made of Fnoods and Fonods, so that’s a direct statement that Blokkers encapsulate those two things.**
      * **Taking Blokkers apart is a process called Jokking, so knowing how to do that isn’t the responsibility of the Zargon. The Jok process is therefore a dependency of the Zargon.**
      * **Quiggles are in short supply and there’s a time crunch on once the Blokker is taken apart. So that identifies the Quiggle as a dependency of the Jok process.**
      * **Aaarfs are some mysterious entity that could conceivably be bigger than all of them, but it could just as easily be a component of the same scale as Fnoods and Fonods. Since the description says Rakar Fnoods must be added to an Aaarf, not the other way around, the process of making a Nomble seems likely to be more involved than just adding a Rakar Fnood to it. So now we have another dependency on the Nomble-making process that needs the output of another process to be input to it in the same way as Fonods need to be input into a Quiggle.**
      * **We’ve now extracted enough information from this apparent gobbldegook to visualise the application layout.**

Diagram

Description automatically generated

* + **Applying the same language analysis to our definition of a shop, “a place where people go to find and purchase things”, the methods in IShopContainer should be** 
    - **OnlineBasket Browse()**
    - **void Checkout(OnlineBasket)**
  + **And these will be encapsulated by** 
    - **void Enter(Customer)**
* **We can see now that Browse is functionally the same as SelectProducts and that Checkout encapsulates CreateOrder and TakePayment. The only method not represented is LoadInventory.**
* **In the next part we’ll see how to resolve the violations from The Hard Way at the same time as arriving at the same solution The Easy Way.**

**Part 2**

**Objectives:**

* **Resolve violations**

**Steps:**

* **Resolve SRP violation**
  + **Identify the responsibility of OnlineShop**
  + **It’s a container for the shop’s operations so none of the operations should be handled inside the class**
  + **Delete the implementations of IOnlineShop**
* **Resolve ISP violation**
  + **Since OnlineShop will not be implementing this interface due to an SRP violation, it follows that ISP must also have been violated cuz the methods are all in the same interface.**
  + **At the end of Part 1 it was established that the interface should be called IShopContainer to be indicative of its responsibility, so just delete IOnlineShop as it won’t be needed.**
* **With that first interface out of the way, we’ve effectively got the Easy and Hard paths to converge, as these next steps would be what’s needed in either case. So let’s look at the thought process for how to give ourselves a better chance of sidestepping an ISP violation.**
* **Rather than thinking about what you do in a shop, think about the main parts of a shop.**
* **Identify each one as a responsibility and give each a role name.**
  + **These are the sections of the shop as identified in Part 1**
    - **Displays – the displays are the means by which the customer selects the products they want.**

**In a shop, an assistant would be in charge of putting the products on the shelves but in the abstract world of e-commerce, we can represent that by providing a means of filtering the inventory and putting selected items into a virtual basket.**

**So, let’s create IShopAssistant with the single responsibility of Product Selection.**

**We’ve already identified a need to browse the inventory and have that process result in a basket of products, so we’ll have a method called Browse(). IShopAssistant, via this method, will need to pass the selected products back to the IShopContainer to be passed on to the next stage of the shopping process.**

**What details about the products will be needed at that next stage?**

* **The product name**
* **Its price**
* **How many of them**

**So the basket needs to encapsulate all of that for multiple products.**

**Create a class OnlineBasket with the single responsibility of Summarising Product Selection.**

**Now that we know what method will return, we can add this method to the IShopAssistant interface**

**OnlineBasket Browse();**

* + - **Stock room – who’s in charge of the stock room? A manager of some kind.** 
      * **Create IInventoryManager with the responsibility of Stock Information Provision.**
      * **The inventory manager will respond to stock queries and give out product information about all products that match the query.**
      * **Every one of these searches will need to result in the name and price of each match being returned, so that data needs to be encapsulated.**
      * **Create the Product class with the single responsibility of Encapsulating Product Stock Details.**
      * **Add a string field Name and a decimal field Price.**
      * **Now that we have the data encapsulated we can add this method to IInventoryManager**

**List<Product> Search(string searchText)**

* + - * **The ShopAssistant can now use the Search method to fulfill its responsibility.**
    - **Till – Going to the till or checkout at a shop is process with a number of steps.**
      * **we present our items**

**whose responsibility is it to receive the items? In a real shop that’s the checkout assistant.**

* + - * **we’re told how much we owe**

**who responsibility is that? It’s the checkout assistant again. They add up the prices of our selection and that makes our order ready for payment.**

**At this point, we can still leave everything there and just leave the shop. So the checkout assistant needs to be able to cancel orders too.**

**Create an interface IOrderManager with the single responsibility of Order Management**

**The presentation of items can then be achieved by passing the OnlineBasket from IShopContainer to the method that’ll record the order and calculate the total due. All the information from the basket plus the total will need to be encapsulated. And so that a receipt can be printed, an order number will need to be assigned and the date and time of the order recorded.**

**Create a class OnlineOrder with the single responsibility of Encapsulating Order Details.**

**Now we’re ready to add the method signature:**

**OnlineOrder Create(OnlineBasket)**

* + - * **we pay**

**whose responsibility is it to take payment? Well, it’s the checkout assistant, which we’ve represented with IOrderManager, but since paying is a separate responsibility, it needs to be given to a different role.**

**Create interface IPaymentManager with the single responsibility of Order Settlement.**

**Add a method to pay for the order.**

**PaymentResult Pay(OnlineOrder)**

* **Stop & Evaluate**
  + **Use solid-checklist.docx to check each of the entities created.**
  + **You should find that none of them violates any SOLID or OOP principle.**

**Part 3**

**Objectives:**

* **Implement the interfaces**
* **Identify dependencies**

**The Hard Way**

* **Create the concrete classes LocalInventoryManager, LocalPaymentManager, LocalOrderManager, ShopAssisant each implementing their respective interface.**
* **Have OnlineShop implement IShopContainer**
* **Over the next few parts we’ll be completing the implementations for each of the methods created.**
* **OnlineShop**
  + **Enter**
    - **The shopper may not necessarily be a known customer, but they have the option to become one. Altho that requirement has not been specified as yet, it is a near-certainty that this is an oversight.**
    - **Create a Customer class with the single responsibility of Encapsulating Customer Details**
  + **Browse**
    - **To browse the products we need an instance of ShopAssistant**
    - **Declare one at the class level and instantiate it in the constructor so it’s ready to call shopAssistant.Browse()**
  + **Checkout**
    - **To create the order we need an instance of LocalOrderManager.**
      * **Declare one at the class level and instantiate it in the constructor ready to call orderManager.Create(basket)**
    - **To pay for the order we need an instance of LocalPaymentManager**
      * **Declare one at the class and instantiate it in the constructor so it’s ready for when we call payManager.Pay(order)**
    - **If the payment was unsuccessful, cancel the order and inform the user.**
* **Stop & Evaluate**
  + **Use solid-checklist.docx to check the changes to OnlineShop**
  + **The constructor is in violation of the Dependency Inversion Principle (DIP) cuz the parameters and the members they’re assigned to are declared as concrete types.**
  + **To rectify the violation, change the declarations of orderManager, payManager and shopAssistant to be of their respective interface types.**
    - **Dependency Inversion is about decoupling a class from its dependencies.**

**By depending on an abstraction of a class (i.e. in the majority of cases, this means an interface), the implementation of how the particular service is provided by that dependency can change without requiring the class (the client) be modified to adapt.**

* + **The Enter method is also in violation of DIP but we’ll get to that in the next part.**

**The Easy Way**

* **Instead of creating concretions of the new interfaces straightaway, start from the container – i.e. OnlineShop and work on implementing the top level methods first.**
  + **Browse**
    - **When you reach the point of needing to make a call to the shop assistant, you are no longer tempted to use a concrete instance cuz all you have are the interfaces at that point.**
    - **Declare assistant at the class level of type IShopAssistant**
  + **Checkout**
    - **Similarly declare orderManager and payManager as IOrderManager and IPaymentManager respectively.**
* **Use Control-Dot (Right click “Quick Actions”) to “Generate constructor…” and have all 3 dependencies satisfied by passing in their concretions thru constructor arguments.**
* **Stop & Evaluate**
  + **Use solid-checklist.docx to check OnlineShop**
  + **Note the Enter method’s violation of DIP is unresolved.**

**Part 4**

**Objectives:**

* **Get the process to run end-to-end**

**The Hard Way**

* **Take each method in the container interface in turn.**
  + **Enter(Customer)**
    - **This needs a customer object passed into it, so create a Customer class with an email and password in the main program, instantiate it and pass it to Enter on the shop instance.**
    - **Call the Browse method.**
    - **Pass the resultant OnlineBasket to Checkout()**
  + **Browse()**
    - **Call shopAssistant.Browse()**
    - **Return the OnlineBasket**
  + **Checkout()**
    - **Pass the incoming OnlineBasket to orderManager.Create()**
    - **Pass the resultant OnlineOrder to payManager.Pay()**
* **Stop & Evaluate** 
  + **The unresolved DIP violation from Part 3 has now compounded the problem and resulted in breakages of the OOP principles underlying SOLID! Before we look at SOLID violations, we have some general OOP violations to think about.**
    - **Encapsulation**
      * **the Customer object is passed into the Enter method. if the shop is the outermost container of operations, then creation of Customer objects should be encapsulated within it.**
    - **DRY – we have 2 interfaces defining a Browse method.** 
      * **This is a side effect of using the interface-first approach that results in another violation of Encapsulation in terms of only exposing what is needed.**
      * **Why is it a problem? External code can instantiate a shop, fabricate its own baskets and send them to the checkout process. So this violation has left the class open to abuse.**
  + **If we now think about SOLID violations we’ll find this failure:**
    - **DIP: “do any of the public methods accept a concrete type that is used to satisfy an internal dependency?”.** 
      * **The Enter method is not inverting its dependency on the Customer object.**
      * **It isn’t making that available to the rest of the class, but the Enter method is the “Main” of “OnlineShop” – it’s the containing method. So it’s very likely to at least pass it on to other methods if not copy the reference to a class member at some point.**

**The Easy Way**

* **The first thing to recognise is that we’ve carried forward SOLID violations into this part.** 
  + **How did we end up having to do that?**

**The problem wasn’t noticed early enough.**

* + **How did it happen?**

**We didn’t run the whole checklist against every entity.**

**When we assessed the interfaces, we only did the ISP checks based on the assumption that the other principles only apply to classes.**

**We also didn’t check the underlying principles of encapsulation and code re-use (DRY) across the classes.**

* **If we hadn’t skipped those extra checks, we’d have caught one of these problems in Part 1, when IShopContainer was created.** 
  + **This demonstrates that even when you’re trying to follow SOLID principles in the strictest possible terms, you’ll still miss things if you don’t run all the checks every time.**
* **Part 1-3 Replay with corrections:**
  + **Start with IShopContainer**
  + **Identify methods Enter, Browse and Checkout**
  + **Remember the shops’ Responsibility is to contain the customer journey so all clients of the class should have to start that process from the beginning.**
    - **Add void Enter(Customer) to the interface, but nothing else.**
  + **Run the checklist against ALL of SOLID.**
    - **Enter(Customer) is flagged for DIP cuz it’s a concrete type passed as a method parameter.**
    - **This method represents the entire path of execution for the class and that makes it a class level dependency cuz that top level method can pass it to any other method in the class.**
  + **Before we think about resolving this violation, let’s make sure we thought the Enter method through properly. Look at the section of Part 1 where it was first introduced. The decision is seemingly made without any analysis. There’s no explanation of why this method was needed, whereas the other two are explained in detail.**
    - **So why did we choose the name Enter? To simulate the customer entering the shop.**
    - **Why is the Customer object a dependency of that? In case it’s a known customer.**
    - **How would we really know if it’s a known customer if the client can instantiate any Customer they like and pass it in? We wouldn’t.**
    - **A Customer object can only be created if the user has gone thru some kind of authentication process.**
    - **Whose responsibility should that be? A security manager.**
      * **Create a new interface ISecurityManager with the single responsibility of User Authentication**
      * **Add a method Customer Login()**
      * **Remove Customer from IShopContainer.Enter()**
    - **Whose responsibility is it to initiate user authentication? As an operation of the online shop, it is IShopContainer’s job to provide the means of authenticating (and registering) users.**
    - **What are the consequences of making everybody login? The shop will lose business if people can’t be bothered to register.**
    - **So, we need to find out if (a) the user wants to login, (b) wants to register or (c) wants to be a “guest”.**
    - **Whose responsibility is that? It’s an operation of the shop, so…**
    - **What are the consequences of making that the first step of the journey? The shop will lose business cuz people just wanna start browsing.**
    - **So, we don’t need a customer object from the start, if at all. Remove the parameter from Enter(Customer)**
    - **So if “Entering” the shop is potentially never actually going to happen in terms of authentication, then the shop’s single entry point for the whole process should not be called Enter. It’s counterintuitive.**
    - **Upon reviewing the real life journey of a customer, it’s not entering the shop that’s important. It’s getting served.**

**If we’d used a different shop as an analogy, it might have been more obvious. But – it’s same for any shop.**

**You can join the queue at a kebab van and your experience there means nothing to either you or the business until it’s your turn – i.e. until an order is placed.**

**Thinking about the customer journey from their perspective is very important so that the empathic effects will yield considerate features like not demanding they login straightaway. But, the application is being written to serve the business via serving the customer and if we’d remembered that, the first point of contact with the customer would’ve emerged as the opening step of the journey.**

* + - **In IShopContainer replace the Enter method with:**

**void ServeCustomer();**

* + **Create the interfaces IShopAssistant, IOrderManager and IPaymentManager exactly as before.**
  + **In OnlineShop**
    - **Implement IShopContainer**
    - **Declare the following private members:**
      * **IShopAssistant shopAssistant**
      * **IOrderManager orderManager**
      * **IPaymentManager payManager**
      * **ISecurityManager secManager**
    - **Use Control-Dot (or Right Click->Quick Actions) “Generate constructor” and let all four be injected as constructor arguments.**
    - **In ServeCustomer()**
      * **Assign a variable basket to the result of calling shopAssistant.Browse();**
      * **It’s possible the user doesn’t want to complete their purchase on this visit so we need to establish how they terminated their browsing process.**
        + **Checking for the count of items in the basket being 0 isn’t enough. They might want to save their basket for later.**
        + **So the Browse method needs to change.**
        + **STOP! It needs to be thought through.**
        + **Add a comment “// TODO: refactor IShopAssistant.Browse to also provide user decision.” and move on.**
      * **Once we know their decision we can determine if we need to ask them to decide to login, register or checkout as a guest.**
        + **STOP! It needs to be thought through.**
        + **Add a comment “// TODO: prompt for login etc depending on browse status”**
      * **Create the order:**

**var order = orderManager.Create(basket);**

* + - * **Assign a variable paymentStatus to the result of calling payManager.Pay(order);**

**var paymentStatus = payManager.Pay(order);**

* **Stop & Evaluate**
  + **By this point you should have:**
    - **OnlineShop implementing IShopContainer**
      * **ServeCustomer**
        + **Calling Browse, Create and Pay**
        + **2 x TODO items**
      * **4 private interface abstractions of concrete instances to be passed in as constructor arguments**
    - **OnlineBasket – empty class**
    - **OnlineOrder**
      * **OrderId, Total, Paid (bool) and Created date/time**
    - **Product**
      * **Name and Price**
    - **5 interfaces**
      * **IInventoryManager – List<Product> Search(string)**
      * **ISecurityManager – Customer Login()**
      * **IOrderManager – OnlineOrder Create(OnlineBasket)**
      * **IPaymentManager – PaymentResult Pay(OnlineOrder)**
      * **IShopAssistant – OnlineBasket Browse()**
  + **Run the whole checklist against all of them – should be clean.**
* **We still can’t achieve the objective of this part cuz the second we call one of the dependencies we’ll get a NullReferenceException, but we’ve certainly got a strong idea of the direction we’re heading in.**

**Part 5**

**Objectives**

* **Implement IShopAssistant**

**The Hard Way**

* **By now you should realise there’s no point typing in any of this 😊**
* **Create a class ShopAssistant implementing IShopAssistant**
* **In Browse()**
  + **Firstly we know that this method needs to return an OnlineBasket so instantiate one.**
  + **Next, we need to identify the steps required to simulate a shopping experience.**
    - **We look at one or more displays, scanning them for what we want and if we like something, we select it.**
  + **So we need to let the user enter repeated queries until they’re done browsing/selecting products. In order to do this, we need:**
    - **A loop**
    - **A means for the user to terminate the loop**
    - **To ask the user what they’re looking for**
      * **Use the Console to get the search string**
    - **Search the inventory**
      * **Searching the inventory is part of LocalInventoryManager so we need an instance of this to be able to call the Search method.**
      * **Declare one at the class level and instantiate it in the constructor so that it’s ready for us to call invManager.Search()**
    - **Consider the “unhappy path”: what do we do if:**
      * **There are no matches**
        + **Report no matches and ask what to do:**

**Search again**

**Checkout (if there are items in the basket)**

**save the basket (if not empty)**

**Leave the shop (quit)**

* + - * **the user enters nothing**
        + **ask what to do:**

**search again**

**checkout (if basket not empty)**

**save the basket (if not empty)**

**leave the shop (quit)**

* + - **Display the results**
      * **If not available add an indication that it would need a special order?**
    - **Give the user a choice of:**
      * **Selecting an item**
        + **If not available, ask if they want to order it.**
        + **Ask the user how many they want**

**If not available, ask if they want to order it.**

* + - * + **Add the product details and the quantity to the basket.**
        + **Go back to the results view to see if they want to select another product.**
      * **Starting a new search**
        + **Discard the results and restart the loop**
      * **Checking out**
        + **Return the basket**
      * **Save the basket** 
        + **Return the basket**
      * **Abandon the basket**
        + **Discard the basket and return nothing**
* **Stop & Evaluate**
  + **SRP: why is this class talking directly to the console? This class’s single responsibility is…hang on! We got so excited about the implementation of the interface that we were distracted from the fact that its responsibility was not named!**

**We shouldn’t beat ourselves up too much, tho, cuz the interface’s responsibility has been clearly defined and so that is inherent in the implementation.**

**So ShopAssistant’s job is Product Selection. Interacting with the user is a whole different kettle of ball games 😉**

* + **DRY: how many times do we ask the user the same question?**
  + **DIP: why are we instantiating a LocalInventoryManager in the constructor? That’s a concrete class.**
  + **TODO: we haven’t considered the outstanding TODO items we added as comments in OnlineShop, which seriously affect how we write this method.**

**The Easy Way**

* **As part of Part 4’s (and every part’s) S&E process we need to add “Carry any TODO items forward”. Those issues may not have a bearing on the next task but it’s important to consider them before continuing.**
* **This should also include that once all violations have been resolved (and/or tests passed in TDD), that is the time to do a git commit 😊**
* **So let’s look at the TODO items in turn:**
  + **refactor IShopAssistant.Browse to also provide user decision**
    - **As we’ve seen above, the user decision is a multiple choice question with differing options, depending on where in the browsing process they are.**
    - **With so many questions being asked (and in rich graphical interfaces like web apps etc, even more so) and in particular because the app has financial ramifications, the importance of keeping track of decisions is paramount.**
    - **So, not only should the Browse method be returning the \*last\* decision made before it returned, it should be recording \*every\* decision the user makes and every decision the application makes as a result. This information would become critical in the event of a dispute.**
    - **Whose responsibility is it to store that information? It’s basically an auditor, isn’t it?**
    - **Create an interface IAuditor with the single responsibility of Recording Business Events.**
    - **But it needs to be thought thru so don’t get distracted by it. Just add a comment to the new file “// TODO: define and implement this interface”**
    - **At each point the user is asked a question, they have the option to “leave the shop”, so all of the possible answers need to be encapsulated in one place so that the method can return a consistent type.**
    - **Create a new enum CustomerDecision with the single responsibility of Encapsulating Customer Decisions.**
    - **Don’t get distracted. Add a comment to the new file, “// TODO: define enumerations”**
    - **It is counter intuitive to have the method status returned in a ref/out parameter. This is greenfield code, so there are no OCP considerations: i.e. there are no clients that would need to refactor other than our own code, which is the reason we’re thinking about this in the first place.**
    - **But if we change the return type to CustomerDecision, what do we do with OnlineBasket? We’ll need to make that an out parameter so that we can instantiate it for the caller.**
    - **Modify IShopAssistant so that Browse() looks like this:**

**CustomerDecision Browse(out OnlineBasket basket);**

* + **“// TODO: prompt for login etc depending on browse status”**
    - **This item is not directly related to the functionality of the Browse method, and that would normally be enough to carry this forward to the next part.**
    - **But prompting the user is something the Browse method also needs to do, and now that we have 2 classes needing to interact with the console, a dependency is emerging on something to take responsibility for User Interaction.**
    - **Create an interface IInteractor with the single responsibility of User Interaction.**
    - **Don’t get distracted. Add a comment, “// TODO: define and implement IInteractor”**
  + **Now we can create ShopAssistant with the inherent single responsibility of Product Selection knowing that the Browse method is set up exactly as it needs to be.**
  + **In Browse…**
    - **The method still needs to be written as detailed above but now that we know the interaction is gonna be handed off to IInteractor, let’s skip those bits for the time being.**
    - **We know the return type is CustomerDecision but we’re gonna return as soon as we \*can\* so no need to declare one (yet).**
    - **The method is essentially an endless loop, terminated by one of a range of user decisions. So, we know the while(condition) will be based on something of the CustomerDecision type. And now we need that declaration so it’ll be in the right scope when the while() statement evaluates its condition.**
    - **Let’s use the technique of writing comments for each thing to be done in the loop.**

**var decision = default(CustomerDecision);**

**while(/\* TODO: setup the enum \*/)**

**{**

**// get user search string**

**// if blank, do they wanna quit/checkout/search again?**

**// if quit, do they wanna save their basket?**

**// if checkout return basket**

**// if search again continue**

**// search inventory**

**// if no results do they wanna quit/checkout/search again**

**// (same conditional code as before)**

**// display results**

**// do they wanna add to basket/search again/checkout/quit**

**// (same conditional code + add to basket branch)**

**}**

* + - **As previously noted, the loop is heavily dependent on user interaction and we can see the same base interaction is prompted several times per iteration.**
    - **This suggest that functionality should be encapsulated in a method of its own before we even think about how to build an implementation of IInteractor.**
      * **This method could be made to be very flexible and deal with any combination of decisions. The most straightforward way of doing that is to pass in a group of possible decisions and a prompt.**
      * **Add a local method GetCustomerDecision() returning CustomerDecision with these params**
        + **String prompt**
        + **Params CustomerDecision[] options**
    - **We know that adding to basket involves selecting a product and then entering a quantity, so that is best encapsulated too.**
      * **Add a local method**

**void AddToBasket(Product product, OnlineBasket basket)**

* + - **We also know that quitting involves asking a further question as well so we’ll encapsulate that too.**
      * **Add a local method. This one will need to indicate another CustomerDecision**

**CustomerDecision QuitWithoutSaving()**

* + - **It appears there is a frequent need to associate the user’s choice with a further small method, so our IInteractor implementation should provide a means of specifying those methods when the particular interaction method is called.**
    - **QuitWithoutSaving could be changed to return a bool, which would oblige the caller to set the CustomerDecision but then AddToBasket could also return a bool and the “extra method” then has a standard return type, that’ll make it easier for the interactor to implement.**
    - **AddtoBasket can then be renamed TryAddToBasket which indicates that if the product already exists in the basket, it’ll just increase its quantity by whatever is specified by the user this round.**
    - **It follows then that GetCustomerDecision won’t actually be called directly from the main Browse loop but from another method that manages the initial interaction (to quit etc) and any subsequently required interaction, which lightens the load on the main loop.**
      * **Add a local method**

**CustomerDecision SelectProduct(List<Product> results)**

* + - **By passing in the whole result set (whether it’s empty or not), we are further streamlining the main loop, as SelectProduct can now not only display any results but it can decide whether to ask the 3 option question or the 4 option version, which includes TryAddToBasket.**
    - **That just leaves searching the inventory to take care of, which is a simple matter of declaring this class level variable**

**private IInventoryManager invManager;**

* + - **And “Generate constructor” but while we’re here we should also add the interactor and auditor dependencies too, so…**

**private IInventoryManager invManager;**

**private IInteractor interactor;**

**private IAuditor auditor;**

* + - **Now, we can generate the constructor and gain the full benefit.**
    - **We now have enough information to add the enum values to CustomerDecision**
      * **None (the default as above)**
      * **Quit**
      * **SaveAndQuit**
      * **Checkout**
      * **SearchAgain**
      * **AddToBasket**
    - **And while we’re here, we might as well add the decisions we identified for the shop as well**
      * **CheckoutAsGuest**
      * **Login**
      * **Register**
    - **And that enables us to write the while(condition)**
      * **It’s gonna look really messy if we express this as a direct boolean express – i.e. comparing the decision with each of the 3 possible decisions that can terminate the loop.**
      * **We \_could\_ advantage of C#’s implicit strong typing and create an array on the fly, but it’s not gonna improve matters that much and it’ll make the code look more arcane.**

**while(!new []{CustomerDecision.Quit, CustomerDecision.SaveAndQuit, CustomerDecision.Checkout}.Contains(decision))**

**{**

**}**

* + - * **So let’s declare the array before the loop.**
    - **The loop will end up looking like this:**

**var decision = default(CustomerDecision);**

**var exitOptions = new []**

**{**

**CustomerDecision.Quit,**

**CustomerDecision.SaveAndQuit,**

**CustomerDecision.Checkout**

**};**

**while(!exitOptions.Contains(decision))**

**{**

**var searchText = string.Empty;**

**decision = this.GetSearchString(out searchText);**

**if (decision != CustomerDecision.None) continue;**

**var results = invManager.Search(searchText);**

**decision = this.GetBrowseDecision(results);**

**}**

* + - **Add TODO items**
      * **Write GetSearchString()**
      * **Write GetBrowseDecision()**
      * **Write QuitWithoutSaving()**
      * **Write TryAddToBasket()**
      * **Add auditing.**
* **Stop & Evaluate**
  + **No new concretions, so SRP still okay.**
  + **The new interfaces are just stubs so ISP is fine.**
  + **The ShopAssistant had its dependencies injected as Ixxx so DIP is good too.**
  + **OCP, as mentioned earlier, isn’t really an issue as there’s only one client and the requirement to change the interface came from that client.**
  + **LSP isn’t a factor – even tho we haven’t covered it yet – as there’s no polymorphism to consider yet.**
  + **Carry the TODO items forward.**
  + **Do a git commit**

**Part 6**

**Objectives**

* **Review TODO list**

**The Easy Way**

* **Write GetSearchString()**
* **Write GetBrowseDecision()**
* **Write QuitWithoutSaving()**
* **Write TryAddToBasket()**
* **Add auditing.**

**The Hard Way**

* **Create a class LocalInventoryManager implementing IInventoryManager**
* **We know the Search method has to return a List<Product> so declare one in that method.**
* **The Search method return a subset of the inventory so the inventory itself must also be a List<Product>. Declare a private List<Product> inventory and instantiate it in the constructor.**
* **This is where we actually need to load the inventory but it should happen as soon as the application opens. Create a private method LoadInventory() and call it from the constructor.**
* **In LoadInventory()** 
  + **We know the inventory is in a csv file so use File.ReadAllLines to retrieve the records.**
  + **Loop thru the records, splitting them into name and price and instantiating Products. Add each new Product to the class level inventory.**