**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 assistant.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**
  + **You should find that the constructor is in violation of the Dependency Inversion Principle (DIP)**
  + **To rectify the violation, change the declarations of invManager, payManager and assistant 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. 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 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 the interface.**
    - **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**
  + **You should find that none of them violates any SOLID or OOP principle.**