**Dependency Inversion Recipe Exercise**

**(that one’s for you, Helen 🤣🤣)**

**Intro**

**Exercise objectives are to practice:**

* **abstraction of a process**
* **Practice existing SOLID principles SRP & ISP**
* **Introduce new SOLID principles OCP and DIP**
* **use of anonymous methods/lambda expressions**
* **use of generic type parameters at class, interface and method level**

**This exercise also introduces:**

* **local file i/o**
* **JSON (javascript object notation)**
* **generic type constraints**
  + **"where " : a means of filtering the number of types that can be passed as the generic type argument(s)**
  + **"where new()" : a means of enabling a generic type to be instantiated**
* **cursory introduction to specifying "attribute" syntax**
  + **applying the [Flags] attribute to an enum**

**Overall task:**

**Simulate the process of cooking a meal from getting the ingredients out to dishing up.**

**The complete code for each part can be found in Kata/CustomTypes/Demo/Recipe/Part(n)**

**The main code is in Kata/Demos/Demo/RecipeDemo.cs**

**NOTE: Each part is separated into its own method and to allow the use of the same class names each time, the class must be prefixed with Part(n). Part(n) has been declared as an alias for Kata.CustomTypes.Recipe.Part(n) with a using statement.**

**Part 1**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part1**

**objective: encapsulate the smallest units of data**

**- create a class Ingredient, with 2 readonly string properties Name and Amount**

**- generate a constructor to initialise the properties**

**- override ToString to return "Ingredient" with "amount" of "name"**

**Part 2**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part2**

**objectives:**

**- practice using an anonymous method in isolation**

**in your main program (or the Run method of your demo class if you're using my template)**

**- declare a list of ingredient**

**- declare an anonymous method, "measure" that returns an Ingredient and accepts 2 strings as arguments**

**- make the body of the method instantiate and return an ingredient using the 2 strings as the name and amount of the ingredient.**

**- call the method with the following sample data:**

**egg 4**

**salt a pinch**

**beef mince 1/2 lb**

**spaghetti 250g**

**- add the ingredients created to the list**

**- loop thru the ingredients, writing them to the console**

**Part 3**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part3**

**objective:**

**- encapsulate the functionality to prepare an ingredient**

**- practice coding interfaces**

**steps:**

**- create an interface IIngredient with a single method Prepare accepting no arguments**

**- implement the interface in the Ingredient class**

**- make the Prepare method virtual**

**- have the Prepare method output a message saying it is preparing the ingredient, specifying its name**

**- update your program to loop thru the list, calling the Prepare method on each**

**Part 4**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part4**

**objectives:**

**- practice overriding a virtual method**

**- practice providing a parameterless constructor while satisfying the requirements of the base constructor**

**steps:**

**- declare a class Tomato inheriting from Ingredient**

**- generate the appropriate constructor**

**- remove the name argument and pass the literal "Tomato" to the base class**

**- in your program, instantiate the class and add it to the list**

**- loop thru the list again.**

**Part 5**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part5**

**objective:**

**- provide a means of reflecting the milestones in the preparation of a meal**

**steps:**

**- declare class Stage to represent the combinations of ingredients at various points**

**- add a bool Ready to it**

**- add a string Name**

**- override ToString() to show the state of the Stage**

**- instantiate a stage and output it**

**Part 6**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part6**

**conceptual objectives:**

**- practice class-level generics**

**- practice interface-level generics**

**- practice anonymous methods**

**functional objective:**

**- provide a means of abstracting and encapsulating the functionality of following a recipe**

**steps:**

**- declare an interface IRecipeFollower<T>**

**- add a void method Follow with no arguments**

**- declare a class Recipe<TDish>**

**- declare a private member Dish of the generic type**

**- generate a constructor**

**- have Recipe implement IRecipeFollower and pass TDish on to it as T**

**- override Tostring to provide some text indicating the recipe being followed.**

**- instantiate a Recipe of string and code a call to the Follow method**

**Part 7**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part7**

**conceptual objectives:**

**- introduce generic type constraints**

**functional objectives:**

**- have the recipe class instantiate the dish**

**steps:**

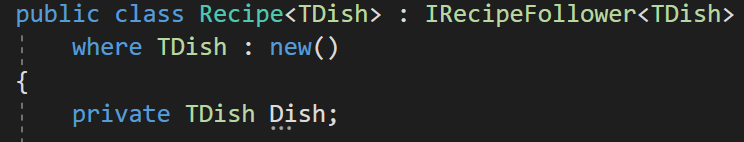
* **Remove the constructor parameter from Recipe**
* **Change the assignment of Dish to new TDish();**

**| the squiggly says you can’t do that cuz “it doesn’t have the new() constraint”.**

**| previously, I’ve used that as an example of “bad generics” but that was only**

**| cuz we hadn’t learnt about generic type constraints.**

* **Change the Recipe class definition so it looks like this:**

****

**| You can now use the Recipe class to instantiate any type. This is not good.**

**| It leaves the class open to abuse. We can address that problem by adding another**

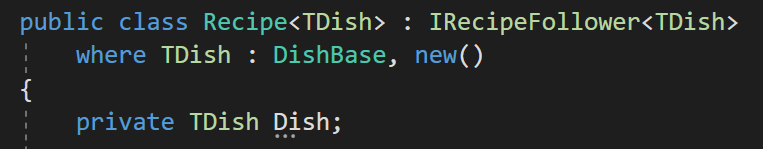
**| generic type constraint**

* **create a class DishBase that inherits from List<Ingredient>**
* **add a public string property DishName**
* **create a class SpagBol that inherits from DishBase**
* **create a class Salad that doesn't inherit anything**
* **add a public string property SaladName to Salad**
* **Override tostring() in both DishBase and Salad**
* **override ToString in Recipe and output the DishName property of Dish**
* **test this with both SpagBol and Salad**

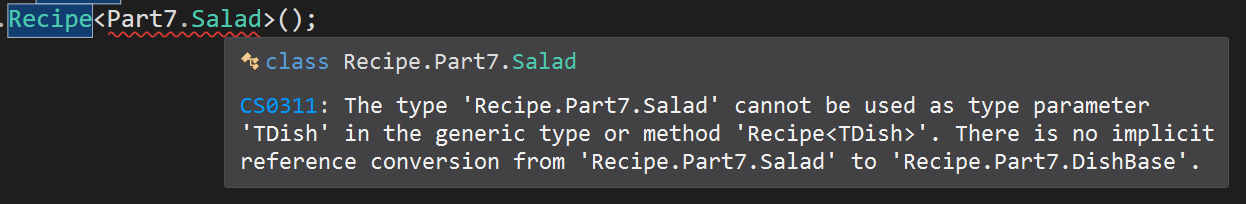
**| Salad doesn’t output anything cuz we couldn’t set hte SaladName**

**| And possible rogue types can be created.**

* **change the Recipe class definition again:**

****

**| when you now go back to the program, you’ll see you have a squiggly on Salad.**

****

* **To complete this part, have Salad inherit from DishBase.**

**part 8**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part8**

**conceptual objectives:**

**- using generic types with anonymous methods**

**- practice generic type constraints**

**- practice anonymous methods**

**functional objectives:**

**- provide a means of instantiating the ingredients for the dish**

**steps:**

* **In Recipe**
  + **Update the Follow method to**
    - **Announcing following recipes for “Dish”**
    - **Loop thru the ingredients, calling the Prepare method of each**
* **create an interface IPantry**
* **add a method Measure returning Ingredient and accepting name and amount (like the first anonymous method “measure” in Part 2)**
* **Add an overload for Measure that** 
  + **returns a generic type TIngredient (9A)**
  + **Accepts an anonymous method “instantiate” that:**
    - **Returns TIngredient (9B)**
* **declare a class Pantry that implements IPantry (9C)**
* **add a pantry to DishBase that inheritors can access but not external code (9D)**
* **in the Spagbol constructor call pantry.Measure with string arguments and add the result to the list**
* **then call the generically typed pantry.Measure for a Tomato and add that to the list**
* **To protect this method from abuse, you could also add a type constraint for Ingredient, but because the result is being added to a List of Ingredient, that type restriction takes care of that concern.**

**part 9**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part9**

**conceptual objectives**

**- using generic types with anonymous methods**

**- practice inline anonymous methods**

**functional objectives**

**- solve the problem of slicing tomatoes being the wrong preparation method for spaghetti bolognese**

**steps:**

* **In Ingredient**
  + **Add a protected Action AltPrepMethod**
  + **Add a constructor overload receiving altPrepMethod and assigning it to AltPrepMethod**
  + **Call the original constructor with the other two arguments**
  + **Modify the Prepare method to call AltPrepMethod if not null after the default message has been output**

**| To have added the Action to the Prepare method as an argument would require**

**| extensive modification of the Follow method to accommodate every ingredient with an**

**| alternative preparation method.**

**|**

**| That would violate the Single Responsibility Principle. It is not Recipe’s job to**

**| set up the ingredients**

**|**

**| Also, if the code was already in use by other codebases it would also violate the**

**| Open Closed Principle cuz the class would be “Closed to Modification”.**

**|**

**| the Recipe class would still be “Open to Extension”, however, so the steps below**

**| might not be the first idea one came up with. There are a few other options but all**

**| result in violation of other principles.**

**|**

**| rather than argue the problem with each, the way to reach the right answer straight**

**| away is to ask “whose job is this?”**

**|**

**| the job is “preparing ingredients”.**

**|**

**| Which class has the responsibility of preparing ingredients? Ingredient. Or more**

**| correctly, any implementor of IIngredient.**

**|**

**| So, this is why AltPrepMethod was added to Ingredient.**

**|**

* **In Tomato**
  + **Add a constructor overload receiving string “amount” and Action “altPrepMethod”**
  + **Assign AltPrepMethod**
  + **Pass “amount” to the original constructor**
  + **Override the Prepare method** 
    - **call the version in the parent class**
    - **Write out a message “Slicing the tomatoes”**
* **In SpagBol**
  + **Add the altPrepMethod argument to the Tomato instantiation**
    - **Have the method body write a message saying “soak the tomatoes in hot water then drain, skin and mash them”**
    - **That looks a bit cluttered so move the method body to a private void method “PrepTomatoes” and put PrepTomatoes in the new Tomato() argument list.**
* **Have your code instantiate a Recipe<SpagBol> and run the Follow method.**

**| Hmm...we seem to be mashing the Tomatoes and then slicing them.**

**| Let’s come back to that.**

**Part 10**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part10**

**Conceptual objective:**

* **Introduce deserialization from a JSON file**
* **Practice Linq**

**Functional objective**

* **Establish a database of dishes for the Recipe program to make.**
* **Maintain a default method for instantiating ingredients**
* **Maintain an alternate method for instantiating IIngredients**
* **Put a gatekeeper in front of these methods to ensure the specified ingredient exists on the data source.**

**Steps:**

* **In Pantry add a private Dictionary<string, string> “stock”. Do not instantiate it.**
* **In IPantry, Add a new overload for the Measure method that returns an Ingredient and takes a string parameter “strIngredient”**
* **Implement the method to search the dictionary for the ingredient name. If it’s there, call the 2 string Measure overload with the name and amount from the dictionary.** 
  + **Use Linq together with string methods to make the search ignore case and spacing**
  + **If the entry isn’t found, create the ingredient with a default amount.**
* **In DishBase, change your constructor to instantiate the pantry using JsonConvert.DeserializeObject on ingredients.json**
* **Remember to change the properties of the json file so that it is “Copy always” to build directory.**
* **Update your calls to Measure to use mixed case, abbreviation and space padded versions of the ingredient names**

**| when you get to tomato, you have a problem. The lookup on the dictionary will only**

**| ever return you an Ingredient, but for this and a number of others, a child class**

**| will be necessary. You can’t cast down to Tomato so how do you convert the object?**

**|**

**| 1. You could add a 4th overload, to search the dictionary and then**

**| call an anonymous method just like “instantiate”**

**|**

**| 2. You could extend the instantiate method in SpagBol’s constructor. It’s not**

**| possible to cast down but you could use the database to instantiate tomato as**

**| a regular ingredient and then pass its data to instantiate Tomato.**

**|**

**| 3. You could recognise that with the introduction of an external data source,**

**| the dishes possible are restricted by the ingredients available. (we’re gonna**

**| ignore the concept of stock quantities for simplicity).**

**|**

**| having arrived at 3, it is now a design flaw for the IPantry interface to expose a**

**| method that directly instantiates an Ingredient without checking its data source.**

* **Remove Measure(string name, string amount) from IPantry**

**| The default creation of an Ingredient is still gonna be needed, however, so leave**

**| that original version of the method in Pantry but, you don’t want anyone calling**

**| it unless they know what they’re doing.**

**|**

**| Single Responsibility tells us that the ONLY class that \*should\* know what it’s**

**| doing with regards the responsibility of transforming data-source elements into**

**| Ingredients is implementors of IPantry - i.e. the Pantry class.**

* **Tighten the method access to private**

**| The remaining overloads can now be merged to encapsulate the required**

**| logic to achieve the functional objectives.**

**|**

**| does pantry contain ingredient?**

**| yes**

**| was it ambiguous?**

**| yes**

**| ! behaviour undefined - throw an exception**

**| no**

**| was alternative instantiation method provided?**

**| yes**

**| return result of calling it**

**| no**

**| return result of default instantiation method**

**| no**

**| ! behaviour undefined - throw an exception.**

* **Modify IPantry so that the only version of Measure in it:**
  + **Returns TIngredient**
  + **accepts string “searchName”**
  + **accepts Func<TIngredient> instantiate**
* **Implement IPantry in Pantry**
* **Move the dictionary search code into the new method**
* **Rename strIngredient to searchName**
* **Update the code paths to reflect the logic above**
* **In ingredients.json, add, “mushrooms”: “1 small punnet”**
* **In SpagBol**
  + **Add mushrooms to the ingredients list, specifying an alternate preparation method to de-stem and grate them.**
* **Run your code**

**Part 11**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part11**

**Conceptual objectives**

* **Introduce - DIP - Dependency Inversion Principle**

**| Loading the json file in the Pantry class is a violation of SRP cuz the Pantry’s job**

**| is to lookup and instantiate Ingredients, not determine the source of the ingredient**

**| list itself.**

**|**

**| We need to fix that, but how? By bringing in another SOLID principle.**

**|**

**| what if this file is moved into a NoSql database and you have to**

**| query that for the data instead? The file is in the same format but you’ll still**

**| have to refactor DishBase to load from different source.**

**|**

**| this issue is the reason the Dependency Inversion Principle exists.**

**|**

**| SOLID principle #3 - DIP.**

**|**

**| High-level modules should not import anything from low-level modules.**

**| Both should depend on abstractions (e.g., interfaces).**

**| By decoupling the DishBase from the datasource and having it depend on an interface | instead of a literal file, we make the code resilient to changes in the**

**| implementation of the data source. They could change their db strategy and start**

**| using SQL server. As long as the implementation of the interface still returns a**

**| dictionary to the pantry, it won’t make any difference to DishBase.**

**Steps**

* **Create a new Class Library project in your solution called DataServices.**
* **Create another new Class Library project in your solution called Services**

**| What we now have is an implementation of the industry standard application structure**

**|**

**| Layer**

**| ----------------------------------------------------------------**

**| Presentation (display and interaction)**

**| Business objects (behavioural requirements)**

**| Business services (behavioural dependencies)**

**| Data services (data dependencies)**

**| ---v---^---v---^---v---^---v---^---v---^---v---^----------------**

**| External services - databases, APIs, files etc.**

**|**

* **Let all application layers except DataServices depend on Services.**
* **Let Services depend on DataServices**
* **In Services:**
  + **Create a new interface IRecipeService**

**| It will be the responsibility of this interface to define methods that query a data**

**| source transparently to the user of the service and return the data requested in the**

**| required format**

* **Add a method LoadPantry returning Dictionary<string, string>**

**| Each data source has its own terminology and “m.o.” so it would be a violation of**

**| ISP to attempt to define an interface that could be implemented for any data source**

**|**

**| e.g. to load a local file, no authentication is required but to use a secure API**

**| does. Therefore, the obligation to implement an Authentication method in a file**

**| data service would be a violation of the Interface Segregation Principle.**

* **In DataServices:**
  + **Create a new interface IFileDataService**
  + **This will define methods that perform generic file operations**
  + **Create a new class FileDataService implementing IFileDataService**
* **In IFileDataService**
  + **Add a method GetLines returning a List<string> and accepting a string “path”**
  + **Add a method GetContents returning string and accepting a string “path”**
* **Implement IFileDataService**
* **Remembering to use try-catch, call File.ReadAllText and File.ReadAllLines to return the results of the 2 methods**

**| the Dependency Inversion Principle (DIP) states that classes should depend on**

**| abstractions, not on details. This translates to making sure a class’s dependencies**

**| (i.e. elements beyond the scope of its single responsibility) are interfaces, not**

**| concrete classes.**

**|**

**| NOTE: implementing an interface is NOT the same as depending on one.**

**|**

**| class SomeClass : ISomeInterface ⇐ this is implementing**

**| {**

**| ISomeOtherInterface dependency; ⇐ this is depending  
| }**

* **In Services**
  + **Create CsvRecipeService implementing IRecipeService**
  + **Declare a private IFileDataService fileDataService**
  + **Generate a constructor to have its implementation injected**
  + **implement IRecipeService**
  + **In LoadPantry call fileDataService.GetLines(“ingredients.csv”)**
  + **Loop thru the resultant list of string, splitting into tokens and adding them to a dictionary which loadPantry then returns**
* **In Pantry**
  + **Declare a private IRecipeService recipeService and generate a constructor to have this injected.**
  + **After the local recipeService has been assigned, populate the “stock” dictionary with recipeService.LoadPantry();**
* **In DishBase** 
  + **In the constructor, instantiate the pantry, passing in a new CsvRecipeService, to which in turn you pass a new FileDataService.**
* **Have you code instantiate a new Recipe<SpagBol>() and call the Follow method.**

**Part 12**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part12**

**Conceptual objective:**

* **Witness the beauty of dependency inversion!**

**Functional objective**

* **Switch to a JSON-based pantry**

**Steps:**

* **In services create a new class JsonRecipeService implementing IRecipeService**
* **Add a private IFileDataService and have it injected thru the constructor as before**
* **In the LoadPantry method using JsonConvert.DeserializeObject to convert the result of fileDataService.GetContents(“ingredients.json”) to the dictionary return value.**
* **In DishBase, change the instantiation of Pantry to pass in JsonRecipeService instead.**
* **Test your code. Pretty cool, huh?**

**Part 13**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part13**

**Conceptual objectives:**

* **More with method references**
* **Introduction to indexers**

**Functional objectives:**

* **Leverage the Stage class to define more general steps in meal preparation**

**Steps:**

* **TBA**

**Part 14**

**Completed code location:**

**Kata/CustomTypes/Kata/Recipe/Part14**

**Conceptual objectives:**

* **Cursory introduction to Attributes**
* **Enums as booleans (Flags)**

**Functional objectives:**

* **Fix the preparation method problem**

**Steps:**

* **TBA**