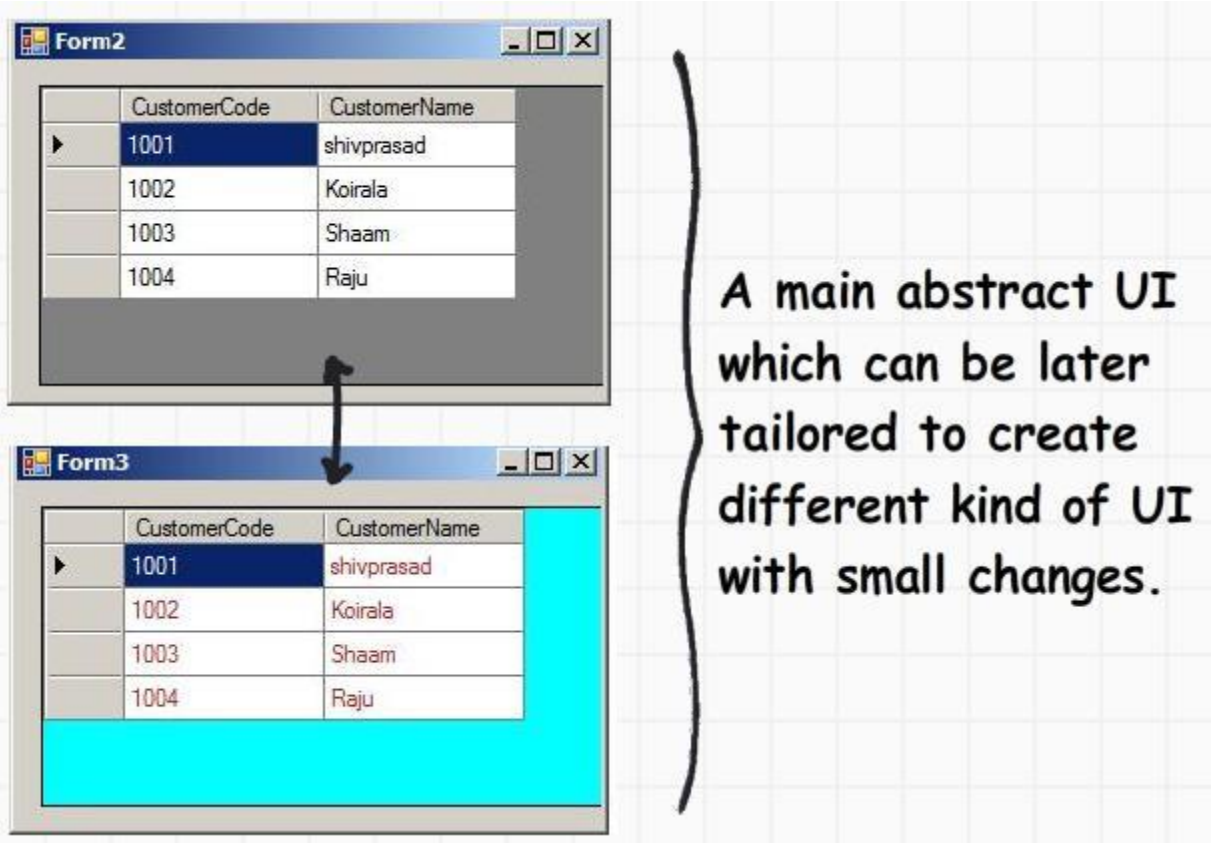
Class work: Kindly implement scenarios 2 or 3 or 6. (Only one)

Scenario 1: Flexible extendable generalized specialized user interfaces

Many times we come across UIs which look almost the same but with small differences in look and feel. For instance, in the below picture, the data is all the same for the screens, but the background colors are different.



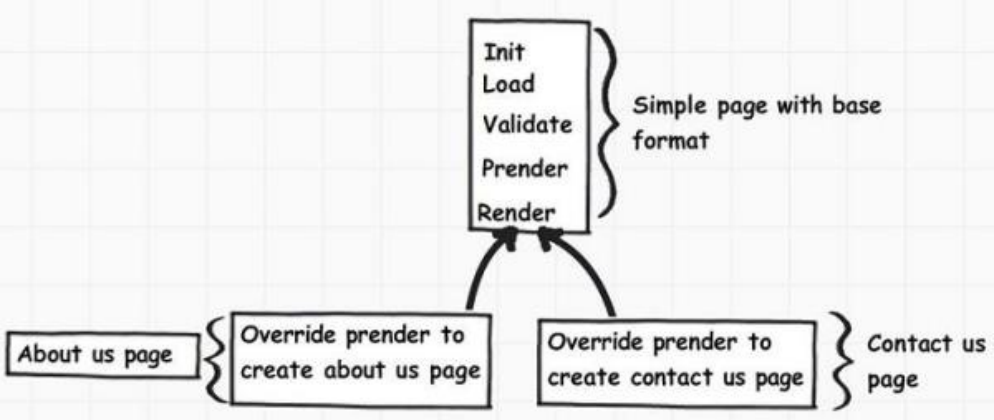
In this scenario, the form constructor will become the main process which will call three processes/functions:

- `InitializeComponent`: This will create the UI objects needed for the form.
- `LoadCustomer`: This function will load data and bind to the grid.
- `LoadGrid`: This function will define the look and feel of the grid.

Scenario 2: ASP.NET page life cycle

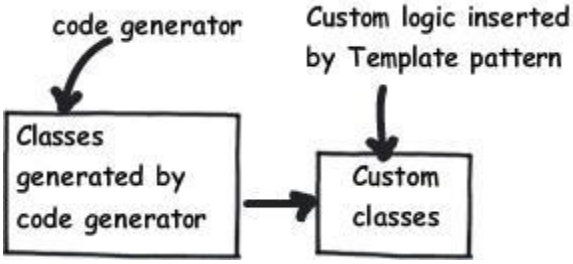
One scenario where we see the Template pattern very much visible is in the ASP.NET page life cycle. In the ASP.NET page life cycle, the life cycle sequence is fixed but it provides full authority to override the implementation for each sequence.

For instance, in the ASP.NET page life cycle, we have various events like Init, Load, Validate, Prerender, Render, etc. The sequence is fixed, but we can override the implementation for each of these events as per our needs.



Scenario 3: Code generators

Lots of times we generate code by using code generators like the T4 template, LINQ, EF, etc., from the database table design. Now code generators work on a separate physical file where it generates code for you. So if you change the table design, it will regenerate the file again.



Now if you want to add some custom code, you cannot change the auto generated code file because your code will be replaced when the DB design changes. So the best approach would be to extend the code generated class using a separate file and put your custom code in that class. This extension can be done very effectively using the Template pattern. The code generated class can define a fixed process but at the same time provide empty virtual methods, properties, and functions which can be extended to be injected to your custom logic.

Scenario 4: XML parser

Another scenario which is applicable for the Template pattern is the XML parser. In XML, we normally parse the parent and child elements. In many scenarios, the parsing is almost common with minor child element changes.

For instance, in the below code snippet, we have `Customer` as the parent element and every customer will have `Orders` and `Orders` will have `Product`s. Now the parsing of `Customer` and `Orders` will be the same but the `Product` tags can have a `Size` property depending on the situation. For instance, in the below code snippet, the `Product` element has only the name of the product and the amount.

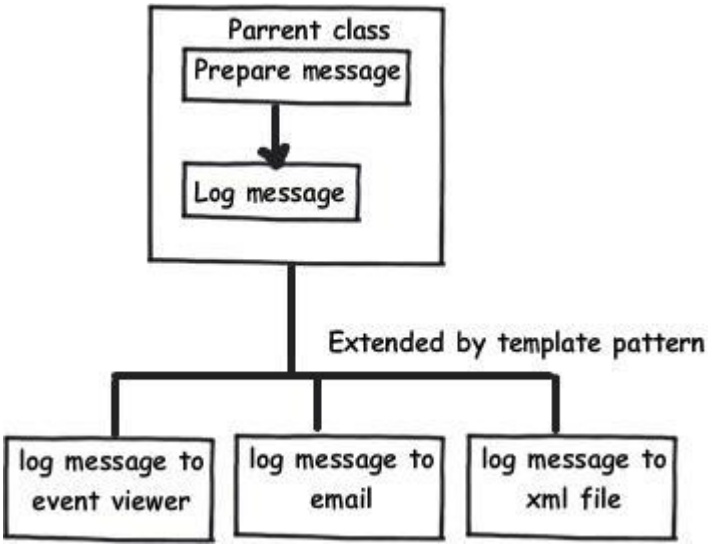
There can be situations where your `Product` element can have other variations as shown in the below XML snippet. In this case, you can just override the parsing process of the `Product` element and keep the overall XML parsing process sequence the same.

Scenario 5: Validation in business components

Business classes have validation and we would like to create different versions of business classes with different validation logic.

Scenario 6: Customizable logging utility

This is one more scenario where the Template pattern fits like anything. If you look at these components, i.e., message loggers, error loggers, etc., they execute in two phases, in the first phase they prepare the message and in the second phase they log it.
For these kinds of scenarios, we can create a parent class which defines two fixed sequences: one which does the preparation of the message and the other which logs the message to the source (file, event viewer, email, etc.).



Later we can create child classes which can inherit and override the logic of those phases but keeping the sequence intact.