Edge: Guidelines for self/peer code review

Common:

1. Don't do spelling mistakes and typos.
2. Descriptive and use meaningful names — clarity over brevity. i.e. avoid the use of x or y variables. Use variables names like “product” rather than ‘x’ or ‘p’. Do not worry about lengthy names they will be automatically shortened by a compiler/transpiler.
3. Grammar rules should be followed while naming variables and methods. Singular and plural names.
4. Avoid too many parameters in a method
5. Avoid optional parameters in methods
6. Avoid flag arguments. (https://martinfowler.com/bliki/FlagArgument.html)

Avoid this.

```
SendNotification(string msg, bool sendSMS, bool sendEmail, bool sentFax){
        if(sendSMS){
                messageService.sendSMS(msg);
        }
        if(sendEmail){
                messageService.sendEmail(msg);
        }
        if(sentFax){
                messageService.sentFax(msg);
        }
}

SendNotification("Hello John!!", false, false, true);
SendNotification("Hello John!!", true, false, true);

// In this situation code can be complex to maintain and read.
```

Better way - Create separate methods

```
SendSmsNotification(msg);
SendEmailNotification(msg);
SendFaxNotification(msg);
```

7. Always remove commented code
8. Don’t add unrelated files to commits (files which have no changes etc.)
9. The comments must be added for hacks and complex logic.

Useless comment

```
// iterating over sequence  <--- This is a useless comment
for(int i = 0, i < myList; i++){
        // code
}
```

10. Try to move the complex conditions to self-described functions. For e.g.

```
// check if meal is healthy or not
if (meal.calories < 1000 && hasVegetables) {
...
}

// could be refactored to

if (this.isHealthy(meal)) {
...
}

isHealthy(): boolean{
        return meal.calories < 1000 && hasVegetables;
}
```

11. **There should be exactly one line space between methods in a class.**
12. Inside the **methods, line breaks are optional**. But if we add one then please make sure it is of **one line space only**.
13. Rather than clutter logic in a single function, it should be split into **small pure functions**. It is also helpful in writing unit test codes.
    a. A pure function has the following characteristics:
        i. It always returns the same output for the same input
        ii. it produces no side-effect

Example of pure function and impure function

```
// Pure function
int foo(int n)
{
  return n*2;
}

// Impure function
int i = 42;

int bar(int n)
{
  ++i;
  return n*i;
}
```

angular/typescript:

1. The folder structure should be followed. Split the application in independent small modules. e.g.

```
|-- modules
| |-- home
| | |-- home.spec|module|component|scss|routing.module|.ts
| |-- about
| | |-- about.spec|module|component|scss|routing.module|.ts
```

2. File names should be consistent and describe the feature by dot separation. e.g. **my-feature.component.ts and my-service.service.ts**
3. Avoid the use of **any** type. If any type can't be avoided, a comment should be added to it why it is needed.
4. Always have **return type** defined.
5. **Async and await** is preferred for single calls. This way we can avoid unnecessary subscriptions.
6. Always prefer '**const**'. **'var' is strictly prohibited**. Use '**let**' only when it is reassigned in the given method.
7. Implement typescript data models as an interface, not as classes. We generally have the prefix 'I' in interfaces but in angular, we should not use it unless these interfaces have method signatures. That way we can distinguish between data models and other application interfaces.
8. For optional variables, they must be marked optional, like:

```
interface MyInterface
{
    myProperty: string;
    myOptionalProperty: string | undefined; //or
    myAnotherOptionalProperty?: string
}
```

Another important example could be

```
foo(obj: MyType): void {
        if(!obj){
                return;
        }
        // code..
}
// Problem: obj argument is not optional but the code still checks its existence/validity which confuses
other team members.
// Solution: This check is not required unless obj is optional i.e. obj?: MyType or obj: MyType |
undefined
```

9. Always define property as "undefined" if they are not initialized in the constructor or during the declaration. Or, if possible, try assigning a default value to such variables.

```
export class MyClass {
        enabled : boolean | undefined;
        isLoaded = false; // better than line #2

        constructor() { }
}
```

10. For this code:

```
return this.tabHasError ? true: false;
=>
return this.tabHasError;
```

```
this.codingTemplates.length > 0 ? false : true
=>
!this.codingTemplates.length
```

11. The code should be **prettified with prettier**.
12. Always sort and remove unused **imports.** Shortcut: Shift+Alt+O
13. Use filter RxJs operator when possible like:

```
budgetDetailService.budgetDetail$
        .pipe(filter(budgetBasicData =>
            budgetBasicData !== undefined && budgetBasicData.budgetBasicDataDTO.status ===
BudgetAccountStatus.Removed)
        .subscribe(isRemoved => { //code....  });
```

14. Use async pipe in HTML templates in order to avoid subscriptions. As such subscriptions are destroyed by the angular framework itself.
15. Avoid memory leaks by destroying the instance of subscription.

```
  private destroy$: Subject<void> = new Subject<void>();
  ...
   this.itemService.findItems()
    .pipe(
       map(value => value.item)
       takeUntil(this.destroy$)
     )
    .subscribe();
  ...

  public ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
    this.destroy$.unsubscribe();
  }
```

16. Always declare barrels (index.ts file) so that shorten paths can be used. This will make changes in the folder level easier.

```
// src/app/shared/components/index.ts
export * from './reusable/reusable.component.ts';
// src/app/shared
export * from '/components';
```

17. Use guard methods.

```
function something(): void{
        if(false) {
      return; // return early
    }
          // code….
}
```

Tips: Try to use angular CLI for creating new services, components, or routers.

C#:

1. Microsoft recommends the Upper Camel case for naming. (PascalCase) i.e. **MyClass**, **HelloWorldFunc**.
2. The interface should start with letter capital “I”. e.g. **ISaveHeaderAction**, **IProcessOrder**
3. Private variables should be in lower camel case. e.g. **version**, **taskId**.
4. Public Fields and Properties must be in the Upper camel case. e.g.

```
public Guid PrSourceLineId { get; set; }
public Guid Version;
```

5. Constants should be in the Upper Camel case. e.g. **DeliveryEndDate**, **DeliveryDate**.
6. The reference variables used in constructor injection must be **private and readonly**. e.g.

```
private readonly IValidationRunner validationRunner;  // Should be in lower camel case
private readonly IAnyErpValidator anyErpValidator;

public SaveHeaderDataAction(IValidationRunner validationRunner, IAnyErpValidator anyErpValidator)
{
    this.validationRunner = validationRunner;
    this.anyErpValidator = anyErpValidator;
}
```

7. Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields.
8. **use var** instead of type while declaring variables.

9. Always sort and remove unused **namespace imports.** Shortcut: Ctrl + R + G
10. Avoid long lines. As a code reviewer, it is easy to see code in git's stash if there is **no horizontal scroll bar**.

```
public IPanelSaveAction CreatePanelSaveActionForTask(Guid requisitionId, Guid version, string taskId,
RequisitionState requisitionState, bool isProToolsUser, IEnumerable<ModifiedField> modifiedFields)
```

can be formatted to

```
public IPanelSaveAction CreatePanelSaveActionForTask(
            Guid requisitionId,
            Guid version,
            string taskId,
            RequisitionState requisitionState,
            bool isProToolsUser,
            IEnumerable<ModifiedField> modifiedFields)
```

11. Switch case is more readable than if-else-if statements.

Readable switch case

```
if (status == 0)
{
        return BudgetAccountStatus.Inactive;
}
else if (status == 1)
{
        return BudgetAccountStatus.Active;
}
else
{
        return BudgetAccountStatus.Removed;
}
```

can be refactored to

```
switch (status)
{
        case 0:
                return BudgetAccountStatus.Inactive;
        case 1:
                return BudgetAccountStatus.Active;
        default:
                return BudgetAccountStatus.Removed;
}
```

12. The excessive code indentation makes code unreadable.

Wrong formatting

```
return data.BudgetBasicDataDTO.Users.Select(user =>
                                                    new BudgetUserDataModel
                                                    {
                                                        Id = user.Id,
                                                        UserId = user.UserId,
                                                        UserName = user.UserName
                                                    });// this code has unnecessary code indentation
```

should be like this:

```
return data.BudgetBasicDataDTO.Users
             .Select(user => new BudgetUserDataModel
             {
                Id = user.Id,
                UserId = user.UserId,
                UserName = user.UserName
              });
```

13. Better to return early. We should try to write code outside the braces. for e.g.

```
private void MyFunc(object data)
{
        if (data != null)
        {
                // code...
        }
}
```

could be refactored to

```
private void MyFunc(object data)
{
        if (data == null)
                return;
        // code...
}
```

This way we can avoid unnecessary code indentation.

14. Using a Dictionary as an alternate to switch case if there is a key-value relation between input and output. Consider this example:

```
private static Dictionary<CrossRefExceptionTypes, string> mapping = new
Dictionary<CrossRefExceptionTypes, string>()
{
   { CrossRefExceptionTypes.DataExists, ImportErrorIdentifier.DataExists},
   { CrossRefExceptionTypes.InvalidData, ImportErrorIdentifier.InvalidData},
   { CrossRefExceptionTypes.InvalidTemplate, ImportErrorIdentifier.InvalidTemplate},
   { CrossRefExceptionTypes.DuplicateClassificationCode, ImportErrorIdentifier.
DuplicateClassificationCode},
   { CrossRefExceptionTypes.InvalidFile, ImportErrorIdentifier.InvalidFile},
};

private static ImportResultDTO GetErrorResponse(CrossRefParsingException ex)
{
   return new ImportResultDTO(false)
   {
           ErrorIdentifier = mapping.ContainsKey(ex.ExceptionMessageType)
                        ? mapping[ex.ExceptionMessageType] : ImportErrorIdentifier.Unknown
   };
}
```

15. When we throw known exceptions at multiple locations, consider this code block. It can reduce code complexity and make the code more readable.

```
public static class PoEditing
{
    public static void ThrowIfSaveNotAllowed(
        List<ValidationMessage> validations)
    {
        if (validations.Any(v => v.MessageType == EdgeMessageType.Error))
        {
            throw new ValidationException(validations);
        }
    }
}
// Usage:
PoEditing.ThrowIfSaveNotAllowed(validations);
```

16. If the method return type is IEnumerable<string> we shouldn't do .ToList(). Either return type IEnumerable and return enumerable or we return IList and do .ToList(). This will help the consumer to know that if the data is already into the memory or it is a deferred query.
17. Consider these code blocks, with ToList() we can reuse resultant, whereas without ToList() the result is re-evaluated each time we use LINQ static methods. However, one needs to be careful while using this practice as this can improve or decrease the application performance.

Without ToList()

```
var data = new List<string>() { "Test",
"Test2" };
var result = data.Select(value =>
{
    Console.WriteLine("inside select
func.");
    return value;
});
if (result.Any())
{
    var output = result.Count() == 1
        ? $"Single entry: {result.
ElementAt(0)}"
        : $"Multiple entries: {string.
Join(", ", result)}";
    Console.WriteLine(output);
}

// Output
inside select func.
inside select func.
inside select func.
inside select func.
inside select func.
Multiple entries: Test, Test2
```

With ToList()

```
var data = new List<string>() { "Test",
"Test2" };
var result = data.Select(value =>
{
    Console.WriteLine("inside select
func.");
    return value;
}).ToList();
if (result.Any())
{
    var output = result.Count() == 1
        ? $"Single entry: {result.
ElementAt(0)}"
        : $"Multiple entries: {string.
Join(", ", result)}";
    Console.WriteLine(output);
}

// Output
inside select func.
inside select func.
Multiple entries: Test, Test2
```

18. **LINQ** is more readable than traditional for/foreach. Consider this example.

```
IEnumerable<string> GetExpenisveProducts()
{
        var filteredInfos = new List<string>();
        foreach(Product product in Products)
        {
                if (product.UnitPrice > 75.0M)
                {
                        filteredInfos.Add($"{product.ProductName} - ${product.UnitPrice}"));
                }
        }
        return filteredInfos;
}
```

can be re-written using **LINQ**

```
IEnumerable<string> GetExpensiveProducts()
{
        return from product in Products
                where product.UnitPrice > 75.0M
                select $"{product.ProductName} - ${product.UnitPrice}";
}

or,

IList<string> GetExpensiveProducts()
{
    return Products
                .Where(product => product.UnitPrice > 75.0M)
                .Select(product=> $"{product.ProductName} - ${product.UnitPrice}")
                .ToList();
}
```