
# WRITING QUALITY CODE

Ruwan Wijesinghe
&
Niruka Ruhunage

virtusa | POLARIS

TIQRI

ICTA
ideas actioned

# Why Maintainability

- If code is not written well
  - It might work well
  - However it is not maintainable
    - Difficult to fix defects
    - Difficult to change
    - Difficult to add new features

- From client's perspective, bad code would result in
  - Higher maintenance cost
  - Cannot be changed with the changing business needs
  - Most application has to be re-written after some time, due to bad code

- From developer's perspective, bad code would make them
  - Less productive
  - Frustrated, by having to do deal with complex mess everyday

# What makes the code maintainable

- Simple
- Readable
- Easy to Change
- Has good diagnostic infrastructure
- Has good Unit Test coverage


virtusa | POLARIS™ TIQRI ICTA ideas actioned

# What makes the code simple

- Keep it simple and stupid (KISS)
  - Best code is not the shortest code
  - Best code is not the highest performant code
    (pre-mature performance optimizations may have negative impact on both performance and maintainability)
  - Best code is the code that anybody can understand easily.

- You aren't Going to need It (YANGI)
  - Don't add complexity to cater for future requirements
  - Don't use design patterns just to cater for unseen future requirements or sake of using them

# What makes the code Readable

- Use meaningful names
- Follow consistent naming convention
- Use small and simple methods
- Properly format the code
- Add proper code comments
- Separate different concerns into deferent classes


virtusa | POLARIS™ TIQRI ICTA ideas actioned

# Use meaningful names

- Class names, method names and variable names should be meaningful, so that the code can describe itself without the need of comments
- Method names should correctly represent exactly what the method does (no more, no less)
  - Never call a method ‘getOrder’, if it creates a new order if one does not exists
- Class and variable names should correctly represent exactly what the  class, variable represents
  - Never use variable names such as x, y, i, j, obj, emp1, emp2, etc.
- Use nouns to name classes and variables (e.g. class – Order,  variable – order)
- Use verbs to name methods (e.g. getName, setAddress, save, createEmployee, etc.)

# Follow a consistent naming conventions

- Use consistent verbs to create method names
  - A method with ‘get’ prefix should never alter the state and should always return a value
  - A method with ‘set’ prefix should always modify the sate and should never return a value
  - Never use different prefixes to represent the same action

    (e.g. never use below methods to indicate that some entity is inserted to the database, within the same application)

    insertOrder,  createInvoice, saveEmployee, addCatalogItem
- Never use two words to mean the same thing (same concept) within the same application
- Never use same word to mean multiple things within the same application

# How to make methods looks simple

- Remove nested conditions
- Remove control flags
- Introduce descripting local variables
- Split temporary variables
- Break long methods into small methods

# Simple Methods- Remove Nested Conditions

- A method with nested conditions

```
public decimal GetCallCharges(CallInfo callInfo)
{
    decimal charge;

    if (callInfo != null)
    {
        if (callInfo.IsIDDCall)
        {
            charge = GetIDDCallCharges(callInfo);
        }
        else
        {
            if (callInfo.IsPeakTime())
               charge = GetPeakTimeCallCharges(callInfo);
            else
               charge = GetNormalCallCharges(callInfo);
        }
    }
    else
    {
        throw new ArgumentNullException(nameof(callInfo));
    }

    return charge;
}
```


virtusa | POLARIS™ TIQRI ICTA ideas actioned

# Simple Methods- Remove Nested Conditions

- A method with guard clause

```
public decimal GetCallCharges(CallInfo callInfo)
{
    if (callInfo == null)
        throw new ArgumentNullException(nameof(callInfo));

    if (callInfo.IsIDDCall)
        return GetIDDCallCharges(callInfo);

    if (callInfo.IsPeakTime())
        return GetPeakTimeCallCharges(callInfo);

    return GetNormalCallCharges(callInfo);
}
```

# Simple Methods - Remove Control Flags

- A method with control flags

```
public string GetUsername(int userId)
{
    User requiredUser = null;
    bool userFound = false;
    for (int i = 0; i < users.Count && !userFound; i++)
    {
        if (users[i].UserId == userId)
        {
            requiredUser = users[i];
            userFound = true;
        }
    }

    if (userFound)
        return requiredUser.Name;
    else
        return null;
}
```

# Simple Methods - Remove Control Flags

- A method with no control flags

```
public string GetUsername(int userId)
{
    foreach (var user in users)
    {
        if (user.UserId == userId)
        {
            return user.Name;
        }
    }

    return null;
}
```

# Simple Methods - Descripting local variables

- A lengthy expression

```
double totalArea = triangle.width * triangle.height / 2 + rectangle.width * ractangle.height;
```

- After descriptive variables are introduced

```
double areaOfTriangle = triangle.width * triangle.height / 2;
double areaOfRectangle = rectangle.width * ractangle.height;
double totalArea = areaOfRectangle + areaOfRectangle;
```

# Simple Methods - Split Temp Variables

- Code that reuse the same variable to represent two logical concepts

  E.g. Area of triangle & area of rectangle

```
double totalArea = 0;
double area = triangle.width * triangle.height / 2;
totalArea += area;
double area = rectangle.width * ractangle.height;
totalArea += area;
```

- Code after the temp variable is split

```
double areaOfTriangle = triangle.width * triangle.height / 2;
double areaOfRectangle = rectangle.width * ractangle.height;
double totalArea = areaOfRectangle + areaOfRectangle;
```

# Break long methods into small methods

- Number of lines in a method should not exceed the number of lines that can be viewed using your IDE
- If a method is too long or too complex, break it into small methods by extracting a set of small private methods with the same abstraction level
- The initial method should represent the high level algorithm as a story
- Subsequent methods should represent the detailed implementation
- If the is harder to break a method, since its internal logic is tightly coupled through lots of local variables,
  - Extract the method into a separate class
  - Convert local variables that cause coupling into member variables of that class
  - Break the method (extract private methods that are in same abstraction level)

# Break long methods into small methods (e.g.)

```
public TransactionReponse Withdraw( string accountNo, decimal amount, int userId)
{
    var account = accountRepository.GetAccount(accountNo)
    if (account.Balance >= amount)
    {
        transactionRepository.AddTransaction(
            new Transaction(accountNo, -amount, userId, DateTime.UtcNow);
        account.Balance -= amount;
        accountRepository.Save(account);
        return new TransactionReponse(TransactionStatus.Completed);
     }
     else
     {
         var odStratergy = odStratergyFactory.GetStratergy(account.AccountType);
         if (odStratergy.CanIssueOD(account))
         {
             decimal preApprovedCreditLimit = odStratergy.GetPreApprovedCreditLimit(account);
             if (preApprovedCreditLimit < amount)
             {
                 odRepository.AddOD(
                     new OverDraft(accountNo, amount, userId, DateTime.UtcNow));
                 transactionRepository.AddTransaction(
                     new Transaction(accountNo, -amount, userId, DateTime.UtcNow);
                 account.Balance -= amount;
                 accountRepository.Save(account);
                 return new TransactionReponse(TransactionStatus.Completed);
             }
         }
         return new TransactionReponse(TransactionStatus.Rejected, StringResources.InsufficientFunds);
    }
}
```


virtusa | POLARIS™ TIQRI ICTA ideas actioned

# Break long methods into small methods (e.g.)

```
public TransactionReponse Withdraw( string accountNo, decimal amount, int userId)
{
        var account = GetAccount(accountNo);
        if (account.Balance >= amount)
        {
                AddTransaction(account, -amount, userId);
                return new TransactionReponse(TransactionStatus.Completed);
        }
        else
        {
                ProcessAsOD(accountNo, amount, userId);
        }
}
```

# Break long methods into small methods (e.g.)

```
private TransactionReponse ProcessAsOD( string accountNo, decimal amount, int userId)
{
        var odStratergy = GetODStratergy(account.AccountType);
        if (odStratergy.CanIssueOD(account))
        {
                decimal preApprovedCreditLimit = odStratergy.GetPreApprovedCreditLimit(account);
                if (preApprovedCreditLimit < amount)
                {
                        RecodeOD(account, amount, userId);
                        AddTransaction(account, -amount, userId);
                        return new TransactionReponse(TransactionStatus.Completed);
                }
        }

        return new TransactionReponse(TransactionStatus.Rejected, StringResources.InsufficientFunds);
}
```


virtusa | POLARIS™ TIOBI ICTA ideas actioned

# Break Methods - Use methods objects

Use method objects to break complex methods that are tightly coupled through local variables

See the below example

```
public void ComplexMethod()
{
    //Load Data
    var initialData = LoadInitialData();
    var secondaryData = LoadSecondaryData(initialData);

    //Validate secondry data
    //...

    int totalProp1 = 0;
    int averageProp2 = 0;
    int minProp3 = 0;
    foreach (var item in secondaryData)
    {
        //calculate summary
        //...
    }

    //calculate return value using summary result
    //... res = some expression of totalProp1, averageProp2 and minProp3
    return res;
}
```


virtusa | POLARIS™ TIOBE ICTA ideas actioned

# Break Methods - Use methods objects

```
private class ComplexMethod
{
    public IList<int> InitialData { get; set; }
    public IList<MyEntity> SecondaryData { get; set; }
    public decimal TotalProp1 { get; set; }
    public decimal AverageProp2 { get; set; }
    public decimal MinProp3 { get; set; }

    public decimal ComplexMethod3()
    {
        LoadData();
        Validate();
        CaclulateSummeries();
        return CalculateResult(parameters);
    }

    //Private methods comes here
}
```

# Properly Format the Code

- Vertical Formatting
  - Let the users to read a class in summary and details format
    - Keep constants and member variables of the class at the top, to make them easy to find
    - Public interface methods that describe the high level flows (algorithms) of the actions should come next
    - Private methods with low level implementation details should follow them
  - Higher the density of information in the screen, higher the readability
    - Reduce the number of empty lines
    - Never add unnecessary comments

- Horizontal Formatting
  - Break the statements into multiple lines if they are too long to be seen in a single screen in your IDE
  - Break the statements in logical places so that the readability of the code is not impacted

# Add Proper Comments

- Say WHY, Not WHAT
  - What is being done is visible from the code (proper code should be self explanatory without comments)
  - Use comments to say why you did it in that way, when it is not very obvious

- Below are the valid exceptions to above rule
  - Add comments as warnings to other programmers on the consequences of possible changes that they might do in future
  - Add comments to emphasis the importance of some actions in the code, that otherwise looks not so important
  - Add comments to indicate further actions needed (TODO:), etc.

- Remove bad comments
  - Redundant comments increases the code length and hence the readability
  - Inaccurate (not properly updated) comments can be very harmful
  - Commented code reduces the readability (act as Zombie code that nobody dare to remove later)

# What makes the code easy to change

- One should be able to read, understand and change each code unit, module, etc. without having to analyze the rest of the code base.
  - Use small cohesive classes
  - Encapsulate the implementation logic and expose only a stable interface

- There should be only one unit that needs to be changed to accommodate a singe change in the requirements
  - Ensure that the application is broken into cohesive set of classes
  - Avoid duplicate logic (DRY – Don’t Repeat Yourself principle)
  - Avoid hard-corded constants

# Use small and cohesive classes

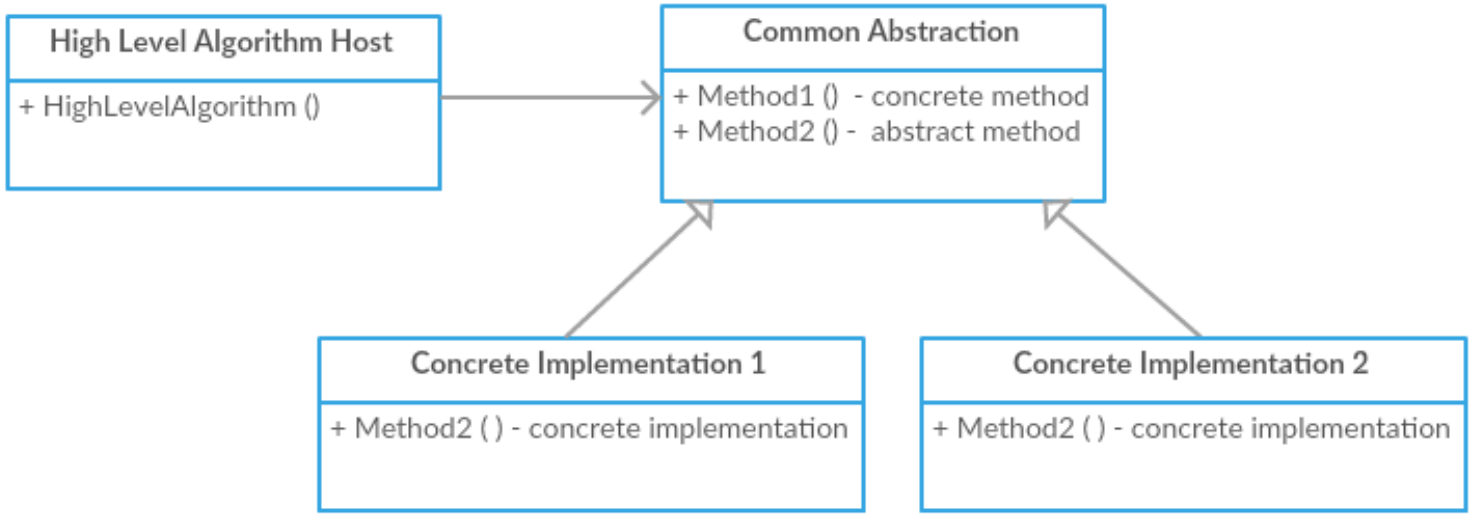
- Create classes that does only one thing and does it completely
- Break application functionality into classes so that any functional change can be achieved by changing minimum number of classes
- Create classes that represent domain concepts
- Create classes to represent domain commands when there is complex logic associated with processing them
- Always wrap all calls to external services and components in separate classes, so that the rest of the code remains stable from the changes in external dependencies

# How to encapsulate the implementation

- Encapsulate an implementation of an algorithm within a method
  - When possible, make output of a method to only depend on its inputs
  - When possible, make the output of a method to only depend on its input and immutable state of the containing object
- Encapsulate the internal data structures and implementation logic of classes using basic OOP features of your programming language (e.g. private and protected keywords)
- Encapsulate the implementation details of software components using façade pattern

# Avoid duplicate logic

- Shared Algorithms can be classified into two types based on how they are handled
  - Technical algorithms such as  encryption, encoding, searching, sorting, formatting, mathematical calculations, etc.
  - High level algorithms that operate on common abstractions
  - Concrete implementations shared between instances of a common abstractions



# Avoid duplicate logic

- Create static helper methods to implement technical algorithms

- Share logic between entities that can be derived from a common abstraction
  - Use inheritance when there is a single common shared abstraction
  - Use strategy design pattern when there are multiple such common abstractions (use composition over inheritance)

- Share high level algorithms that operate on shared abstraction
  - Use polymorphism to create shared logic that can operate on common abstraction
  - Use template design pattern to share high level algorithms within an inheritance hierarchy

# Has good Diagnostic Infrastructure

- Input Validation
- Exception Handling
- Logging
- Monitoring


virtusa | POLARIS™ TIQRI ICTA ideas actioned

# Input Validation

- Input validations makes the integration of libraries simple
- Proper validations helps us to detect the true cause of the errors, without debugging
- Ensure all input parameters are validated to be not null, if null is not an acceptable input value
- Ensure all values returned by all internal method calls are validated to be not null, if null value is not an acceptable value
- Ensure all exceptions that can be arise due to invalid inputs (e.g. FileNotFound exception) are handled and a friendly custom exception that explains the cause is re-thrown

# Exception Handling

- Prefer Exceptions over Error Codes

- At application or thread boundaries
  - Catch all exceptions
    - Log all exceptions
    - Generate a meaningful message for a known exception
    - Generate a generic message for unknown exceptions
    - Display message to user if this is a UI application
    - Return the error message to client if this is a service application

- At all other places
  - Catch known exceptions and re-throw custom exceptions with meaningful messages with sufficient diagnostic details

# Logging

- Look from the point of view of person who diagnose a problem, when deciding what to log
- Log sufficient details to diagnose problems
- Log messages does not require localization (unless requested by the client)
- Log context details (request parameters, etc.) together with exceptions, whenever possible
- Log information such as thread id, request id, session id, etc. to correlate the log entries
- Make the log level configurable
- Ensure that no unhandled exception can go without getting logged
- Always log errors even in production environment

# References


- ‘Clean Code: A Handbook of Agile Software Craftsmanship’ – By Robert C. Martin
- ‘Refactoring: Improving the Design of Existing Code’ – By Martin Fowler
- ‘Design Patterns: Elements of Reusable Object-Oriented Software’ – by Erich Gamma, Richard Helm, Ralph Johnson], John Vlissides



virtusa | POLARIS™ TIQRI ICTA ideas actioned

# Thank You

Ruwan Wijesinghe

&

Niruka Ruhunage