CLEAN CODE

CODE DESIGN

Code design
Agenda

- ✓ **Object and Data Structure**
- ✓ **Class**
- ✓ **Error handling**
- ✓ **Boundaries**
- ✓ **Unit tests**

BOSCH

Clean Code
Abstraction and Encapsulation

- **Abstraction :** Process in which you collect or gather relevant data and remove non-relevant data
- **Encapsulation:** Process in which you wrap of functions and members in a single unit

Clean Code
Abstraction in C

private.c

```
struct Contact
{
    int mobile_number;
    int home_number;
};
struct Contact * create_contact()
{
    struct Contact * some_contact;
    some_contact = malloc(sizeof(struct Contact));
    some_contact->mobile_number = 12345678;
    some_contact->home_number = 87654321;
    return( some_contact );
}
void delete_contact( struct Contact * some_contact )
{
    free(some_contact);
}
```

private.h

```
struct Contact;

struct Contact * create_contact();

void delete_contact( struct Contact * some_contact );
```

main.c

```
#include "private.h“
#include <stdio.h>
void main()
{
    struct Contact * Tony;
    Tony = create_contact();
    printf( "Mobile number: %d\n", Tony->mobile_number);
    delete_contact( Tony );
}
```

Clean Code
Encapsulation in C

Area.c

```
Class Rectangle {
Public :
            int length;
            int breadth;

            int getArea()
            {  return length * breadth;
            }

};
```

Clean Code
Object and Data Structure

- **Data Structure** class reveals or exposes its data (variables) and have no significant methods or functions.
- **Object Structure** class conceals their data, and reveals or exposes their methods that work on those data.

```
public class Square
{
    public Point topLeft1;
    public double side1;
}
```

```
public class Geometry
{
    private final double PI = 3.141592653589793;
    public double area(Object shape)
    {
        // do something
    }
}
```

Clean Code
Object and Data Structure

Case 1: Code using data structures makes it easy to add new functions without changing the existing data structures.

Clean Code
Object and Data Structure

```
public class Square
{
      public Point topLeft;
      public double side;
}

public class Rectangle
{
      public Point topLeft;
      public double height;
      public double width;
}
```

Use Case 1

```
public class Geometry
{
      public final double PI = 3.141592653589793;
      public double area(Object shape)
      {
            if (shape instanceof Square)
            {
                  Square s = (Square)shape;
                  return s.side * s.side;
            }
            else if (shape instanceof Rectangle)
            {
                  Rectangle r = (Rectangle)shape;
                  return r.height * r.width;
            }
       }

       public double perimeter(Object shape)
       {
             // do some thing
       }

 }
```

Clean Code
Object and Data Structure

Use Case 1

```
public class Square
{
      public Point topLeft;
      public double side;
}

public class Rectangle
{
      public Point topLeft;
      public double height;
      public double width;
}

public class Circle
{
      public Point center;
      public double radius;
}
```

```
public class Geometry
{
      public final double PI = 3.141592653589793;

      public double area(Object shape)
      {
            if (shape instanceof Square)
            {
                  Square s = (Square)shape;
                  return s.side * s.side;
            }
            else if (shape instanceof Rectangle)
            {
                  Rectangle r = (Rectangle)shape;
                  return r.height * r.width;
            }
       }
}
```

Clean Code
Object and Data Structure

Case 2: Object oriented structure, makes it easy to add new classes without changing existing functions.

Clean Code

```
public class Square
{
     private Point topLeft;
     private double side;
     public double area()
     {
         return side * side;
     }
}
public class Rectangle
{
     private Point topLeft;
     private double height;
     private double width;
     public double area()
     {
         return height  * width;
     }
}
public class Circle
{
     private Point center;
     private double radius;
     public final double PI = 3.141592653589793;
     public double area()
     {
          return PI * radius * radius;
     }
}
```

Use Case 2

Clean Code

```
public class Square
{
      private Point topLeft;
      private double side;
      public double area()
      {
            return side * side;
      }
      public double perimeter()
      {
            // do something
      }
}

public class Rectangle
{
       private Point topLeft;
       private double height;
       private double width;
       public double area()
       {
             return height * width;
       }
       public double perimeter()
       {
            // do something
       }
}
```

Use Case 2

BOSCH

Clean Code
Object and Data Structure

- **Prevent exposing details of the data** instead express data in abstract terms.
- **Objects Structures** hides data behind abstractions and exposes functions.
- **Data structures** exposes data and have no significant functions.
- **Decision on choosing Objects Structure and Data Structure**.

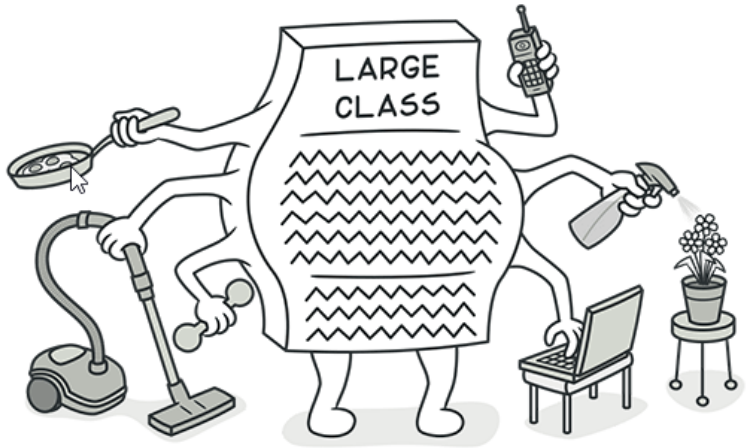
CLASSES

Clean Code
Classes

Classes Should Be Small!

A class can contains many fields/methods/lines of code.

Classes usually start small. But over time, they get bloated as the program grows!



Clean Code
Classes

```
public class Employee
{
    public string Name { get; set; }
    public string Address { get; set; }
    ...
    public void ComputePay() { ... }
    public void ReportHours() { ...}
}
```

Finance Team : I Want to change Condition for Payout??

Operations: I Want to change Condition for Login/Logout??

Clean Code
Classes

SRP --- Single Responsibility Principle.

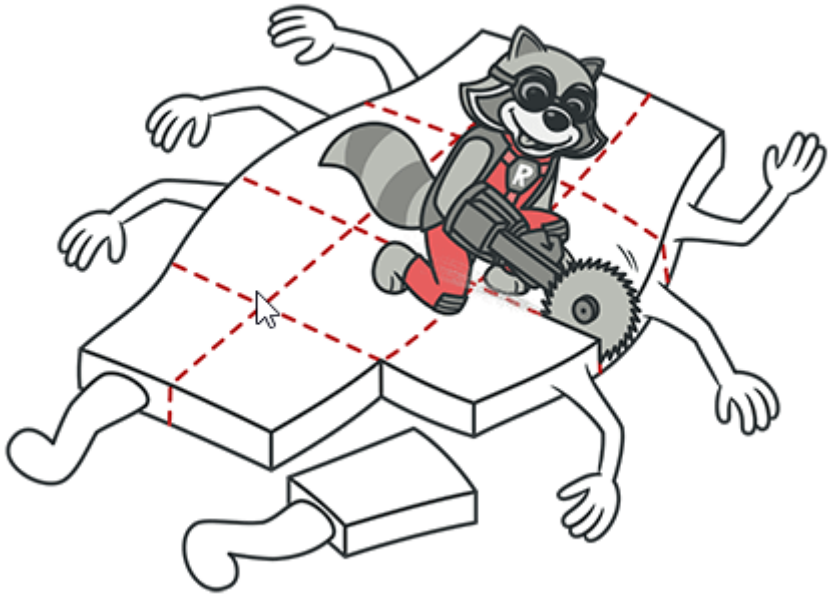
Classes should have one responsibility — one reason to change!!

```
public class Employee
{
    public string Name { get; set; }
    public string Address { get; set; }
    ...
    public void Create() { ... }
    public void Update() { ...}
}
public class Payroll
{
    public int BaseSalary { get; set; }
    public int Allowance { get; set; }
    ...
    public void ComputePay() { ... }
}
```

Clean Code
Classes

Treatment --- **Refactoring**

- ✓ Extract Class
- ✓ Extract Subclass



Clean Code
Classes

```
public class Employee
{
    public string Name { get; set; }
    public string Address { get; set; }
    ...
    public void Create() { ... }
    public void Update() { ...}
}
public class Intern : public Employee
{
    public string Name { get; set; }
    public string Address { get; set; }
    ...
    public void Create() { ... }
    public void Update() { ...}
}
```

Clean Code
Classes

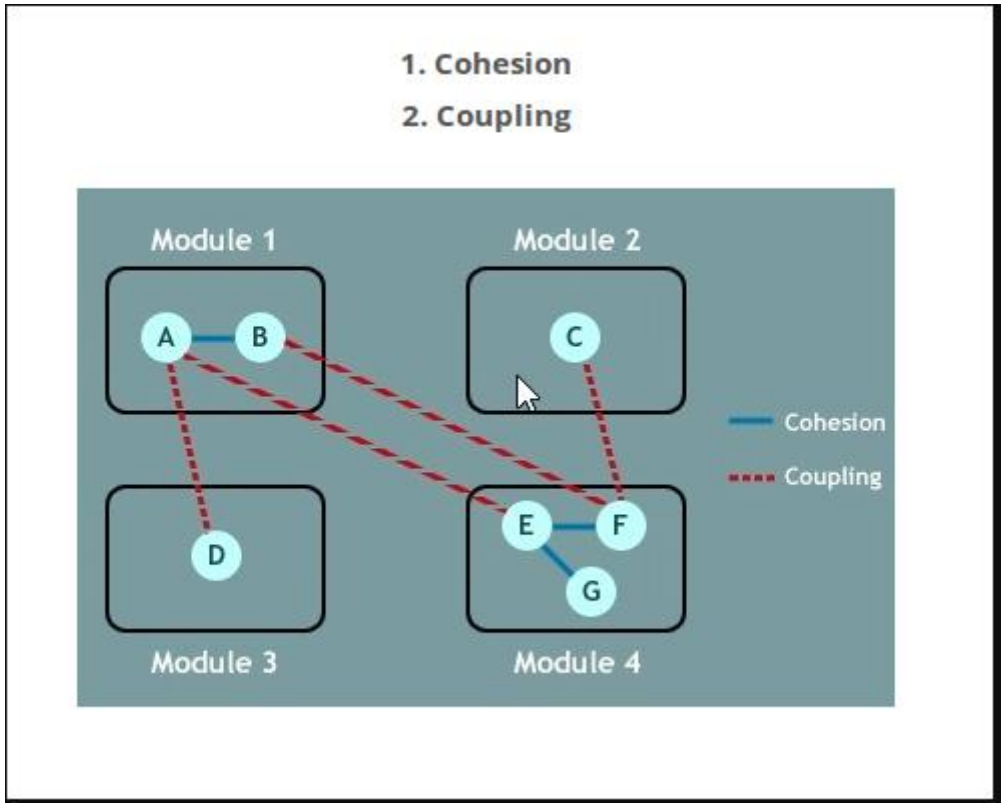
Payoff

- ✓ Refactoring spares developers from needing to remember a large number of attributes for a class.
- ✓ Splitting large classes into parts avoids duplication of code and functionality.

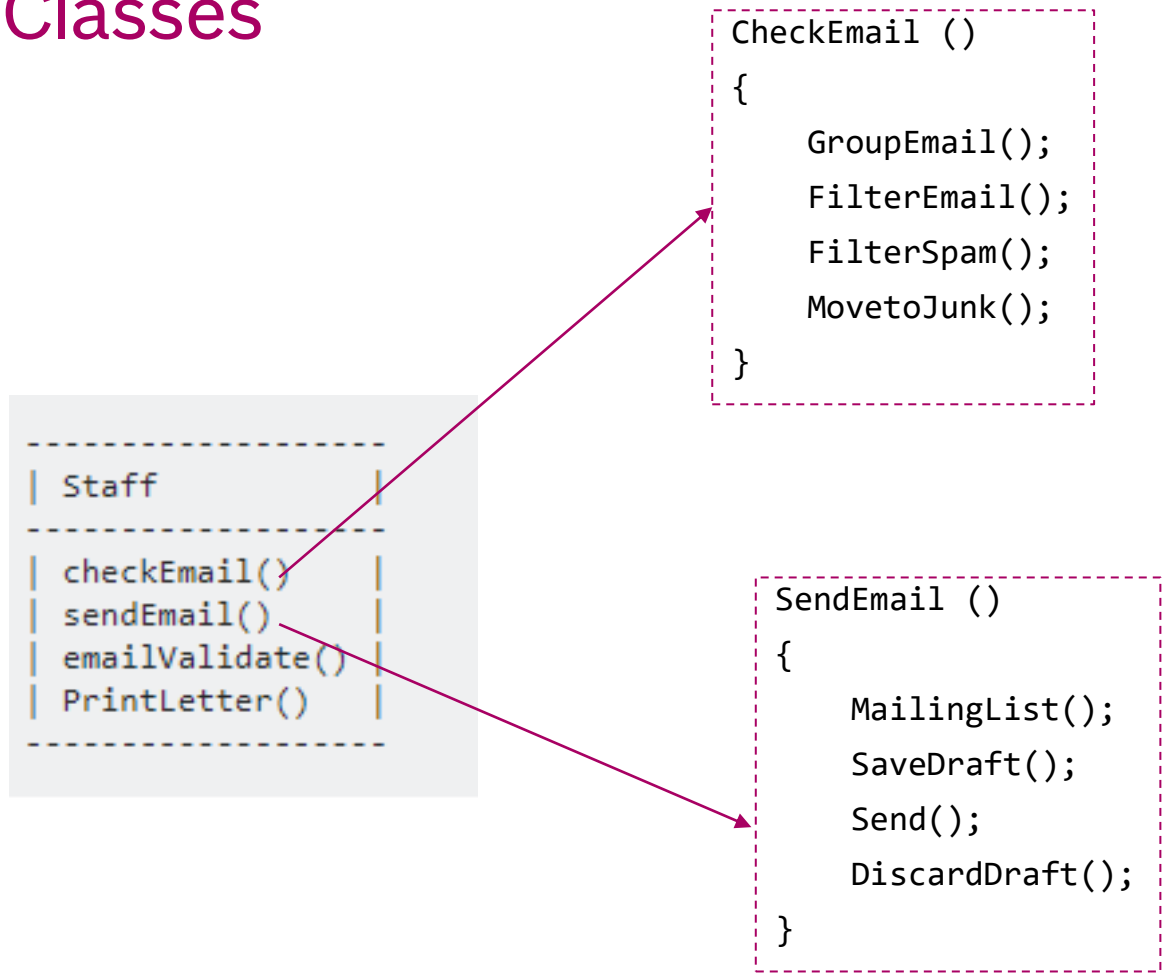
Clean Code
Classes

Cohesion and Coupling

- **Cohesion-** Indication of Relationship within a module
- **Coupling** – Indication of Relationship between modules



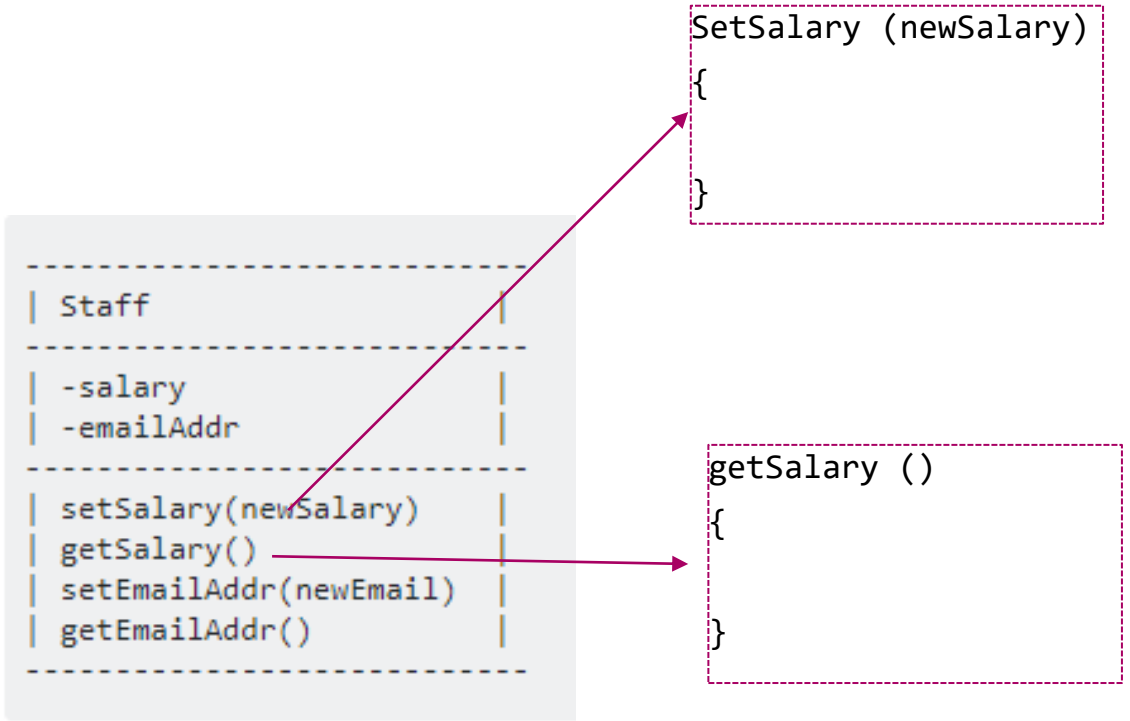
Clean Code
Classes



This class does a great variety of actions

Low Cohesion

Clean Code
Classes



This class is focused on what it should be doing

High Cohesion

Clean Code
Classes

Coupling refers to how dependent two classes/modules are towards each other.

- Loose coupled classes, changing something major in one class should not affect the other.
- Tight coupling would make it difficult to change and maintain your code.

Clean Code
Classes

```
class Car
{
   private Engine engine = new Engine( "X_COMPANY"
);
   public Car()
   {
       // do something
   }
}
```

Object of the class Car can only be created with Class Engine of “X_Company”

```
class Car
{
   private Engine engine;
   public Car (Engine engine)
   {
       this.engine = engine;
    }
}
```

Car is not dependent on an engine of "X_COMPANY" as it can be created with any types

Clean Code
Classes

Good Software Design

- ✓ Should strive for **high cohesion** with little interaction with other modules of the system.

- ✓ Should strive for **loose coupling** i.e. dependency between modules should be less

ERROR HANDLING

Clean Code
Error Handling

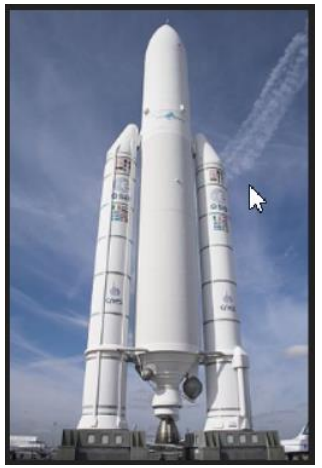
Exception is..

Abnormal or exceptional conditions requiring special processing – often changing the normal flow of program execution

Handling requires..

Specialized programming language constructs or computer hardware mechanisms.

Clean Code
Error Handling



Ariane 5 rocket launch failure in 1996

Issue

- Navigation system failure

Root cause

- Reused Initial reference platform SW from Ariane 4.
- Horizontal acceleration calculation caused a data conversion from a **64-bit floating point number** to a **16-bit signed integer** value to **overflow**.
- Error Handling was suppressed for performance reasons.

Impact

Start! **37 seconds** of flight. **BOOM!** 10 years and 350 million $ are turning into dust.

Clean Code
Error Handling

Error Handling is a Must and Good but when it is **not clean** ??

- **When we end up with code where only error handling can be seen.**
- **Impossible for us to find the details about the real functionality.**

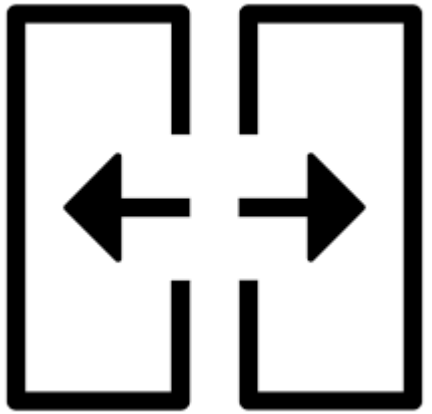
Clean Code
Error Handling

```
function(void)
{
    if(Error_1)
    {
        /* Handle Error1*/
     }
    if(Error_2)
    {
        /* Handle Error 2*/
     }
    else
       /* Do actual function*/
}
```

Clean Code
Error Handling

ErrorHandler.c

```
typedef enum
{
  ERROR_1 = 0,
  ERROR_2 = 1,
  E_OK= 5
}ErrorIdTypeDef;
extern ErrorIdTypeDef ErrorCode;
Error1_Handler()
{
    /* Handle Error 1 */
}
Error2_Handler()
{
    /* Handle Error 2 */
}
```



Function.c

```
ErrorIdTypeDef  function(void)
{
  ErrorIdTypeDef retVal = E_OK;
  if(Error_1)
  {
     Error1_Handler();
     return ERROR_1;
  }
  if(Error_2)
  {
      Error2_Handler();
      return ERROR_2;
  }
  if (E_OK)
  {
     /* Do actual function*/
  }
  return retVal;
}
```

Clean Code
Error Handling

```
SearchResult someResult = searchForStuff();
if ( someResult == null )
{
    // do something
}
```

```
try
{
    SearchResult someResult =
searchForStuff();
}
catch ( ResultNotFoundException rnfe )
{
    // do something
}
```

Clean Code

Don’t Pass Null

```
void getarray(int arr[ ])
{
    printf("Elements of array are : ");
    for(int i=0;i<5;i++)
    {
        printf("%d ", arr[i]);
    }
}

int main()
{
   int arr[5]={45,67,34,78,90};
   getarray(arr);
   return 0;
}
```

getarray(Null);

getarray(arr1);

Clean Code
Don’t Return Null

```
int *getarray(int *a)
{

    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &a[i]);
    }
    return a;
}

int main()
{
  int *n;
  int a[5];
  n=getarray(a);
  printf("\nElements of array are :");
  for(int i=0;i<5;i++)
    {
        printf("%d", n[i]);
    }
    return 0;
}
```

return Null;

return a;

Clean Code
Error Handling

- **Clean and Robust.**
- **Error Handling should be separate from main Logic.**
- **Treat them independently which provides maintainability.**

Don’t pass or return NULL!!!

BOUNDARIES

Clean Code
Boundaries

Defining clean boundaries

```
if (stEventCounter_u8 > EventThreshold_C)
{
    ActivateNextEvent();
    stEventCounter_u8 = 0;
}
else
{
    stEventCounter_u8++;
}
```

Both "*stEventCounter_u8*" and "*EventThreshold_C*" are 8-bit data.

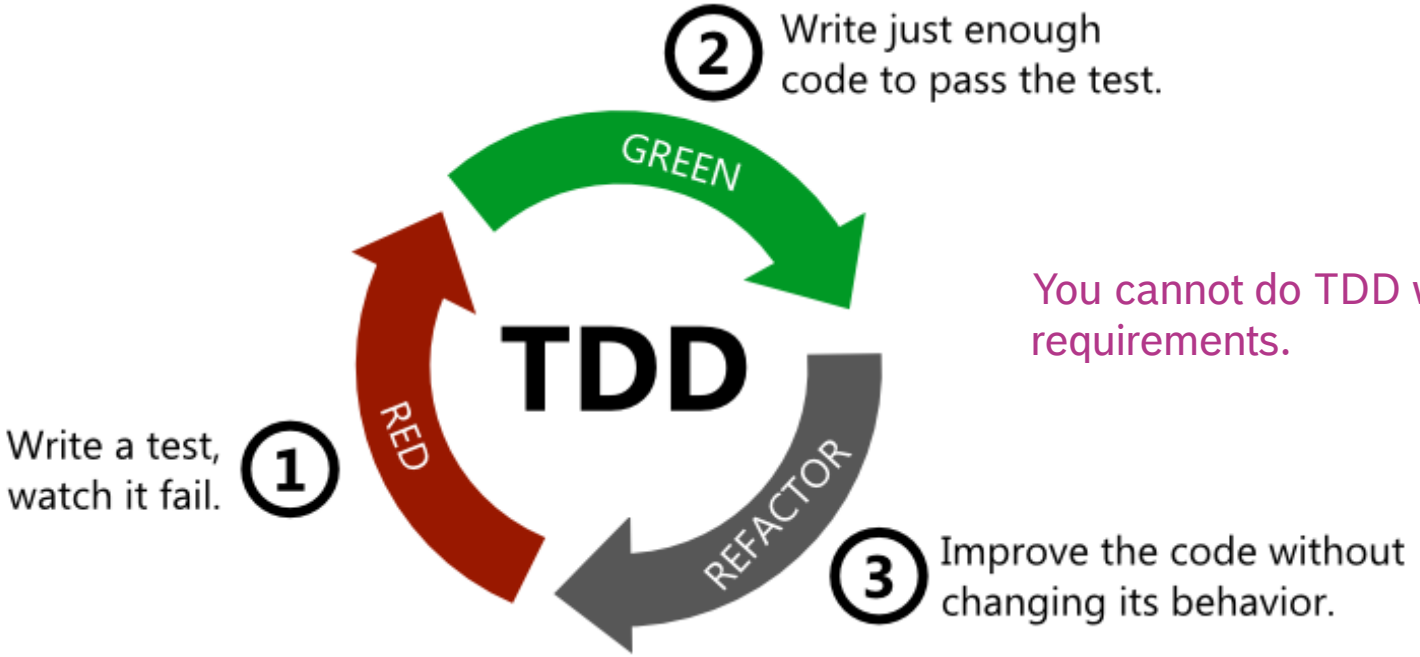
What happens if *EventThreshold_C* changes to 255??

Clean Code
Boundaries

- Keep boundaries **Clean** and **Separated**.

UNIT TESTS

Clean Code
Test Driven Development



You cannot do TDD when you don’t understand the requirements.

Clean Code
Unit Tests

Why TDD??

- Easy to validate your code because you have made tests for all of it.
- Your tests describes how your code works.
- No fear to change code.

Clean Code
Unit Tests

Rules for Clean Tests:

- ✓ **F**ast
- ✓ **I**ndependent
- ✓ **R**epeatable
- ✓ **S**elf Validating
- ✓ **T**imely