Refactoring to Bind It All Together

Zoran Horvat
OWNER AT CODING HELMET CONSULTANCY
@zoranh75 codinghelmet.com

```
int GetControlDigit(long number)
{
    int sum = 0;
    bool isOddPos = true;

    while (number > 0)
    {
        int digit = (int)(number%10);
        if (isOddPos)
            sum += 3 * digit;
        else
            sum += digit;
        number /= 10;
        isOddPos = !isOddPos;
    }

    int modulo = sum%7;
    return modulo;
}

int digit = GetControlDigit(12345);
```

- **Returns control digit for a document number**
  Control digit is used to detect typing omissions
- **Loop iterates through digits**
- **Modulo 10 returns the last digit**
- **Odd position digits multiplied by 3**
  Pondered digits summed up
- **Least significant digit removed**
- **Flag flipped in every iteration**
- **Reduce sum to a single digit**
- **Invoke the method to get the control digit value**

```
int GetControlDigit(long number)
{
    int sum = 0;
    bool isOddPos = true;

    while (number > 0)
    {
        int digit = (int)(number%10);
        if (isOddPos)
            sum += 3 * digit;
        else
            sum += digit;
        number /= 10;
        isOddPos = !isOddPos;
    }

    int modulo = sum%7;
    return modulo;
}
```

- ✓ **Short and simple**
- ✗ **It won't remain simple for long**
  Real algorithms are more complex
- ✗ **No correlation between user requirements and implementation**

  Multiply every other digit by 3
  Sum the digits up
  Take modulo 7

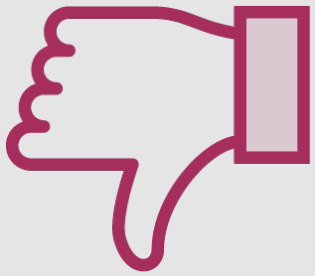
  vs.

  Loop while greater than zero
  Take modulo 10
  Multiply by 3 if at odd position
  Flip the position indicator
  Add to the sum
  Take modulo 7

```
int GetControlDigit(long number)
{
    int sum = 0;
    bool isOddPos = true;

    while (number > 0)
    {
        int digit = (int)(number%10);
        if (isOddPos)
            sum += 3 * digit;
        else
            sum += digit;
        number /= 10;
        isOddPos = !isOddPos;
    }

    int modulo = sum%7;
    return modulo;
}
```



- **The problem of cognitive mapping**
  - Often causes defects
  - It appears when implementation differs from requirements in natural language
- **The problem of lacking flexibility**
  - What if requirements have changed?
  - Mapping between requirements and implementation becomes the obstacle
  - Requirements are changing around axes defined by rules of the business
  - Implementation changes around axes defined by the programming language
  - E.g. reading digits left-to-right
  - Or change algorithm parameters?
  - Or make parameters time-dependent?

Cognitive Mapping

requirements

requirements

s causing mismatches here

The Flexibility Problem

Implementation is only flexible around axes defined by the programming language

Requirements are flexible around logical axes defined by the business

General-purpose libraries help decouple implementation from programming language details

```
int GetControlDigit(long number)
{
    int sum = 0;
    bool isOddPos = true;

    while (number > 0)
    {
        int digit = (int)(number%10);
        if (isOddPos)
            sum += 3 * digit;
        else
            sum += digit;
        number /= 10;
        isOddPos = !isOddPos;
    }

    int modulo = sum%7;
    return modulo;
}
```

In the remainder of this module:

Separate infrastructure from domain logic

Turn domain elements into objects

Summary

Separation between domain logic and infrastructure

- A good preparatory step to introducing proper objects
- Objects support future requirements
- But objects require separated domain logic

General advice on OO design

- Separate domain from infrastructure
- Then turn domain into objects

Make this your daily routine

Course Summary

Removing branching instructions

- Replace entire instruction with an object reference
- Each concrete object represents one branching outcome

Removing loops

- Make sure to operate on a sequence
- Supply transformation function and let the sequence loop through itself

Course Summary

Turning sequences into collections

- Don’t let the caller iterate through the data structure
- Let the data structure do that instead

Generalizing and reusing algorithms

- Turn parts of the algorithm into replaceable strategies
- Don’t write the same algorithm twice

Course Summary

Using value objects

- Build value-typed semantic into reference types
- This leads to reduced complexity and reduced number of defects

Avoiding use of null references

- Null reference in object-oriented code means we don't have an object
- Apply Null Object and Special Case patterns to provide replacement
- Use `Option<T>` type to represent potentially missing objects

Course Summary

Removing multiway branching

- Organize rules into a map
- Map current state into a single action

Removing nested branching instructions

- Turn them into a chain of rules
- First rule that is applicable handles the request

Separating domain logic from infrastructural operations

Course Summary

Object-oriented design

vs.

Design patterns

vs.

Refactoring techniques

Course Summary

Ultimate goals of software design

- Bring operations close to data
  - i.e. base the design on objects
- Bring polymorphism to operations
  - i.e. use virtual functions

Benefits of proper object-oriented design

- Increased flexibility and extensibility