Increasing Flexibility by Avoiding Switch Statements

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 codinghelmet.com

Multiway Branching

Criterion is not Boolean

That is what makes it different from two-way branching (`if` instruction)

There are multiple outcomes

One outcome is chosen at any given time

Traditional Multiway Branching

Computed jump (switch instruction)

There is an array of operations

Branching condition turned into an index

Operation at that index is executed

Nested if-then-elses

Test one condition and execute operation if satisfied

Otherwise, step to the nested if-then-else and repeat

Terminate with unconditional operation

Comparison of Methods

Computed jump

Must have a method to compute the jump target

Enforcing this method may sometimes be cumbersome

Nested branching

Applicable when there is prioritization between decisions

Branch on highest priority decision first

If that is not satisfied, try the next one etc.

The Case Against Enumerations

Avoid using `enum`

That makes implementation rigid

Code with **`enum`**s is hard to maintain

Consequences

What if new value has to be added?

We have add code to all using places

Goals of OO design

Added requirement means add a class

Do not change any existing classes

Grow system through composition

The Case Against `switch` Instruction

Its value in the past

Compiler comes with integer transformation

Condition on the input transformed to index

Jump runs in $O(1)$ time

Its value today

It only supports trivial values

We want to work with objects

`switch` instruction not applicable

The Case Against `switch` Instruction

Retrofitting code to support switch

Bend the class to support switch

Added or modified case?

Modify this class

Root cause of the problems

Class with switch has to change

But it has nothing to do with change in requirements

Maintenance takes a long time and leads to bugs

The Case Against `switch` Instruction

Mixing representation into domain

Representation breaks into higher level code

Changing representation becomes very hard

Lots of code has to change when it changes

Mapping representation to operations

This mapping is hard-coded

All actions known in advance

Renders polymorphic execution impossible

Value of Dynamic Dispatch

We might not know the call target

Decision might be based on dynamic conditions

```
void ClaimWarranty(action)
{
  Action<Action> target =
    map[status];

  target.Invoke(action);
}
```

- **Call to `map` to discover the target**
  This is the first dispatch
- **Polymorphic call on `target`**
  This is the second dispatch
- **Possible call to `action` is again polymorphic**
  That makes it a triple dispatch

Value Added by Dynamic Dispatch

Class doesn't know outcomes of calls it makes

It doesn't have to change when the target is changed

SoldArticle contains no logic to treat warranties

Keeps warranties, state and rules

Class doesn't change when rules change

Changing class text invites regression

Prior feature might stop working

We prefer adding new types instead

Summary

Solution based on `switch` instruction

- Exposes physical representation
- Fixes the branching logic

Summary

Object-oriented substitute to `switch`

- Encapsulate state in a separate class
  - Don’t let consumers see the representation
- Devise a mapping between state object and operations
  - Simple implementation based on a dictionary
- This makes entire mapping logic substitutable
  - Replace the map from the outside

Next module –

Dealing with nested branching