Technology Group

Complexity

Simplified

Functional Programming

Authored & Presented By : **Amit Mulay**

Technical Evangelist

This presentation is the intellectual property of Cybage Software Pvt. Ltd. and is meant for the usage of the intended Cybage employee/s for training purpose only.
This should not be used for any other purpose or reproduced in any other form without written permission and consent of the concerned authorities.

Requirement

System to schedule tasks and meetings and we want to have several ways to specify the calendar

- For event that happen only once <-> (Datetime)
- Events that occur repeatedly <-> (Datetime, timeinterval)
- Events that don’t have time specified yet <-> ?

Object oriented way

Abstract Base class -> Schedule
(GetNextOccurance() : DateTime)

Child classes ->

Never

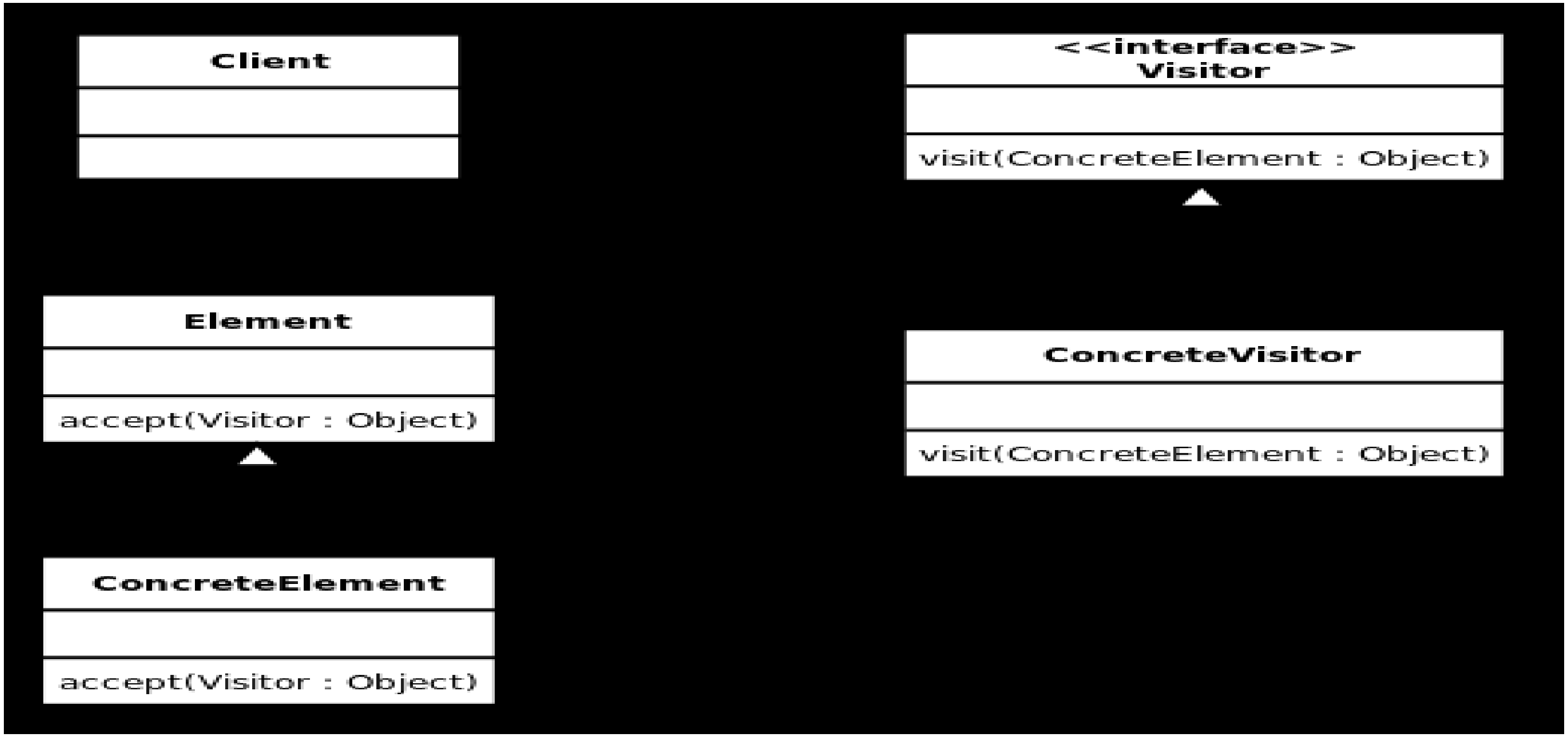
Once(eventdate : DateTime)

Recurring(StartDate : DateTime
TimeInterval : TimeSpan)

- Adding new type of schedule is easy.
- Adding new operation is difficult. (like GetPreviousOccurance() or GetOccuranceNumber())
- Code gets distributed in different files.
- Visitor pattern is used

Visitor pattern

The Gang of Four defines the Visitor as: "***Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.***"



FP way #1

```
type Schedule =
            | Never
            | Once of DateTime
            | Repeatedly of DateTime * TimeSpan
```

- Adding new operation is very easy. (like GetPreviousOccurance() or GetOccuranceNumber())
- Adding new type of schedule is difficult.
- We use pattern matching and hence all code for given operation is at same place.

FP way #2 - OOPS

```
type NextOccuranceF<'a> = 'a -> DateTime
type Schedule<'a> = 'a * NextOccuranceF<'a>
let GetNextOccurance<'a> (obj : Schedule<'a>) =
    (snd obj) (fst obj)

// first subclass
type Never = | Never
let NeverCons  =
    let neverf : NextOccuranceF<Never> = fun _ -> DateTime.Max
    fun () -> (Never, neverf) : Schedule<Never>

// second subclass
type Once = DateTime
let SonceCons  =
    let oncef : NextOccuranceF<Once> = fun d -> d.AddDays(1)
    fun (date : DateTime) -> (date, oncef) : Schedule<Once>
```

FP way #3 – WIN WIN SOLN

```
type Schedule<‘a,’f> =
            | Never
            | Once of DateTime
            | Repeatedly of DateTime * TimeSpan
            | FUTURETYPE of ‘a * ‘f

let GetNextOcurrance sch = match sch with
                                    | Never
                                    ..
                                    |  FUTURETYPE(a,f) -> f a
let r = GetNextOcurrance (Once(d))
let r1 = GetNextOcurrance (Never)

type k = {d : datetime; int a; int b}
let GetNextOcurranceK k = …
let r1 = GetNextOcurrance (FUTURETYPE({d,a,b},
GetNextOcurranceK ))
```

Generic functions

Function should perform operation on value obtained but since code needs to be generic, we don’t want to restrict type on value too much.

OOP’s way

- Interface
- actual value will have operations defined for given interface

```
interface ITestAndFormat
{
        bool Test();
        string Format();
}
void CondPrint(ITestAndFormat tf) {
        if (tf.Test())
                Console.WriteLine(tf.Format());

}
```

Generic functions

Function should perform operation on value obtained but since code needs to be generic, we don’t want to restrict type on value too much.

FP way

- Using type parameter (Generics)

```
void CondPrint<T>(T value, Func<T, bool> test, Func<T, string> format) {
        if (test(value))
                Console.WriteLine(format(value));
}
```

F#

```
let condPrint value test format =
        if (test(value)) then
                printfn "%s" (format(value))
        else ()
val condPrint : 'a -> ('a -> bool) -> ('a -> string) -> ()
```

High order functions - Tuple

- Map functions on Tuple
- You need two!
- MapFirst
- MapSecond

```
let mapFirst f (a, b) = f a, b
 let mapSecond f (a, b) = a, f b
(‘a -> ‘b) -> ‘a  * ‘c -> ‘b * ‘c
(‘a -> ‘b) -> ‘c * ‘a -> ‘c * ‘b
```

Map

- Structure remains unchanged
- Functions acts on component of structure

```
(‘a -> ‘b) -> List<‘a> -> List<‘b>
(‘a -> ‘b) -> Wrapper<‘a> -> Wrapper<‘b>
```

High order functions - Schedule

```
type Schedule =
        | Never
        | Once of DateTime
        | Repeatedly of DateTime * TimeSpan

let mapSchedule rescheduleFunc schedule =
        match schedule with
        | Never -> Never
        | Once(eventDate) > Once(rescheduleFunc(eventDate))
        | Repeatedly(startDate, interval) >
                Repeatedly(rescheduleFunc(startDate), interval)

val mapSchedule : (DateTime -> DateTime) -> Schedule
                        -> Schedule
```

High order functions - Schedule

```
let mapSchedule rescheduleFunc schedule =
          match schedule with
          | Never -> Never
          | Once(eventDate) > Once(rescheduleFunc(eventDate))
          | Repeatedly(startDate, interval) >
                    Repeatedly(rescheduleFunc(startDate), interval)
```

Reschedule by 7 days:

```
  schedule |> mapSchedule (fun x -> x.AddDays(7))
```

High order functions - Options

Req : Given ConsoleIntRead() which gives Some(n) if user enters valid number. Else gives None.

Get 2 inputs from user and return Some(addition of numbers) if both inputs are number else return None.

```
let readAndAdd1() =
        match (readInput()) with
        | None   -> None
        | Some(n) -> match (readInput()) with
                     | None   -> None
                     | Some(m) ->      Some(n + m)
```

High order functions - Options

```
let readAndAdd1() =
            match (readInput()) with
            | None   -> None
            | Some(n) -> match (readInput()) with
                             | None   -> None
                             | Some(m) ->      Some(n + m)
```

Lets check map signature for options

(‘a -> ‘b) -> ‘a option -> ‘b option

```
let optionmap f a =
            match a with
            | None -> None
            | Some a -> Some(f a)
```

High order functions - Options

```
let readAndAdd1() =
            match (readInput()) with
            | None   -> None
            | Some(n) -> match (readInput()) with
                             | None   -> None
                             | Some(m) ->      Some(n + m)
let optionmap f a =
            match a with
            | None -> None
            | Some a -> Some(f a)

let readAndAdd2() =
            match (readInput()) with
            | None       -> None
            | Some(first) > readInput()
                              |>  optionmap  (fun second -> first + second)
```

High order functions - Options

```
let readAndAdd2() =
            match (readInput()) with
            | None        -> None
            | Some(first) > readInput()
                                |> Option.map (fun second -> first + second)
```

With map we eliminated inner match!

Can we eliminate outer match?

(‘a -> ‘b option) -> ‘a option -> ‘b option

(‘a -> Wrapper of ‘b) -> Wrapper of ‘a -> Wrapper of ‘b

Well this is signature for High order function known as bind

High order functions - Options

(‘a -> ‘b option) -> ‘a option -> ‘b option

(‘a -> Wrapper of ‘b) -> Wrapper of ‘a -> Wrapper of ‘b

Well this is signature for High order function known as bind

```
let optionbind f a =
        match a with
        | None -> None
        | Some(a) -> f a
```

High order functions - Options

```
let readAndAdd2() =
          match (readInput()) with
          | None      -> None
          | Some(first) > readInput()
                          |> Option.map (fun second -> first + second)
```

Using optionbind

```
let readAndAdd3() =
          readInput() |> optionbind(fun first ->
                    readInput()
                    |> Option.map (fun second -> first + second)
          )
```

High order functions

```
// map operation
val mapFirst     : ('a -> 'b) -> 'a * 'c   -> 'b * 'c
val List.map     : ('a -> 'b) -> 'a list   -> 'b list
val Option.map   : ('a -> 'b) -> 'a option -> 'b option

// filter operation
val List.filter   : ('a -> bool) -> 'a list   -> 'a list
val Option.filter : ('a -> bool) -> 'a option -> 'a option

// fold operation
val List.fold    : ('a -> 'b -> 'a) -> 'a -> 'b list   -> 'a
val Option.fold  : ('a -> 'b -> 'a) -> 'a -> 'b option -> 'a

Option.bind :  ('a -> 'b option) -> 'a option -> 'b option
List.bind   :  ('a -> 'b list)   -> 'a list   -> 'b list  // Referred as List.collect
```

Any Questions?

Thank you!