Designing function families and bundles with F# lambdas and behaviors parameterization

Alain Lompo
(@alainlompo)

About me

- **I am a software developer at Senacor**
- **Currently building e-banking solutions using**
  - Java (JEE/Spring boot) in the backend
  - Angular/React on the frontend
- **Worked a lot with .Net technologies**
  - MCSD, MCTS .Net and MCT
- **Available at: @alainlompo**

Overview

- **Motivation**
- **Functional programming to the rescue**
- **Behaviors parameterization**
- **Designingf amilies of functions**
- **Applications and Demos(all along)**
- **Hands on labs (all along)**
- **Wrap - up**

User requirements are sinking sand…

MOTIVATION

User requirements are sinking sand

- They always seem good at first
- But they always change later
- It happens generally during the course of implementation
- Sometimes even later

Blaming it on the methodology

- It does not matter which methodology isused, the problem will still be there
- We simply find out about it earlier or later depending on the methodology
- With agile methods we find about it earlier

Re-touching the code is costly

- **Software maintenance costs generally more than its initial implementation**
- **Behavior's parameterization can help reduce these costs significantly**

Making developers unhappy is costly

- **Developers generally feel happy when**
  - The task has been successfully implemented, tested and set as Ready for PROD (green check mark)
  - The task was not specified clearly enough so it is set in a « REQUIRES CLARIFICATIONS » or « BLOCKED » status
  - In both case they can move forward with new (and exciting) tasks and live happily ever after.

To the rescue, behaviors parameterization and...

FUNCTIONAL PROGRAMMING

Core advantages of functional programming

- **Finish on time and meet deadlines**
  - Reduces time to market for yourprojects
- **Write correct code**
  - Avoid mutability and state handling issues
  - Avoid null values handlind and NPEs
  - Avoid external iterations
  - Express intent and « the what » rather than « the how »

Core advantages of functional programming

- **Handle complexity**
  - …with simpler code
  - Leads to using more advanced algorithms and providing better functionalities
- **Efficient and scalable code**
  - Easier to parallelize code
  - Better abstractions for writing reactive code
  - Better abstractions for writing asynchronous code

Illustrating OOP ceremonies

- **The financial service offers two functions, one for computing interest and the other for increased amount**

```
namespace Financeslib
{
   public class FinancialService
   {
      public double computeInterest(double interestRate, double amount)
      {
         return amount * interestRate;
      }

      public double computeNewAmount(double interestRate, double amount)
      {
         return amount + computeInterest(interestRate, amount);
      }
   }
}
```

Illustrating OOP ceremonies

- **Classe instanciation and method call through it are ceremonies that OOP needs but not functional programming**

```
FinancialService financialService = new FinancialService();
 double interest = financialService.computeInterest(interestRate, initialAmount);
 double newAmount = financialService.computeNewAmount(interestRate, initialAmount);
```

Parameterizing behaviors in OOP

- **In case the computing of interest is not trivial, it can be passed as a parameterized behavior. First we define a corresponding interface**

```
namespace Financeslib
{
    public interface IFinanciable
    {
        double computeInterest(double interestRate, double amount);
    }
}
```

Parameterizing behaviors in OOP

- **The new version of the computeNewAmount method receives a parameter of type Ifinanciable and so the real behavior is now only at run-time**

```
namespace Financeslib
{
   public class EnhancedFinancialService
   {
      public double computeNewAmount(double interestRate, double amount,
IFinanciable     interestCalculator)
      {
         return amount + interestCalculator.computeInterest(interestRate, amount);
      }
   }
}
```

Compositionality

- **F# allows us to easily create new functions by composing them from existing ones**

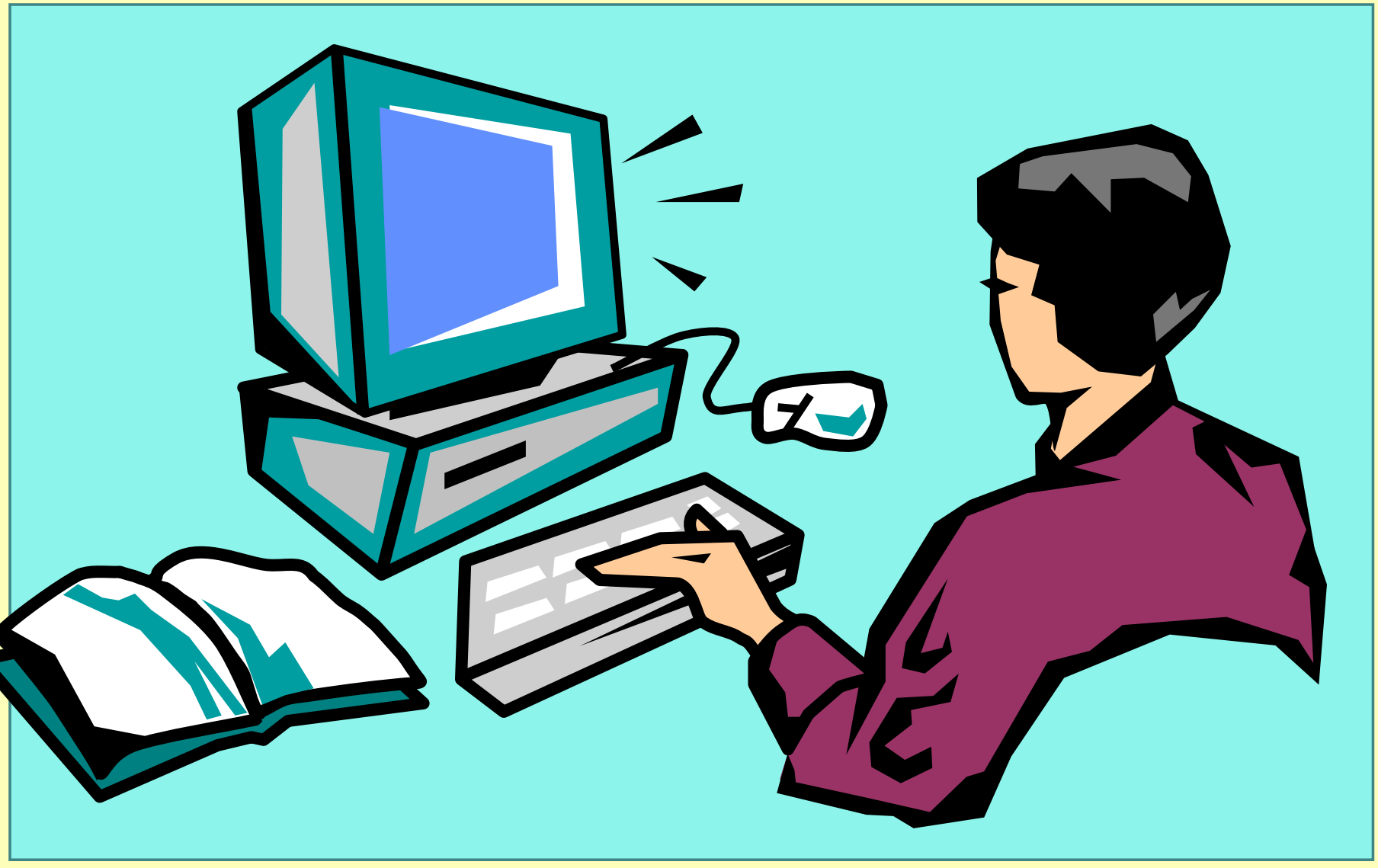
```
Let squareOddValuesAndAddOneComposition       =
       List.filter isOdd >> List.map (square >> addOne)


let fog = fun f g ->
fun n -> f (g n)

llet expXSquare = fog (fun x -> exp x) (fun x -> x*x)
```

Non-blocking code with F#

- **Writing non-blocking IO operations is simplified using F# asynchronous workflows**

```
let op =
async  {

let   req = HttpWebRequest.Create(https://www.allthetests.com)
let   resp = req.AsyncGetResponse()
let   stream = resp.GetResponseStream()
let   reader = new StreamReader(stream)
let   html = reader.AsyncReadToEnd()
Console.WriteLine(html)
}

Async.Run(op)
```

Lab 0: F# core syntax refresher



Introducing discriminated union

- **With this we are able to create new types as exclusive union of all possible combinations of other specified types**

```
typeBoolOrInt =
    | B of Boolean
    | I of int64
```

- **We can also use it to mix custom types into some type of record**

```
typeCustomerFile =
    | CustomerInfos of Customer
    | Accounts of List<Account>
    | Transactions of List<Transaction>
    | AktiveSession of CustomerSession
```

Value declaration and scope

- **The let keyword is often used to declare immutable variables. Let’s illustrate scope with an example**

```
letnum = 42 #1
printfn "%d" num
letmsg = "Answer: " + (num.ToString())
printfn "%s" msg
```

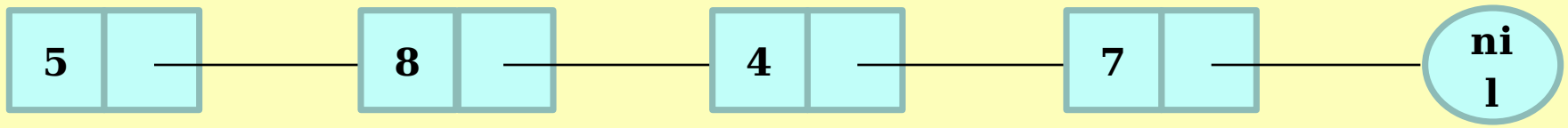
Introducing tuples

- In F#, tuples are very simple types that can gather together values of different types
- The type of a tuple is a combination of the types of its values. The special symbol (*) is used to combine the values

```
let tuple1  = ("Hello world!", 42)
val tuple1 : string * int
```

Functional lists and recursion

- **Functional lists are recursively defined: a list is either composed with an element and a list or is an empty.**



Lab 1: A glimpse into the power of functional programming with F#

Passing a function as an argument in F#

- **In F#, a function is just another type. The type of a function is a combination of its return type and the types of its arguments.**

```
> let rec aggregateList (f:int -> int -> int) init list = #1
match list with
| [] -> init #2
| hd::tl ->
let rem = aggregateList f inittl #3
f rem hd #3
;;
valaggregateList : (int -> int -> int) -> int -> int list -> int #4
> let add a b = a + b #A
let mul a b = a * b #A
;;
val add : int -> int -> int #B
valmul : int -> int -> int #B
> aggregateList add 0 [ 1 .. 5 ];; #C
val it : int = 15 #C
> aggregateListmul 1 [ 1 .. 5 ];; #C
val it : int = 120 #C
```

Signatures of functions in F#

Function taking two numbers
And returning a number

List to be processed

(int -> int -> int) -> int -> int list -> int

Initial value of type int

Returns the aggregated value

Further on discriminated unions

- **Provides support for values that can be one of many named cases**
  - Possibly each with different value and type
  - Used recursively to represent tree data

```
type Schedule =
| Never #A
| Once of DateTime #B
| Repeatedly of DateTime * TimeSpan #C
```

Generic option type in F#

- **The type is declared with a type parameter**
  - It is used as the type of the value stored in the Some alternative

```
type Option<'T> =
| Some of 'T
| None
```

Type inference in F#

- **The type inference mechanism of F# is more sophisticated than what we usually find on other languages**

```
let num = 123 #A
let tup = (123, "Hello world") #B
let opt = Some(10) #C
let input = printfn "Calculating..." #D
if (num = 0) then None
else Some(num.ToString())
```

More on type inference in F#

- **Here are a few non – trivial (tricky) use case of type inference in F#**

```
let a = null #1
let (a:System.Random) = null #2
let a = (null:System.Random) #2
```

More on generic functions

- **An illustration to start with in C#**

```
T ReadValue<T>(Option<T> opt) { #A
T  v;
if  (opt.MatchSome(out v)) return v;
else  throw new InvalidOperationException();
}
```

Function values

- **An illustration to start with in C#**

```
var nums = new int[] {4,9,1,8,6};
var evens = new List<int>(); #A
foreach(var n in nums) #A
if (n%2 == 0) #B
evens.Add(n); #A
return evens;
```

Function as an argument (C# illustration)

- In this illustration the method takes a function as an argument
- The calling code defines the concrete function using a lambda expression

```
int twice (int n; Func<int, int> f) {
   return  f(f(n));
}

var r = twice(2, n => n * n); // r    16
```

Function as an argument (F# version)

- **Tuples can be used to help specify a function of multiple arguments**

```
> let add = fun (a, b) -> a + b;;
val add : int * int -> int
```

Using tuples with functions of multiple arguments

- In this illustration the method takes a function as an argument
- The calling code defines the concrete function using a lambda expression

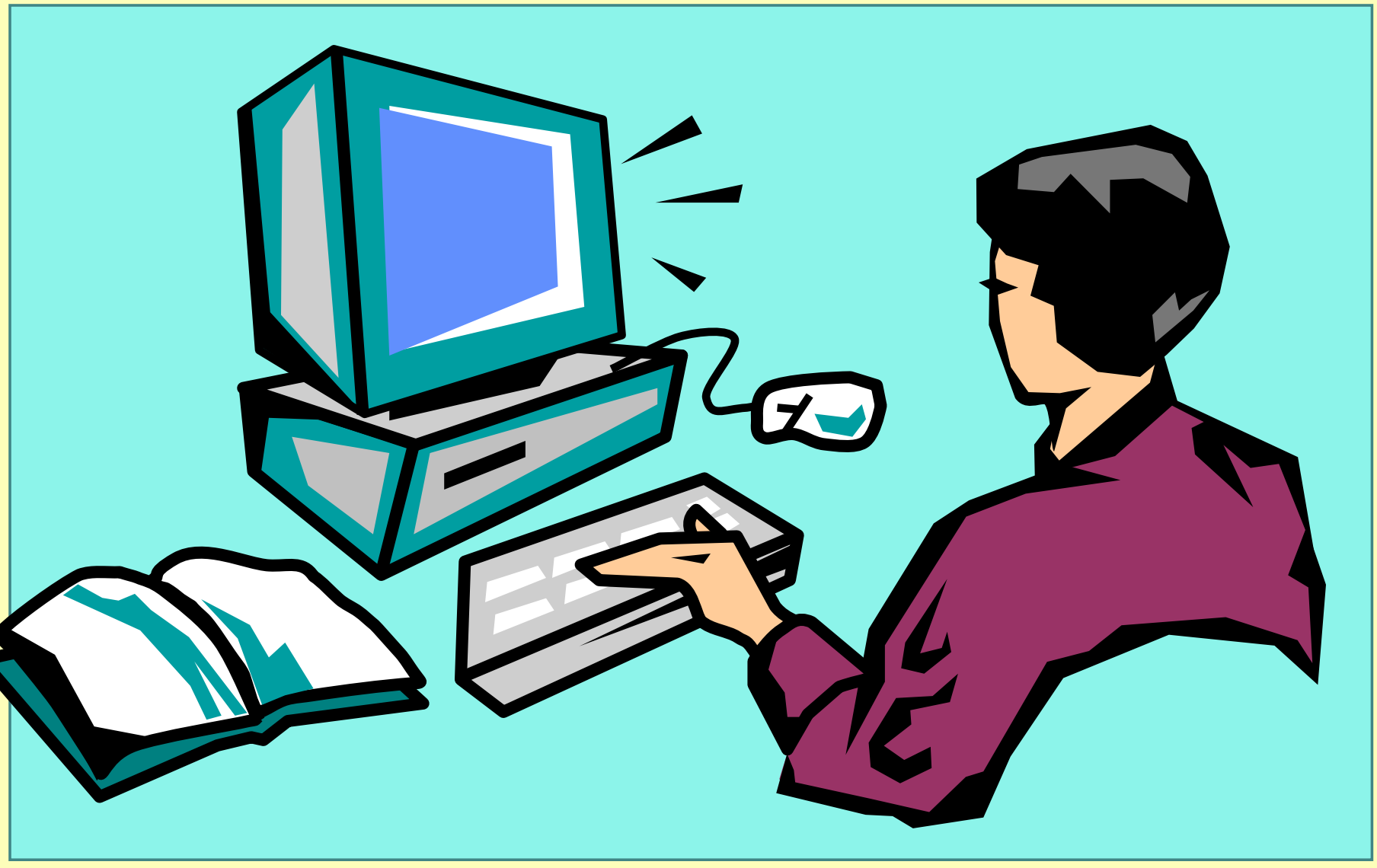
```
➢ let twice n (f: int -> int) = f(f(n))
val twice :
int -> (int -> int)
    -> int

➢ twice 2 (fun n -> n * n)
val it : int = 16
```

Using the idiomatic F# style

- **The signature remains the same as seen earlier and can read as: int -> int -> int**

```
> let add = fun a b -> a + b;;
val add : int -> int -> int
```

Lab 2: Using functions as first class citizens



Generic code through interface in OOP

- **In object oriented programming, the use of interfaces as method parameters can help attain a certain degree of genericity**

```
interface ITestAndFormat {
bool  Test();
string  Format();
}

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

Generic code through type parameters in OOP

- **The use of type parameter helps achieve even a higher degree of generality**

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

Using the map function

- **Map operations usually apply a given functions to values carried by the data type and wrap the result in the same structure**

```
> Option.map;;
val it : (('a -> 'b) -> 'a option -> 'b option) = (...)
```

Using the bind function

- **First input is read and passed to the bind operation**
  - It executes the lambda function only when the input contains a value
  - Inside the lambda, the second input is read and projected into a result value

```
let readAndAdd2() =
readInput() |> Option.bind (fun num -> #1
readInput() |> Option.map ((+) num) ) #2
```

Type inference

- **When calling a generic function we can specify the types of the arguments explicitely**

```
var dt = Option.Some<DateTime>(DateTime.Now);
dt.Map<DateTime, int>(d => d.Year);
```

- **Generally type inference allows types to be automatically deduced:**

```
dt.Map(d => d.Year)
```

Type inference for function calls in F#

- **Specific type information are rarely used in F# function calls: there might however cases where that is needed**

```
> Option.map (fun v -> v.Year) (Some(DateTime.Now));;
error FS0072: Lookup on object of indeterminate type.

> Option.map (fun (v:DateTime) -> v.Year) (Some(DateTime.Now));;
val it : int option = Some(2008)
```

Automatic generalization

- **Thanks to automatic generalization, high order functions can be implemented without needing to specify the types at all**

```
let bind func value = #1
match value with #2
| None -> None #3
| Some(a) -> func(a) #4
```

Writing generic functions

- **In F# there are many way to define generic artefacts**

```
let makeList a b = [a; b]

let function1 (x: 'a) (y: 'a) = printfn "%A %A" x y

let function2<'T> x y = printfn "%A, %A" x y
```

Implementing list functions

- **Let inspect the RemoveAllMultiples function below**

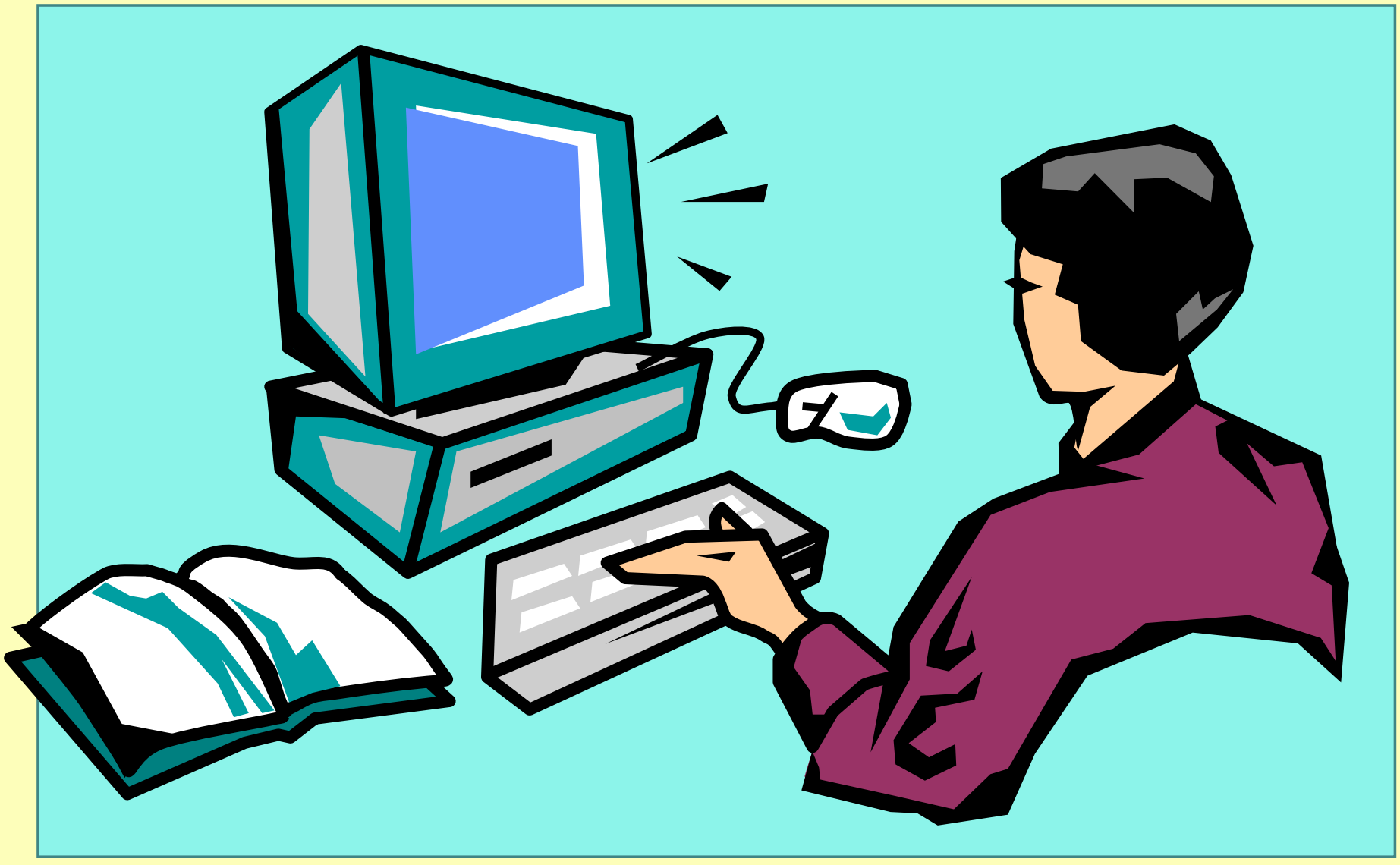
```
let IsPrimeMultipleTest n x =
   x = n || x % n <> 0

let rec RemoveAllMultipleslistnlistx =
   match listn with
   | head :: tail -> RemoveAllMultiples     tail (List.filter (IsPrimeMultipleTest head) listx)
   | [] -> listx


let GetPrimesUpTo n =
    let max = int (sqrt (float n))
RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

The bind operation for lists

- **The bind operation is an extremely important operation**

```
Option.bind : ('a -> 'b option) -> 'a option -> 'b option
List.bind : ('a -> 'b list) -> 'a list -> 'b list
```

Lab 3: Building the hyperbank core components



Designingparameterized...

BEHAVIORORIENTED APPLICATIONS

Three types of applications

- **We can easily classify software applications into three categories**
  - Data centric applications
  - Behavior centric applications
  - Applications that combine both
- **What do you think about the following?**
  - Ms Paint
  - Nginx Proxy server
  - 3D Studio Max
  - A firewall application

How do we distinguish them?

- **The difference is conceptual rather than technical**
  - We should correctly understand the kind of application we are designing
  - This will be helpful for making a correct design

Defining and using collections of behaviors

- **We will write a series of criterias (conditions to be met) for a customer to be eligible for a loan**
  - Each criteria is independent from the others
  - But it should be possible to create a macro-criteria by combining more criteria
  - Storing the criteria in a collection makes it easy to add new criteria when there is a need to do so
  - Correctly designing the application will make it possible to add there criteria dynamically (at runtime instead of compile time) and thus achieving

Representing behaviors as objects

- **Are we reverencing traditions?**
- **No, but it is useful to see how the pattern might be designed using pure OOP**

```
interface IClientTest {
bool Test(Client client); #1}

class TestYearsInJob : IClientTest { #A
    public bool Test(Client client) {
        return client.YearsInJob < 2; #2
    }
}


// Client code
List<IClientTest> loanCriterias = ... // Load (somehow) the list here


// check that all the criterias are met, otherwise no loan
```

Representing behaviors as functions (C# approach)

- The OOP world tries generally to represent a function via an interface with a single method (known as functional interface in some of the languages such as Java 8)
- The IClientTest interface fulfills the criteria of a functional interface, so the

```
Func<Client, bool> testProfessionalExperience = client => client.professionalExp < 5;
```

- This is already a significant reduction of boilerplate code

Representing behaviors as functions (...)

- **Here is a more detailed example**

```
public class Customer
   {
      public string Name { get; set; }
      public Decimal YearlySalary { get; set; }
      public intProfessionalExperience { get; set; }
      public boolSociallyFit { get; set; }
      public boolCriminalRecord { get; set; }
      public intYearlyTrainings { get; set; }

      public static List<Func<Customer, bool>> GetCriteria()
      {
         return new List<Func<Customer, bool>>
         {
customer => customer.CriminalRecord,
customer => customer.YearlySalary.CompareTo (60000)< 0,
customer => customer.SociallyFit,
customer => customer.YearlyTrainings < 2

         };
      }
   }
```

Checking the criteria

- **When considering the possibility of a loan for a client, we want to check the list of criteria.**
  - We count the tests that returned true (which means the criteria has failed, and we are facing a high risk)
  - If the count is null or 1, the program recommends the loan, otherwise there

```
public class LoanRecommender
   {
      public boolShouldOfferLoanTo(Customer customer)
{
intfailedCriteriasCount = Customer.GetCriteria()
                          .Count(criteria => criteria(customer));
         return failedCriteriasCount <= 1;
      }
   }
```

The Count function

- **Count is a high order function.**
  - Takes a predicate as argument
  - Counts the number of elements for which the predicate returns true (the criteria failed)

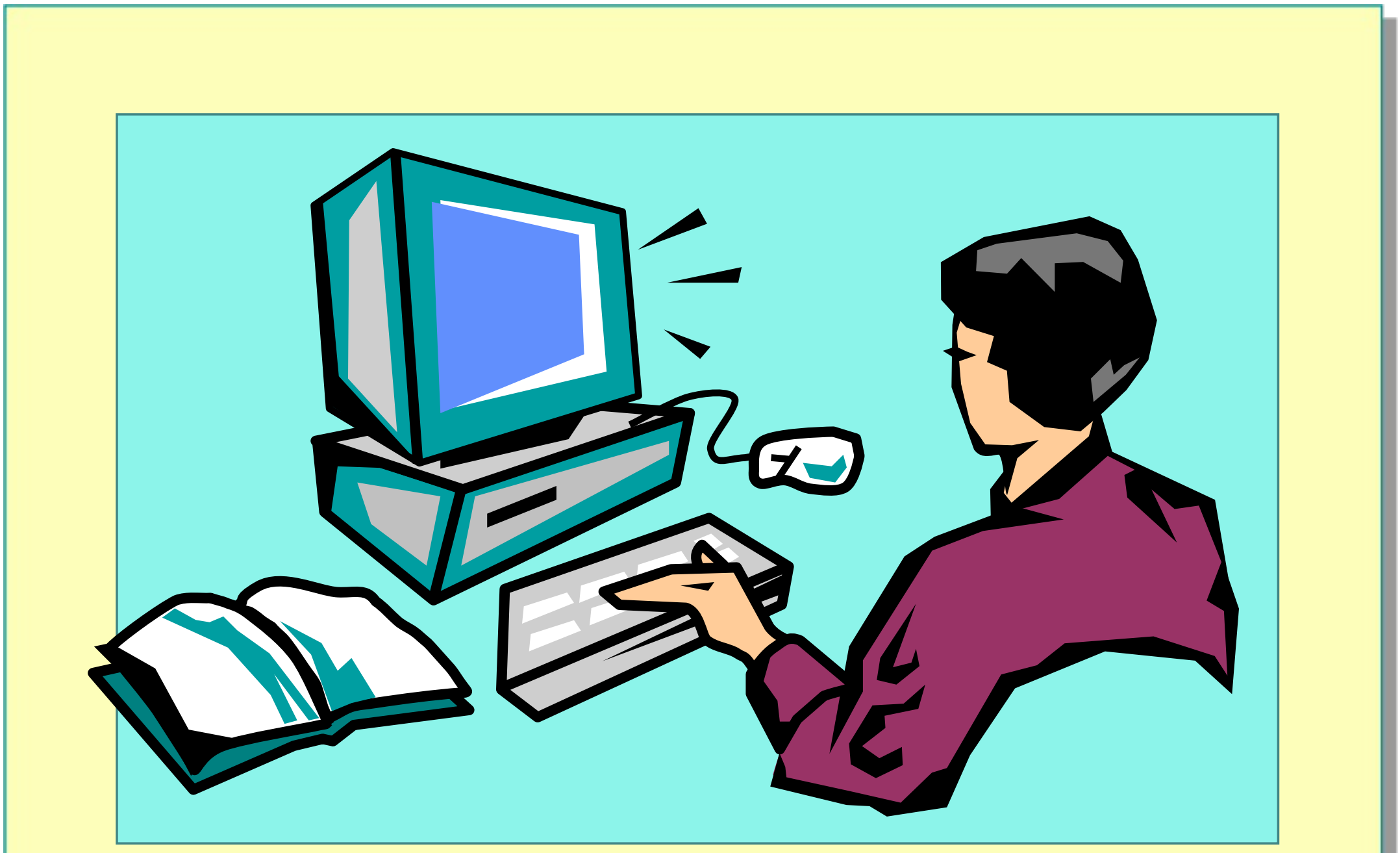
```
public class LoanRecommender
    {
        public boolShouldOfferLoanTo(Customer customer)
{
intfailedCriteriasCount = Customer.GetCriteria()
                                .Count(criteria => criteria(customer));

            return failedCriteriasCount <= 1;
        }
    }
```

Sample demo client

- **The snippet below shows a sample use case in a console application**

```
        public staticvoidassessLoanRequests()
        {
            Customer customer = getCustomer();
if (LoanRecommender.ShouldOfferLoanTo(customer))
            {
Console.WriteLine("Customer " + customer.Name + " should receive the loan");
            }
else
            {
Console.WriteLine("Customer " + customer.Name + " should be refused the loan");

            }
        }
```

Lab 4: Injecting parameterized behaviors



Defining and using lists of functions with F#

- **Let's first define our Customer type**

```
type Customer =
    {
        Name: string; YearlySalary: Decimal; ProfessionalExperience: int; SociallyFit: bool;
CriminalRecord: bool; YearlyTrainings: bool;
    };;
```

- **Then we define our criteria as a list of behaviors**

```
let loanCriteria =
    [
        (fun customer -> customer.CriminalRecord = true);
        (fun customer -> customer.YearlySalary < 60000.0M);
        (fun customer -> customer.ProfessionalExperience < 3);
        (fun customer -> customer.SociallyFit = false);
        (fun customer -> customer.YearlyTrainings < 2);
    ]
```

Assessing the criteria using List.filter (F#)

- In the C# version we used the Count method
- We could implement the equivalent in F# or use a combination of other standard functions to achieve the same

```
let assessShouldOfferLoanTo (customer) =
    let failedCriteria = loanCriteria|>List.filter(fun f -> f customer)
    let offered = failedCriteria.Length <= 1
printfn "Customer: %s\n Offer a loan: %s (unsatisfied criteria = %d)" customer.Name
        (if (offered) then "Yes" else "No") failedCriteria.Length;;
```

Sample demo client (F#)

- **The snippet below shows a sample use case executable in interactive mode**

```
let Jonathan =
    {
       Name = "Jonathan Donovan"; YearlySalary = 53500.5M; ProfessionalExperience = 5;
SociallyFit = true; CriminalRecord = false; YearlyTrainings = 3;
    };;

assessShouldOfferLoanTo(Jonathan);;
```

- **The result looks like:**

```
> val Jonathan : Customer = {Name = "Jonathan Donovan";
YearlySalary = 53500.5M;
ProfessionalExperience = 5;
SociallyFit = true;
CriminalRecord = false;
YearlyTrainings = 3;}

> Customer: Jonathan Donovan
Offer a loan: Yes (unsatisfied criteria = 1)
val it : unit = ()
```

Point free programming approach

- **When using point free style we may not have to write lambda function explicitely when calling a high order function**
  - We are working with data structure with values inside (a list or a set for example)
  - But we assign no name to the value (point)

```
[1..10] |> List.map((*) 15);;

val it : int list = [15; 30; 45; 60; 75; 90; 105; 120; 135; 150]
```

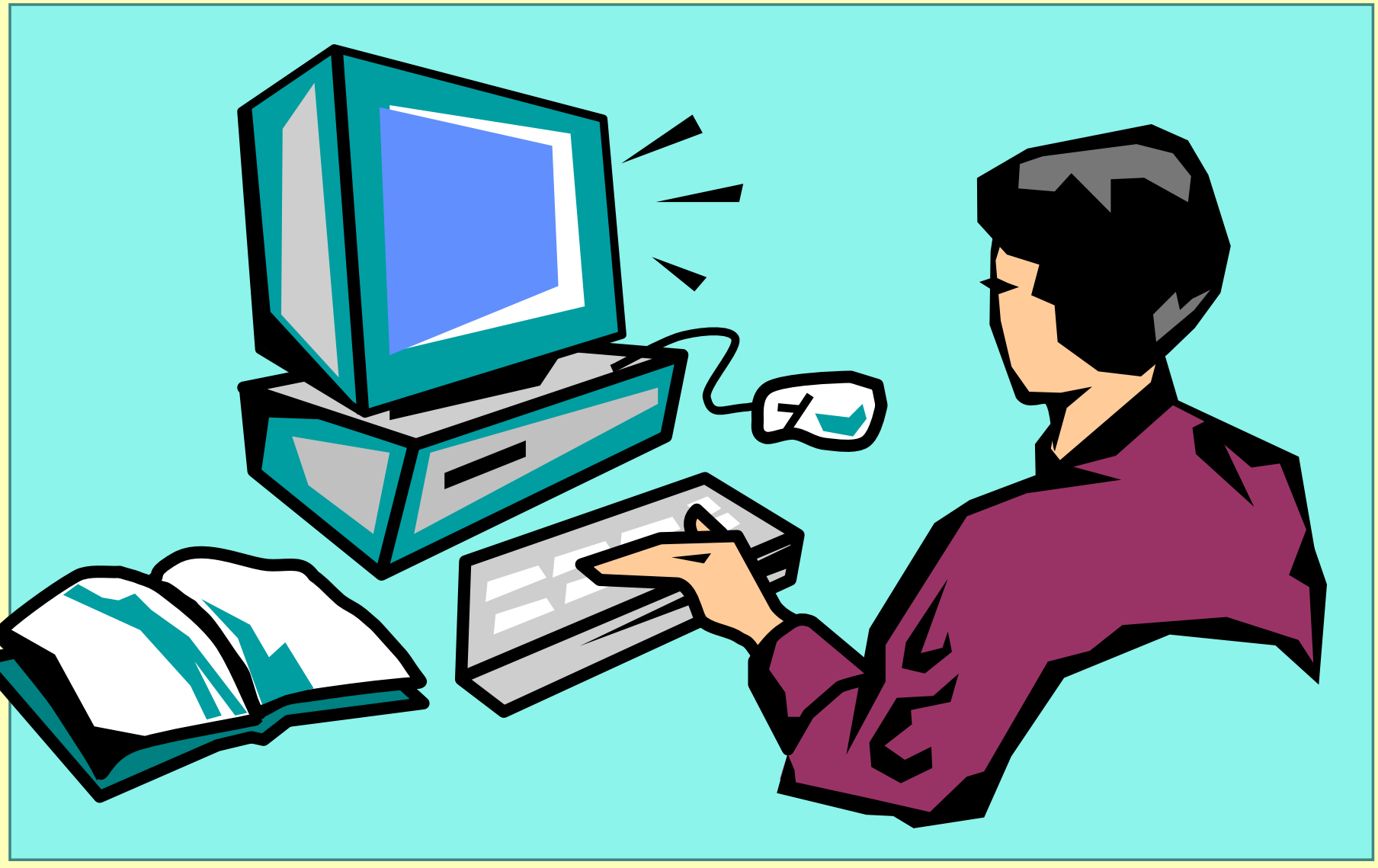
Applying the point free approach

- **Let's refactor our precedent code in point free style**
  - In criteria definition, instead of: fun **f -> f client**, we write **f -> client |> f**
  - Therefore we re-write the filter part as:

```
loanCriteria |> List.filter((|>) customer)
```

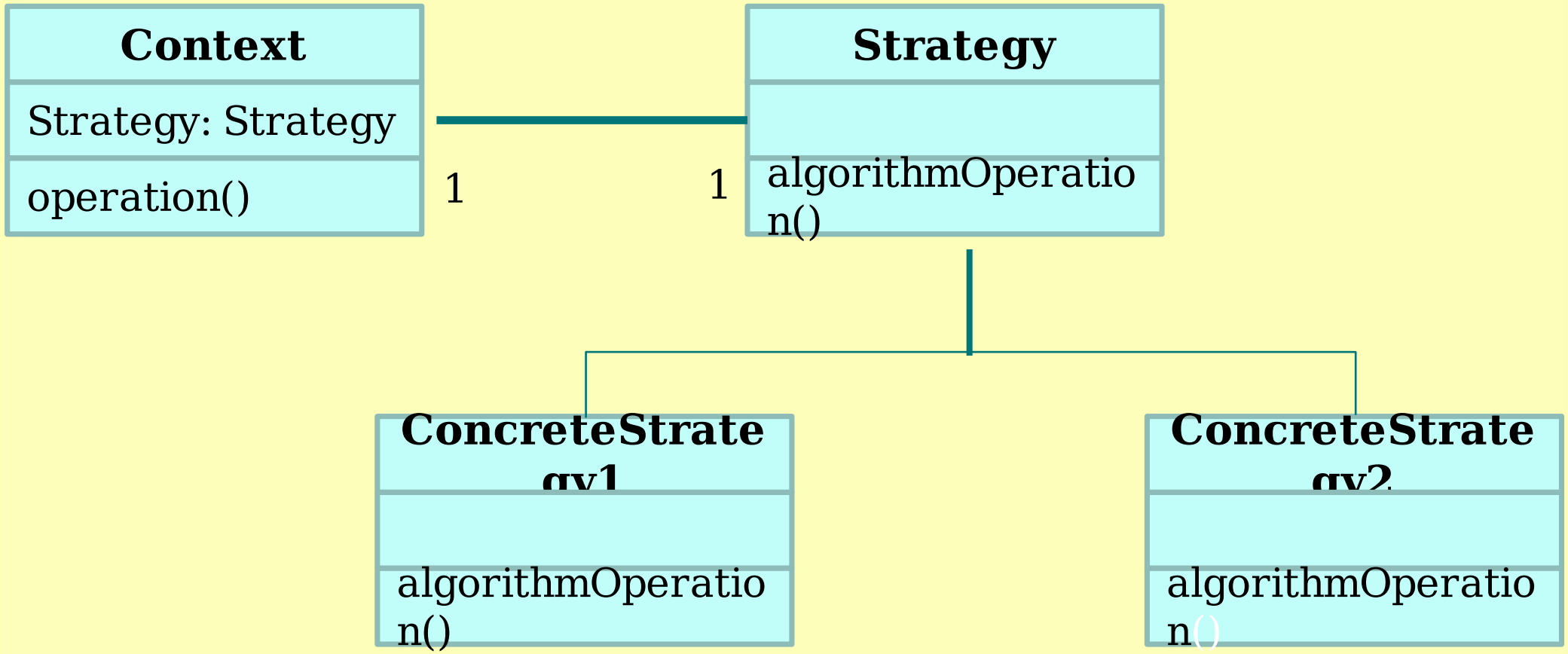
- **This feature should be used wisely**

Lab 5: Creating high order list functions



Parameterizing the strategy behavioral design pattern

- Let’s introduce the strategy pattern first



Parameterizing the strategy behavioral design pattern

- **The strategy patterns is usefull when the application needs to chose between several algorithms or part of algorithms**
- **Examples:**
  - Several tasks are similar except they differ in a small subtask
  - With strategy we define the common parts and we parameterize the varying subtask
  - Applying functional programming to it we can parameterize a family of behaviors

The strategy interface as functional interface

- The strategy interface has only one method
- It is therefore a fit candidate to be used as a functional interface
- In C# we could use a Func delegate to replace it
- In F# we could write a function type

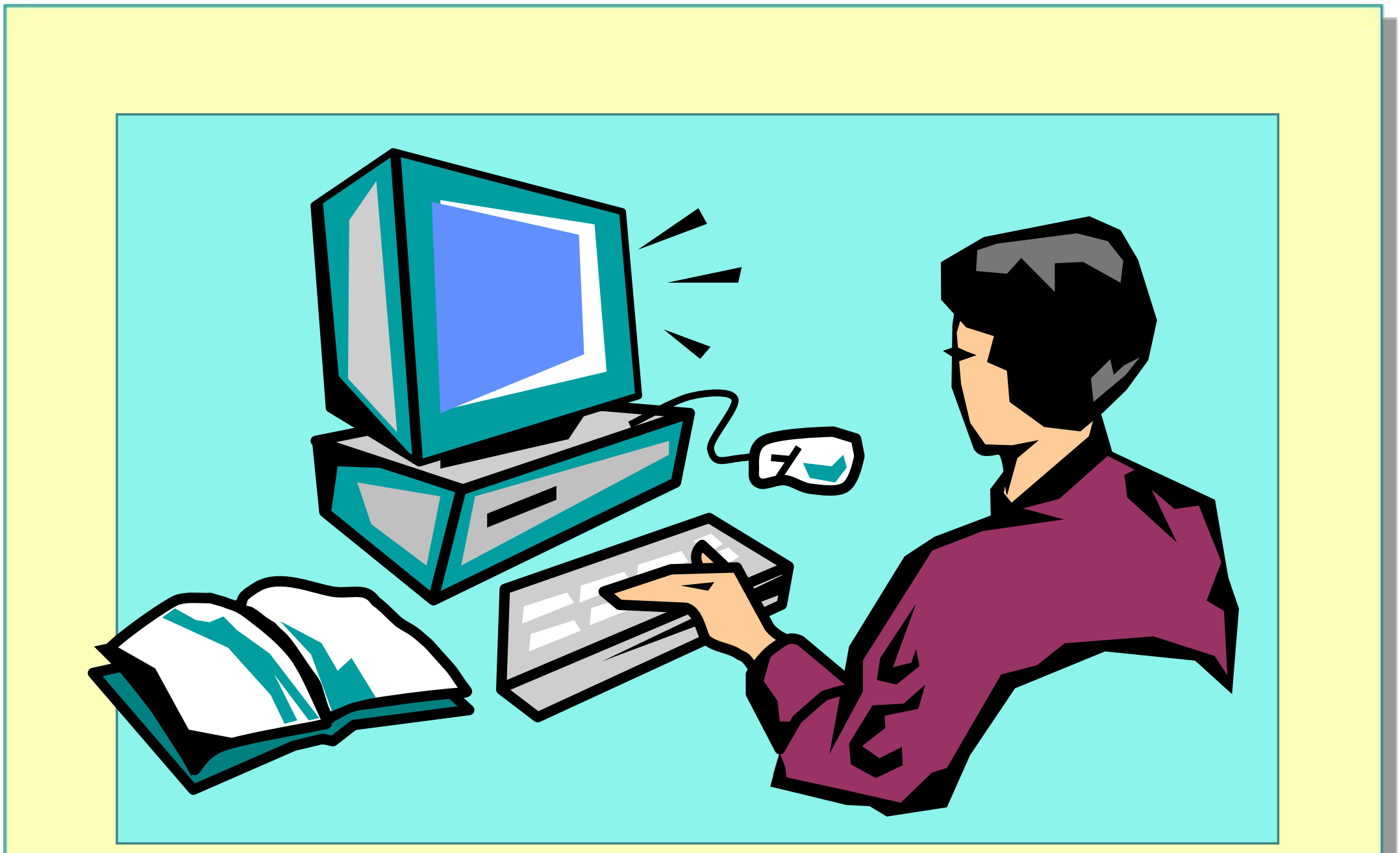
Invocation of strategy via the Context

- We could avoid storing the strategy as a context field and invoke it directly using a lambda

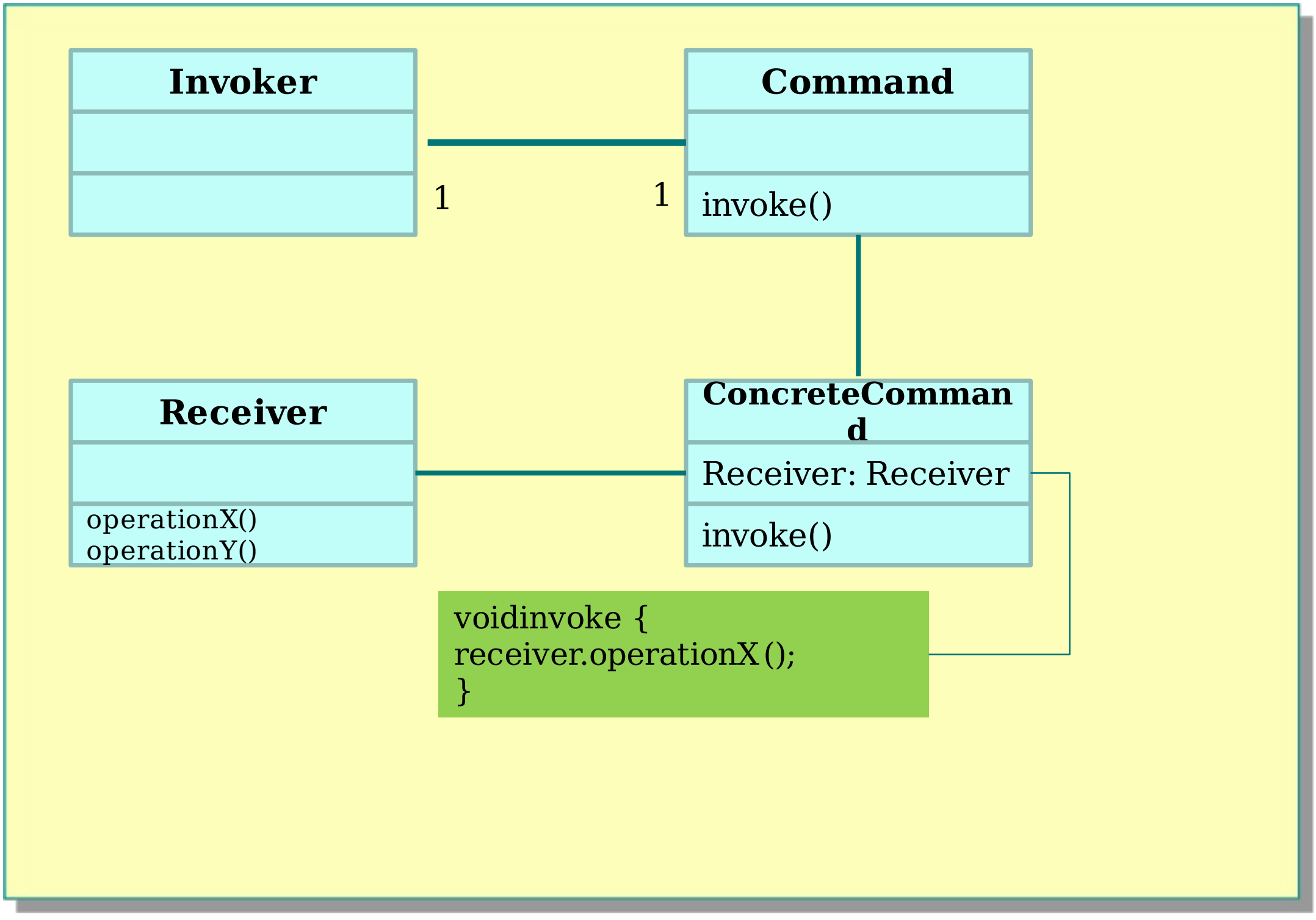
```
Context.operation ( () => { // concretestrategyhere } );
```

- In a functional programming language such as F# we can always replace the Strategy pattern with a high order function

Lab 6: Creating high order filters and mappers



Parameterizing the command design pattern



Parameterizing a family of behaviors

- **The command pattern**
  - Defines a way to represent actions in an application
  - Is used to store “unit of processing” that can invoked later (a trivial case is the Ctrl+Z used to revert actions)
  - Collections of commands are often used to specify steps of operations that the user can choose from (for example in a complex setup wizard)

The command interface as functional interface

- **The command operation interface has only one method**
- **It is therefore a good candidate to be used as a functional interface**

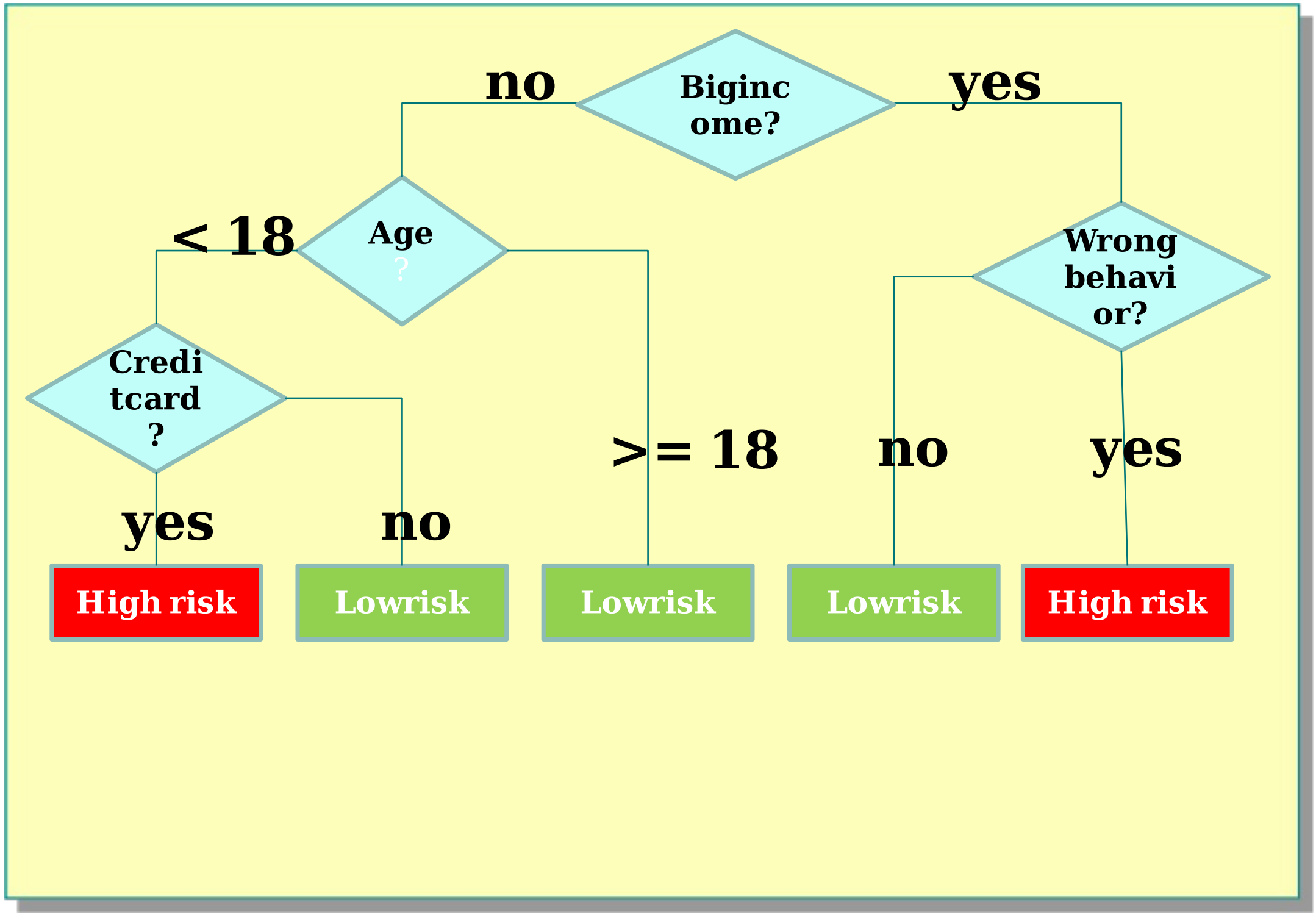
Receiver object and mutable state

- In OOP the receiveris mutable
- It holds somewhat the state and updates it after each command (when necessary)
- In functional programming we DO NOT mutate state
- But we can manage the return value of each command.

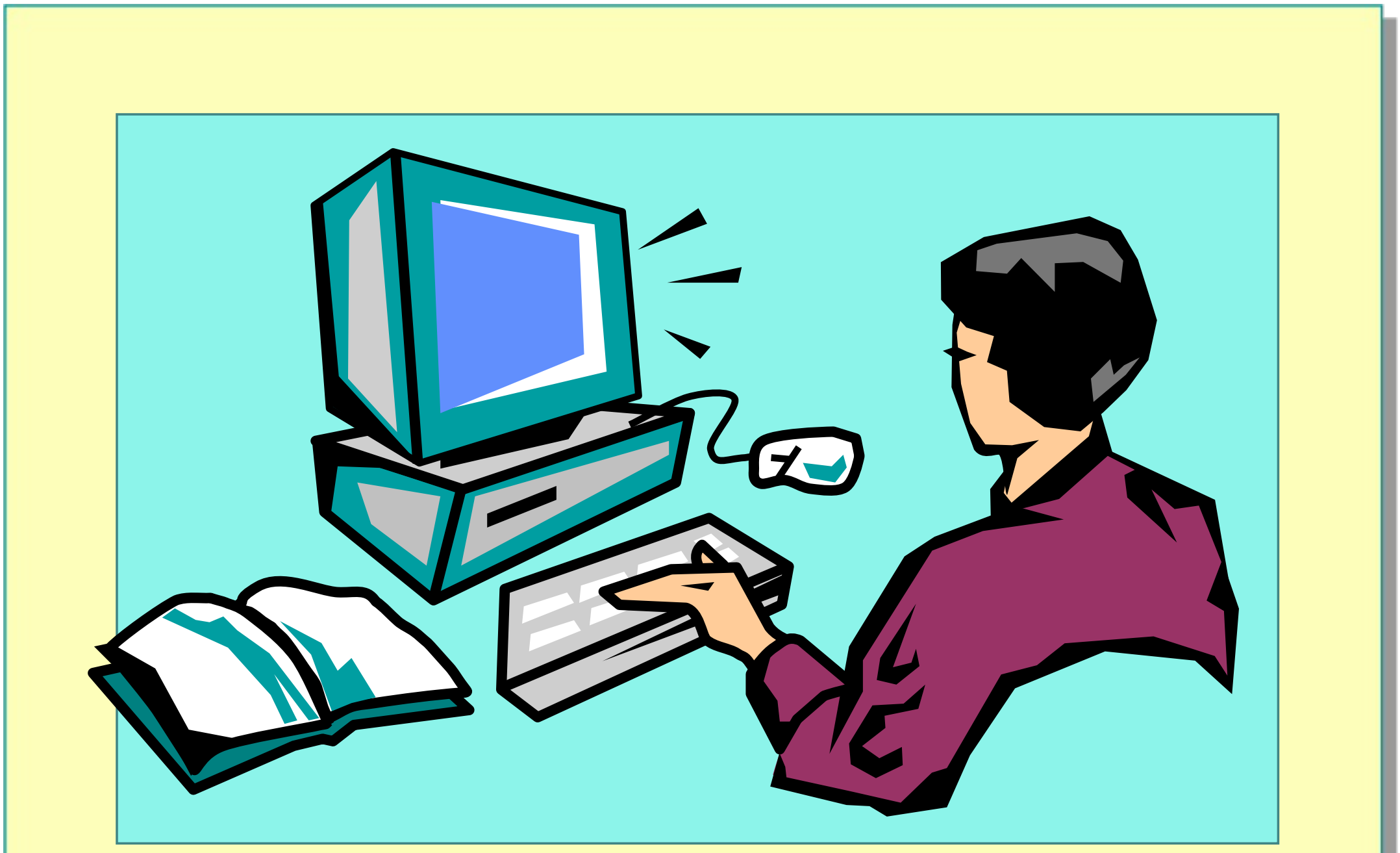
Decision trees

- **Very popular in machine learning**
- **Can be used for:**
  - Makingd ecisions based on some datas
  - Classifying input into various categories
- **The algorithm works with a tree that specifies**
  - What properties of the data should be tested
  - What to be done with each possible answer
- **The reaction can be another test or the final answer**

Decisiontrees (...)



Lab 7: parameterizing behavioral patterns



Wrapping up

- User requirements are sinking sand
- OOP linked behavior to datas and made objects
- But sometimes the ceremony/rituals of OOP are meaningless
- Therefore FP comes to the rescue

Wrapping up (…)

- **Functional programming permits costs optimisations**
- **And helps keep developers happy**
- **It does that by enabling us to parameterize behaviors**
  - By treating functions as values and allowing us to pass them as parameters

Wrapping up (...)

- **We can therefore apply functional programming to behavioral design pattern to build paramaterized families and bundles of functions**
- **And doing so greatly reduce the necessity of modifying code with each fluctuation of user requirements**

QUESTIONS?

THANKYOU!

Resources

- https://fsharpforfunandprofit.com/
- Https://exercism.io
- Https://fsharp.org/
- Https://github.com/alainlompo/fsharp-functions-families-bundles
- Https://repl.it
- Https://www.katacoda.com/courses/fsharp/playground
- Http://www.fsharpworkshop.com/
- http://fssnip.net/