Technology Group

Complexity

Simplified

Functional Programming

Authored & Presented By : **Amit Mulay**

Technical Evangelist

Design

- Data centric design
- Behavior centric design (e.g. filters in batch processing, Web server)

Sometimes you will be using mixture of two design for different parts of product/application

Datacentric Design

OOPs

- Objects/Classes (Data + methods)
- Interaction between different objects

Data FP

- Data
- Data Structures
- Generic functions on Data structures (map, filter, fold, bind)
- Specific functions
- Transformational functions between different data structures

(Data is separate from operations)

(Multiple data structures can be used to represent same data depending on context --- Flat for display/structured for processing/storing)

Datacentric Design

OOPs

- Objects/Classes (Data + methods)
- Interaction between different objects

Data FP

- Data
- Data Structures
- Generic functions on Data structures (map, filter, fold, bind)
- Specific functions
- Transformational functions between different data structures

(Data is separate from operations)

(Multiple data structures can be used to represent same data depending on context --- Flat for display/structured for processing/storing)

Functional datastructures

- Immutable
- List of composed values (Tuple, DU, RecordType)
- Recursive datastructure (like tree, graph) of Tuple/DU/RecordType.

Algebraic datatypes (all of them can be part of other)
Sum – DU
Product – Tuple (positional), RecordType (name based)

```
type Rect = {
        Left : float32;
        Top : float32;
        Width : float32;
        Height : float32
}
let rc = {Left=0.0;Top=1.1;Width=2.0;Height=32}
```

Requirement

Requirement :

Let’s begin by designing a representation of the document that’s suitable for drawing it on the screen. In this representation, the document will be a list of elements with some content (either text or an image) and a specified bounding box in which the content should be drawn.

Sample

[Title : FP]
[Image of Lambda]
[Paragraph containing description]

Requirement

Context : Display

Sample

[Title : FP]

[Image of Lambda]

[Paragraph containing description]

```
type TextContent = {text : string; font : Font}
type ScreenElement =
        | TextElement of TextContent * Rect
        | ImageElement of string * Rect

Datastructure : List of ScreenElements

let Draw loe =
        loe |> list.iter(fun ele ->
                match ele with | TE(txt, rect) ->… | IE(url, rect) ->
        )
```

Hole in the function pattern

```
let Draw loe =
        loe |> list.iter(fun ele ->
                match ele with | TE(txt, rect) ->… | IE(url, rect) ->
        )
```

As a optimization you may want to use in memory bitmaps. But then you need initialization and clean part as well. This is possible only through OOPs concept of constructor, destructor, artificial keywords like “using”.

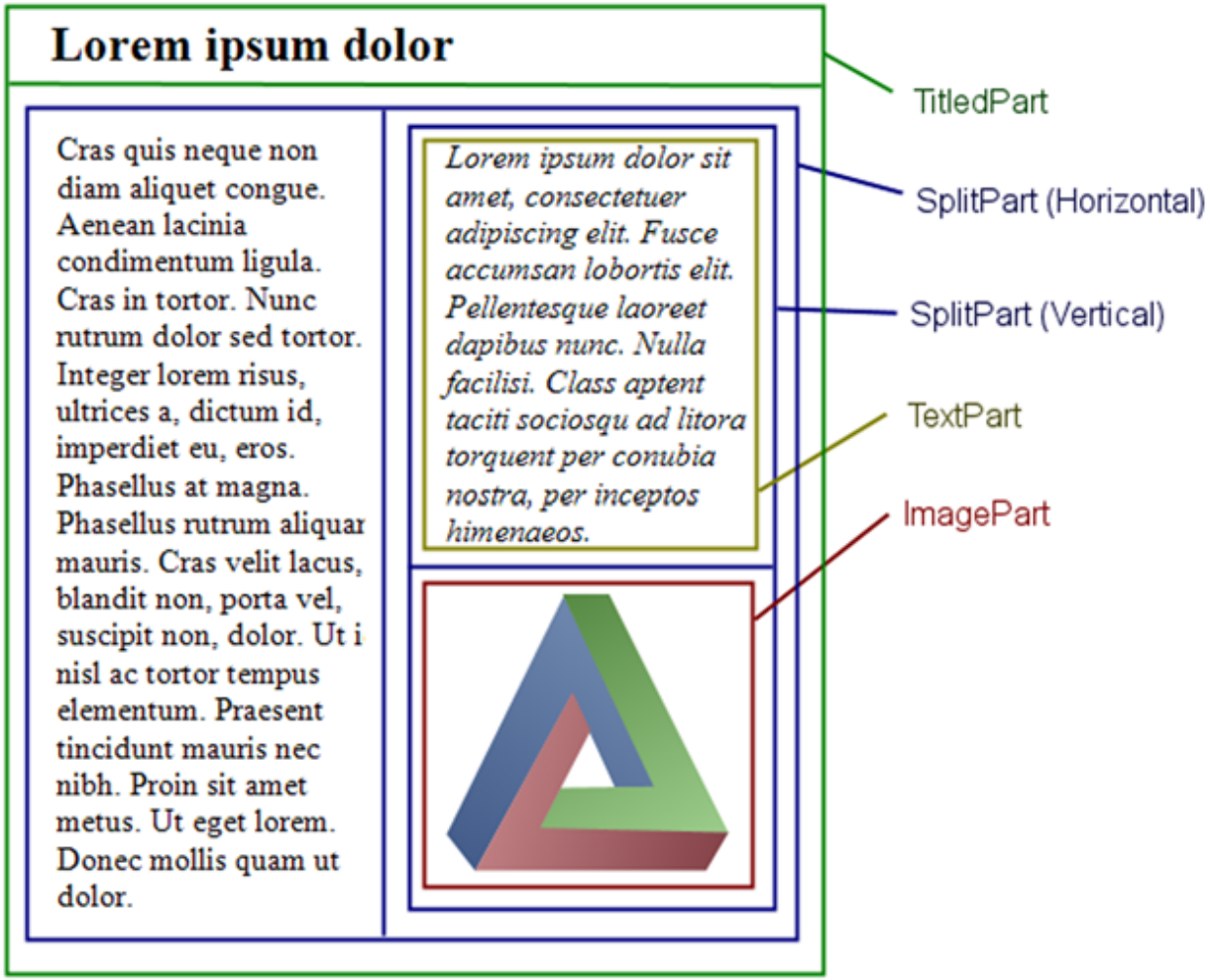
In FP it is achieved through “Hole in the middle” pattern!

```
let Draw wd ht loe coredrawF =
        let bmp = new Bitmap(wd, ht)
        let gr = gr.FromGraphics(bmp)
        coreDrawF loe gr  // Hole
        release gr, bmp // not actual code
```

Requirement

Context : Processing

Sample



Requirement

Context : Processing

type Orientation = | Vertical | Horizontal

type DocumentPart =

| TitlePart of TextContent * DocumentPart

| SplitPart of Orientation * List<DocumentPart>

| TextPart of TextContent

| ImagePart of string

Datastructure : Recursive datastructure DocumentPart

Generic functions :

map : (‘a->’b) -> W<‘a> -> W<‘b>

mapDocument -> (DP -> DP) -> DP -> DP

foldDocument

Specific :

using fold to convert Process DS to Display DS

documentToScreen

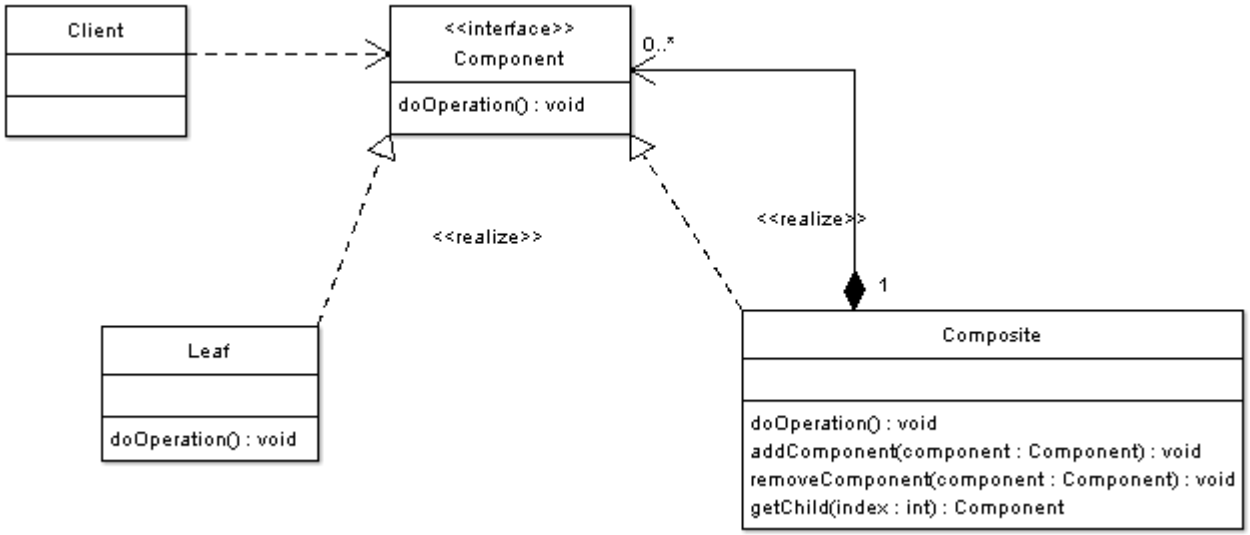
fold

List.fold (+) 0 [1;2;3]
List.fold (*) 1 [1;2;3]
(‘a -> ‘b->’a) ->’a->W<‘b>->’a
(‘a -> DP -> ‘a) -> ‘a -> DP -> ‘a

Can be used to count number of words, creating list of screen elements.

OOPs way

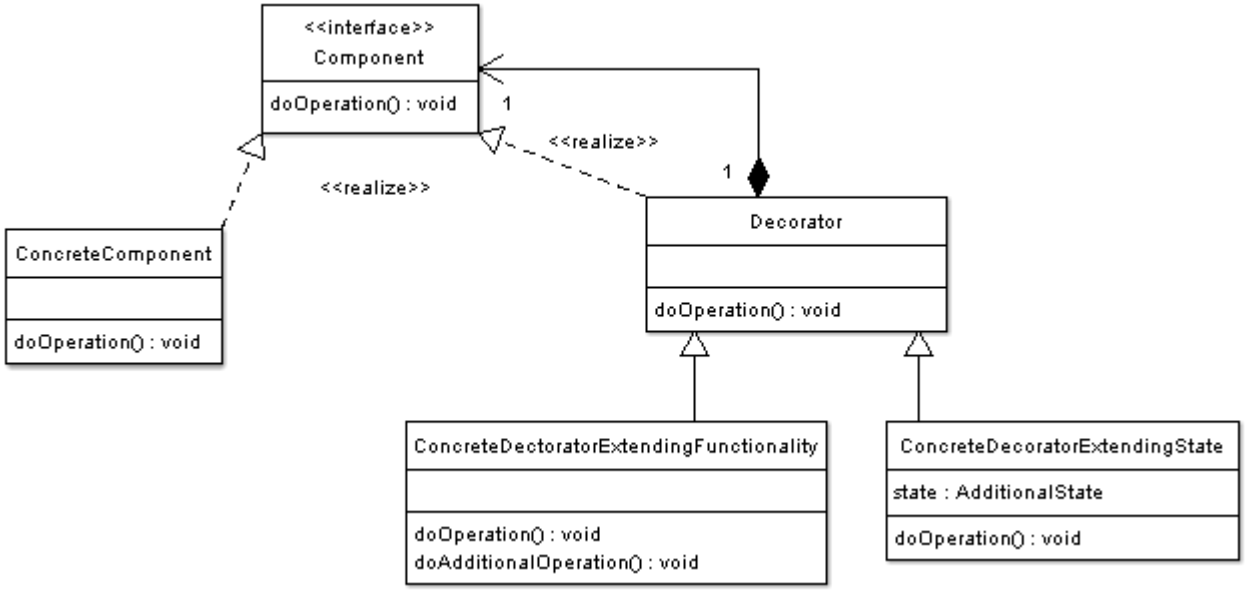
- Composite pattern to structured document



- FP : DU
- FP : Composition is not hidden since adding functionality is more important

OOPs way

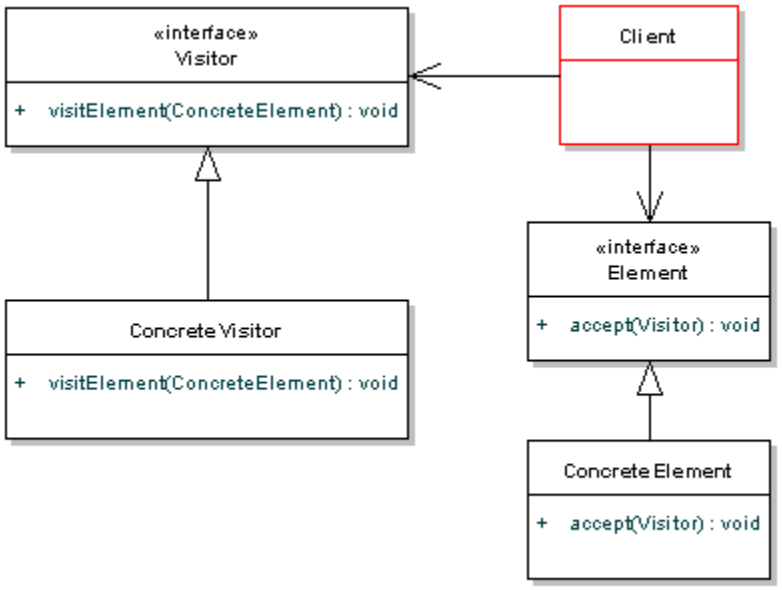
- Decorator pattern for TitledPart



- FP : DU
- FP : Composition is not hidden since adding functionality is more important

OOPs way

- Visitor pattern for adding new operations



Behavior centric Design

- processing of images using filters, the primary data structure would be a list of filters, and from a functional point of view, a filter is a function.
- Preprocessors in webserver
- Filter isn't data
- Filter -> Behavior
- Behavior -> function

If any of the following is true for your application, you're probably designing behavior-centric application:

- You don't (and can't) know the exact set of required features in advance.
- The user or the programmer should be able to add new behaviors easily.
- The program can be extended using external add-ins.
- The user can create tasks by composing simple features.

Behavior centric Design

Develop an application for testing the suitability of a client for a loan offer.

OOPS

```
interface IClientTest {
    bool IsClientRisky(Client client);
}
class TestYearsInJob : IClientTest {
    public bool IsClientRisky(Client client) {
            return client.YearsInJob < 2;
    }
}
```

Collections of IClientTest.

Behavior centric Design

Develop an application for testing the suitability of a client for a loan offer.

Functional (collections of functions)

```
type Client =
{ Name : string; Income : int; YearsInJob : int
UsesCreditCard : bool; CriminalRecord : bool };;
let john =
{ Name = "John Doe"; Income = 40000; YearsInJob = 1
UsesCreditCard = true; CriminalRecord = false };;

 let tests =
[ (fun cl -> cl.CriminalRecord = true);
 (fun cl -> cl.Income < 30000);
(fun cl -> cl.UsesCreditCard = false);
(fun cl -> cl.YearsInJob < 2) ];;

let issues = tests |> List.filter (fun f -> f (client))
```

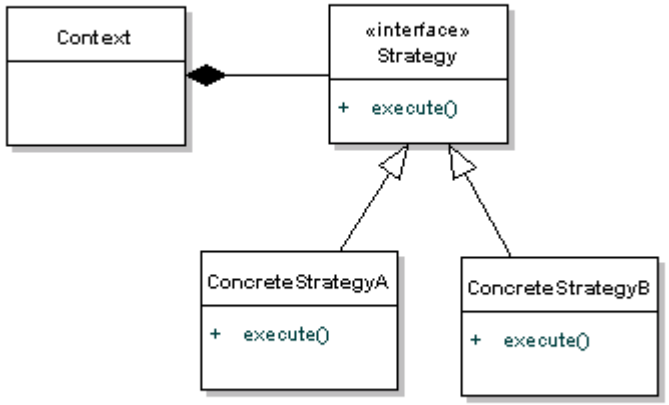
Hint :Single function interface

In FP translates to simple function which will be called by High order function.

Behavior centric Design

Strategy pattern

OOPS



FP : High order function

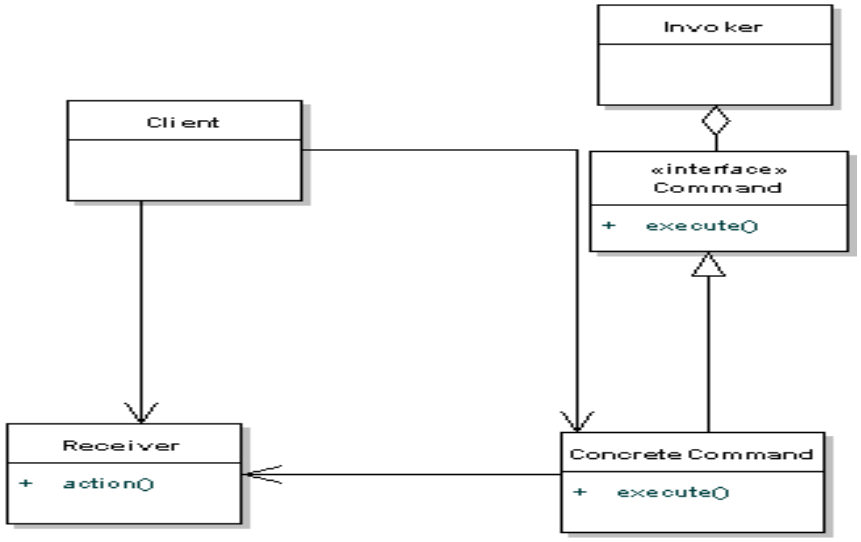
List.filter -> context

Predicate -> concrete strategy

Behavior centric Design

Command pattern

OOPS



FP : collections of functions!

Collection of predicates are command

Hiding State : Closure

Suppose you want use mutable state and at same time want to have feature of hiding state like incase of private in OOPS!

Closure enables us to do it!

```
let createMultiplicationTable table = fun x -> table * x  /*table is closed*/
let table2 = createMultiplicationTable 2 /* int -> int */
let table3 = createMultiplicationTable 3 /* int -> int */

let createIncomeTest() =
        let minimalIncome = ref 30000
        (fun (newMinimal) ->minimalIncome := newMinimal),
        (fun (client) ->client.Income < (!minimalIncome))
val createIncomeTest : unit -> (int -> unit) * (Client -> bool)

let setMinimalIncome, testIncome = createIncomeTest();;
val testIncome : (Client -> bool)
val setMinimalIncome : (int -> unit)
```

Interfaces

```
type iface = {
    f1 : int -> int
    f2 : float -> float
}

// One implementaion
// declaration
let objCons =
    let add (this : (int * float)) b = fst this + b
    let mul (this : (int * float)) b = snd this + b
    fun this -> {f1 = add this; f2 = mul this} // currying

// obj creation
let o = objCons (1, 1.0)

let r1 = o.f1 10
let r2 = o.f2 2.
```

Interfaces

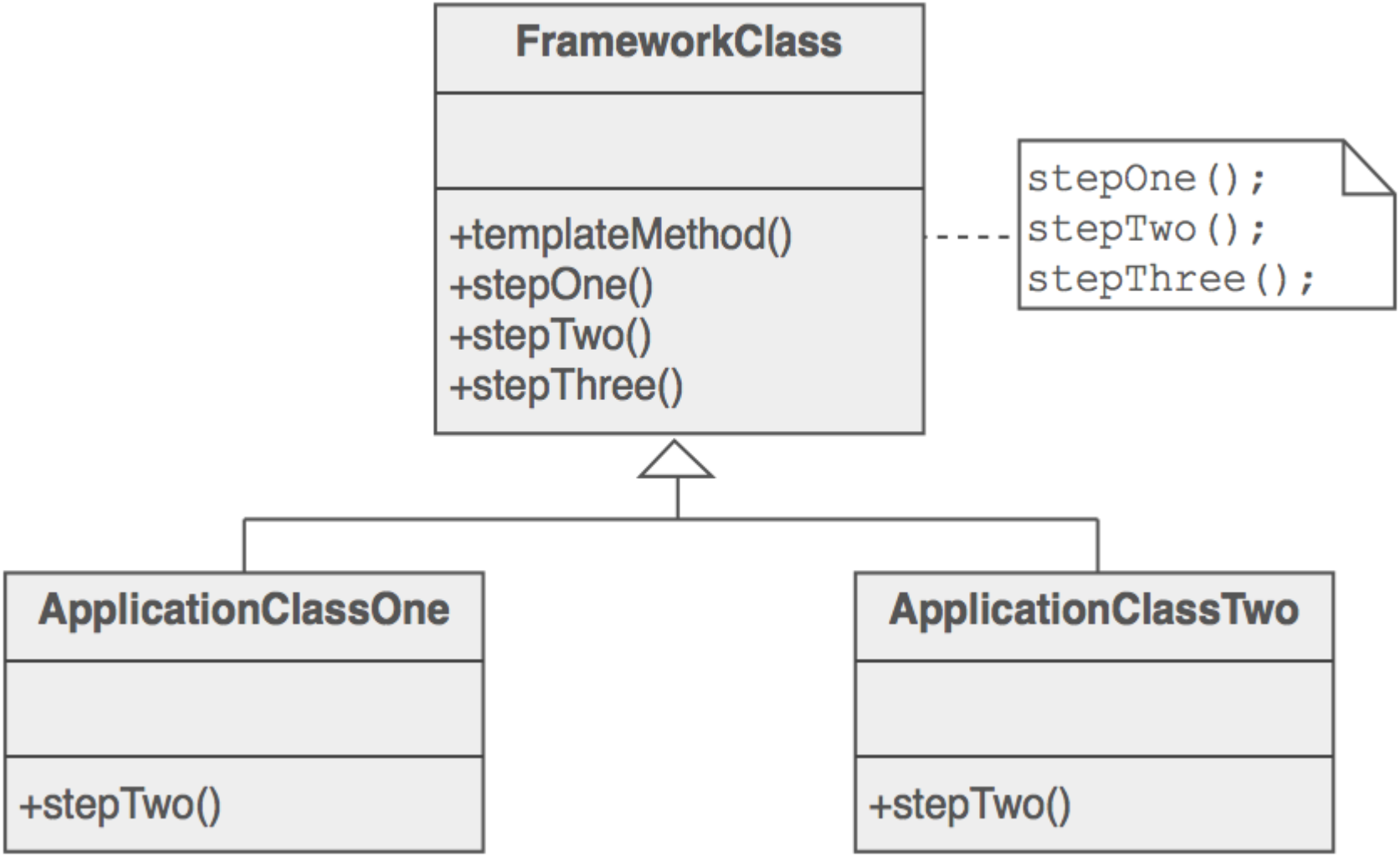
```
type iface = {
    f1 : int -> int
    f2 : float -> float
}

// second implementation
type obja = {a : int ; b: float; c : string}
let objCons1  =
    let addn (this : obja) b = this.a + b
    let muln (this : obja) b = this.b + b
    fun this -> {f1 = addn this; f2 = muln this} // currying

// obj creation
let o1 = objCons1 {a=1;b=2.;c=""}

let r1 = o1.f1 10
let r2 = o1.f2 2.
```

Template Pattern



Any Questions?

Thank you!