UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

#8

30/01/2026

jean-michel.batto@cea.fr

cea

https://gogs.eldarsoft.com/M2_IHPS

Cycle de vie du code

Build

- SFD - Recette
- Livraisons Dev / Préprod / Prod

Run

- PRA

MCO

MCS

Objectifs Prix / Réalisation

Construire un prix ➔ élément déterminant de la vente !

Limiter les risques : connaitre la durée, l'effectif et la contingence.

Durée / Effectif = en JH

Qq métriques = 1 mois 35h = 20 JH en moyenne

La qualification est décrite en 4 niveaux :

- Expert - Xpert
- Senior - P
- Confirmé - Mid
- Junior - A level

Les qualifications

Junior

Sortie d'école

Rôle/Responsabilité : les activités simples/non engageantes contractuellement/pas relations externes

Confirmé

1 a 2 ans

Rôle/Responsabilité : toutes les activités simples sans relations externes/activité moyennement complexes

Senior

3 à 5 ans

Rôle/Responsabilité : toutes les activités avec acteurs internes/externes, supervise les Juniors

Expert

›6 ans

Rôle/Responsabilité : toutes les activités internes/externes, très complexes, supervise Juniors/Confirmés

La productivité du codeur (avant l'IA) - COCOMO

COCOMO est un acronyme pour COnstructive COst MOdel (Modèle de construction de coût).

Nom du langage	Date de création	Cible	Typologie	Cocomo (lignes / jours)	Expressivité
Smalltalk	1980	bytecode	Objet Typ.dynamique	15	6* le C
C++	1985	Compilateur	Objet hybride Typ.statique	25	2* le C
VHDL	1987	FPGA	Procédural Typ.statique	2 (→le FPGA)	1/6 du C
Delphi Pascal	1995	Compilateur	Objet simple Typ.statique	30	5* le C
Java	1995	bytecode	Objet simple Typ.statique	40	4* le C
Python	2003	Interpreteur	Tous les paradigmes Typ.dynamique	40	4* le C

Cocomo / LOC / FP

LOC = Line Of Code, FP = Function Point (=Classe en POO)

IFPUG is a non-profit, member governed organization (USA) that endorses two types of standard methodology for software sizing.

https://www.ifpug.org/wp-content/uploads/2017/04/IYSM.-Thirty-years-of-IFPUG.-Software-Economics-and-Function-Point-Metrics-Capers-Jones.pdf

The LOC metric originated in the 1950's when machine language and basic assembly were the only languages in use. In those early days coding was over 95% of the total effort so the fixed costs of non-code work barely mattered. It was only after high-level programming languages began to reduce coding effort and requirements and design became progressively larger components that the LOC problems occurred.

Go : 554 LOC per Month
C++ : 554 LOC per Month
➔10 FP per Month ➔ 2 * C
25 LOC/day * 22 day = 550 LOC

Table 16: Side-by-Side Comparison of function points and lines of code metrics 2017

	Languages	Size in KLOC	Total Work hours	Work hours per FP	FP per Month	Work Months	Work hours per KLOC	LOC per Month
1	Machine language	640.00	119,364	119.36	1.11	904.27	186.51	708
2	Basic Assembly	320.00	61,182	61.18	2.16	463.50	191.19	690
3	JCL	220.69	43,125	43.13	3.06	326.71	195.41	675
4	Macro Assembly	213.33	41,788	41.79	3.16	316.57	195.88	674
5	HTML	160.00	32,091	32.09	4.11	243.11	200.57	658
6	C	128.00	26,273	26.27	5.02	199.04	205.26	643
7	XML	128.00	26,273	26.27	5.02	199.04	205.26	643

Chiffrer/Deviser

EB : écrire un programme qui fait le tri d'1 fichier **ascii** avec 2 colonnes – séparateur tabulation – trier la valeur de la première colonne qui est un nom de 5 caractères. Le résultat est un fichier.

SFD : - en C

(A) Ouvrir le fichier

(B) Lire le fichier et convertir les valeurs texte (avec vérification si c'est un nom de 5 caractères) – fermer le fichier

(C) Faire un tri

(D) Ecrire le résultat avec une tabulation

TD Estimation des coûts

Etape	Estimation LOC	JH
A	10	1 JH
B	20	
C	5	
D	15	

1 JH : écrire l'EB / SFD / Recette
1 JH : écrire le code et test (50 LOC)

Estimation des coûts

Etape	Estimation LOC	JH
A	10	1 JH
B	20	
C	5	
D	15	

1 JH : écrire l'EB / SFD / Recette
1 JH : écrire le code et test (50 LOC)

Attention : les IA génératives ajoutent un gain de *4 à *5 en productivité

Vision HPC - MPI

SFD : - en C++

(A) Ouvrir le fichier

(B1) Lire le fichier et convertir les valeurs (avec vérification si c'est un nombre) – fermer le fichier

(B2)Initialiser les nœuds MPI

Distribuer le tri MPI (C1)

Fusion du résultat (C2)

(D) Ecrire le résultat avec une tabulation

Estimation des coûts, apport HPC

Etape	Estimation LOC	JH
A	10	2 JH
B1	20	
B2	5	
C1	30	
C2	5	
D	15	

1 JH : écrire l'EB / SFD / Recette
2 JH : écrire le code et test (85 LOC > 1J travail)

Estimation des coûts (apport POO)

Etape	Estimation LOC	JH
A	10	1 JH
B	20	
C	5	
D	15	

1 JH : écrire l'EB / SFD / Recette
1 JH : écrire le code et test (50 LOC)

Les étapes B et C peuvent être abstraites en POO – peut-être que l'on va doubler le code de B et C.

Bénéfice POO : facile si c'est en mode « pas d'ajout plus tard ».
➔PB lisibilité du gain et de l'anticipation (qui paye l'investissement?)

Vision HPC - MPI

SFD : - en C++

(A) Ouvrir le fichier

(B1) Lire le fichier et convertir les valeurs (avec vérification si c'est un nombre) – fermer le fichier

(B2)Initialiser les nœuds MPI

Distribuer le tri MPI (C1)

Fusion du résultat (C2)

(D) Ecrire le résultat avec une tabulation

Estimation des coûts, apport HPC

Etape	Estimation LOC	JH
A	10	2 JH
B1	20	
B2	5	
C1	30	
C2	5	
D	15	

1 JH : écrire l'EB / SFD / Recette

2 JH : écrire le code et test (85 LOC > 1J travail)

Chiffrer/Deviser

EB : écrire un programme qui fait le tri d'1 fichier **ascii** avec 2 colonnes – séparateur tabulation – trier la valeur de la première colonne qui est un nom de 5 caractères. Le résultat est un fichier.

SFD : - en C

(A) Ouvrir le fichier

(B) Lire le fichier et convertir les valeurs texte (avec vérification si c'est un nom de 5 caractères) – fermer le fichier

(C) Faire un tri

(D) Ecrire le résultat avec une tabulation

TD Estimation des coûts

Etape	Estimation LOC	JH
A	10	1 JH
B	20	
C	5	
D	15	

1 JH : écrire l'EB / SFD / Recette
1 JH : écrire le code et test (50 LOC)

Les étapes B et C peuvent être abstraites en POO – peut-être que l'on va doubler le code de B et C.

Bénéfice POO : facile si c'est en mode « dent creuse » sinon, qui paye?
➔PB lisibilité du gain et de l'investissement

Bonne Pratique

1 : utiliser un tableau (exemple joint)
2 : afficher les mois → permet de recoller les jalons

	M1	M2	M3	Durée JH	Durée x Cout JH		Cout JH
DEV1	19	20	20	59	19470		330
DEV2	0	20	20	40	13200		
			Durée Total	99			
				Cout en JH	32 670,00 €		
					34 303,50 €	5%	contingence
				Cout interne	38 419,92 €	12%	garantie
				Prix client	54 885,60 €	30%	marge

TD5 – pour dimanche 8 février 2026, 23h59

Il s'agit de chiffrer le cout du projet cuicui (document cuicui.pdf)

Vous partez de l'EB, de la SFD et des diagrammes présentés dans le document.
Vous estimez le nombre de LOC en C++ & le temps pour les TESTS
Vous estimez les JH pour une équipe de 2 DEV – mois à 20JH
A la fin, je vous demande un prix. A rendre **le tableau XLS** – dans le fichier les infos pertinentes.

La synthèse du prix et vos prompts dans un document PDF.

Les patrons / patterns en Golang

Les 23 patterns :

<table>
<tr><th></th><th>Creational</th><th>Structural</th><th>Behavioral</th></tr>
<tr><td rowspan="2">Class</td><td rowspan="2">Factory Method</td><td rowspan="2">Adapter (class)</td><td>Interpreter</td></tr>
<tr><td>Template Method</td></tr>
<tr><td rowspan="8">Object</td><td>Abstract Factory</td><td>Adapter (object)</td><td>Chain of Responsibility</td></tr>
<tr><td>Builder</td><td>Bridge</td><td>Command</td></tr>
<tr><td>Prototype</td><td>Composite</td><td>Iterator</td></tr>
<tr><td rowspan="6">Singleton</td><td>Decorator</td><td>Mediator</td></tr>
<tr><td>Facade</td><td>Memento</td></tr>
<tr><td>Flyweight</td><td>Observer</td></tr>
<tr><td rowspan="3">Proxy</td><td>State</td></tr>
<tr><td>Strategy</td></tr>
<tr><td>Visitor</td></tr>
</table>

Le fichier PPCS-CM8-2025/23pattern-go/23pattern-go.go donne les exemples d'invocation

Les 23 patterns en Go

```
// creational package pattern
// 1. Singleton Pattern -- une seule instance et fournir un point d'accès global à cette instance
type singleton struct {
        Data string
}

var (
        instance *singleton
        once     sync.Once
)

func GetInstance() *singleton {
        once.Do(func() {
                instance = &singleton{Data: "I am singleton"}
        })
        return instance
}
```

Les 23 patterns en Go

// creational package pattern

```
// 2. Factory Method Pattern -- interface pour la création d'un objet, mais laisse aux sous-classes le choix des classes concrètes à instancier
type Animal interface {
        Speak() string
}

type Dog struct{}
type Cat struct{}

func (d *Dog) Speak() string { return "Woof!" }
func (c *Cat) Speak() string { return "Meow!" }

func CreateAnimal(animalType string) Animal {
        switch animalType {
        case "dog":
                return &Dog{}
        case "cat":
                return &Cat{}
        default:
                return nil
        }
}
```

Les 23 patterns en Go

// creational package pattern

```
// 3. Abstract Factory Pattern -- définit une famille de classes interchangeables sans spécifier leur classe concrète
type Button interface {
        Paint()
}
type WinButton struct{}
type MacButton struct{}

func (w *WinButton) Paint() { println("Windows button") }
func (m *MacButton) Paint() { println("Mac button") }

type GUIFactory interface {
        CreateButton() Button
}
type WinFactory struct{}
type MacFactory struct{}

func (w *WinFactory) CreateButton() Button { return &WinButton{} }
func (m *MacFactory) CreateButton() Button { return &MacButton{} }
```

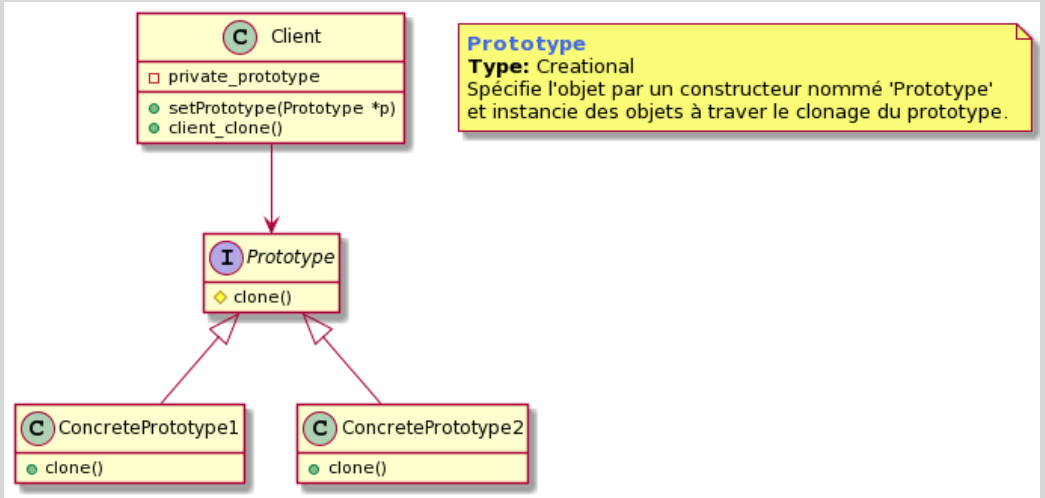
Les 23 patterns en Go

```
// creational package pattern
// 4. Builder Pattern -- composite avec structure indépendante
type House struct {
        Windows int
        Doors   int
        Roof    string
}
type HouseBuilder struct {
        house *House }
func NewHouseBuilder() *HouseBuilder {
        return &HouseBuilder{house: &House{}} }
func (b *HouseBuilder) SetWindows(count int) *HouseBuilder {
        b.house.Windows = count
        return b }
func (b *HouseBuilder) SetDoors(count int) *HouseBuilder {
        b.house.Doors = count
        return b }
func (b *HouseBuilder) SetRoof(style string) *HouseBuilder {
        b.house.Roof = style
        return b }
func (b *HouseBuilder) Build() *House {
        return b.house }}
```

Les 23 patterns en Go

// creational package pattern

```
// 5. Prototype Pattern -- le constructeur de copie fait l'instanciation
type Prototype interface {
        Clone() Prototype
}

type ConcretePrototype struct {
        Name string
}

func (p *ConcretePrototype) Clone() Prototype {
        return &ConcretePrototype{Name: p.Name}
}
```



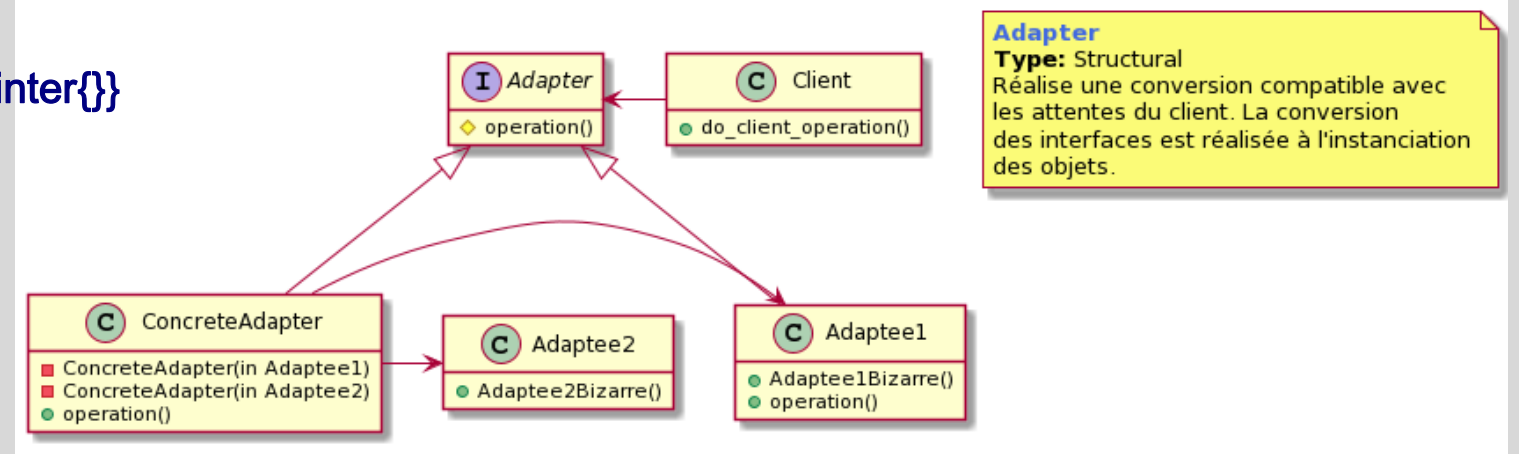
Les 23 patterns en Go

// structural package pattern

https://go.dev/play/p/oQD5kyde2yT

// 6. Adapter Pattern -- crée une interface pour plusieurs sous-systèmes de manière à ce qu'ils puissent interagir de manière interchangeable

```
type LegacyPrinter interface {
        Print(s string) string
}
type MyLegacyPrinter struct{}
func (p *MyLegacyPrinter) Print(s string) string {
        return fmt.Sprintf("Legacy: %s", s) }
type ModernPrinter interface {
        PrintModern(s string) string }
type PrinterAdapter struct {
        OldPrinter LegacyPrinter }
func (p *PrinterAdapter) PrintModern(s string) string {
        return p.OldPrinter.Print(s)
}

adapter := &PrinterAdapter{OldPrinter: &MyLegacyPrinter{}}
```



Les 23 patterns en Go

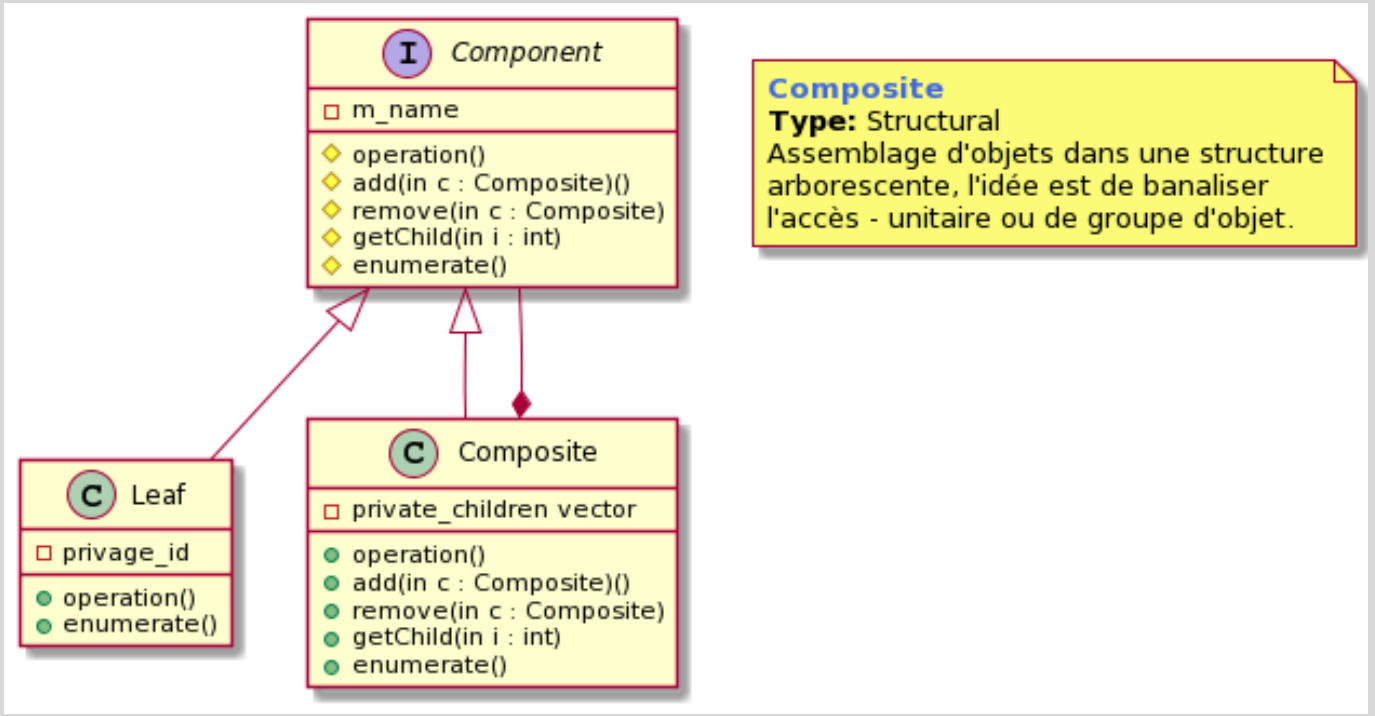
```
// structural package pattern
// 7. Bridge Pattern -- découplage
type DrawAPI interface {
        DrawCircle(x, y, radius int)
}
type RedCircle struct{}
type GreenCircle struct{}

func (r *RedCircle) DrawCircle(x, y, radius int) {
        fmt.Printf("Drawing red circle at (%d,%d) radius %d\n", x, y, radius)
}
func (g *GreenCircle) DrawCircle(x, y, radius int) {
        fmt.Printf("Drawing green circle at (%d,%d) radius %d\n", x, y, radius)
}
type Circle struct {
        api    DrawAPI
        X, Y, Radius int
}

func (c *Circle) SetAPI(api DrawAPI) { c.api = api }
func (c *Circle) Draw() { c.api.DrawCircle(c.X, c.Y, c.Radius) }
```

Les 23 patterns en Go

// structural package pattern

// 8. Composite Pattern -- structure arborescente/uniforme

```
type Component interface {
        Operation() string
}
type Leaf struct {
        Name string
}
func (l *Leaf) Operation() string { return l.Name }
type Composite struct {
        children []Component
}
func (c *Composite) Add(child Component) {
        c.children = append(c.children, child)
}
func (c *Composite) Operation() string {
        result := "Branch("
        for i, child := range c.children {
                result += child.Operation()
                if i < len(c.children)-1 {
                        result += ", "
                }
        }
        return result + ")"}
```

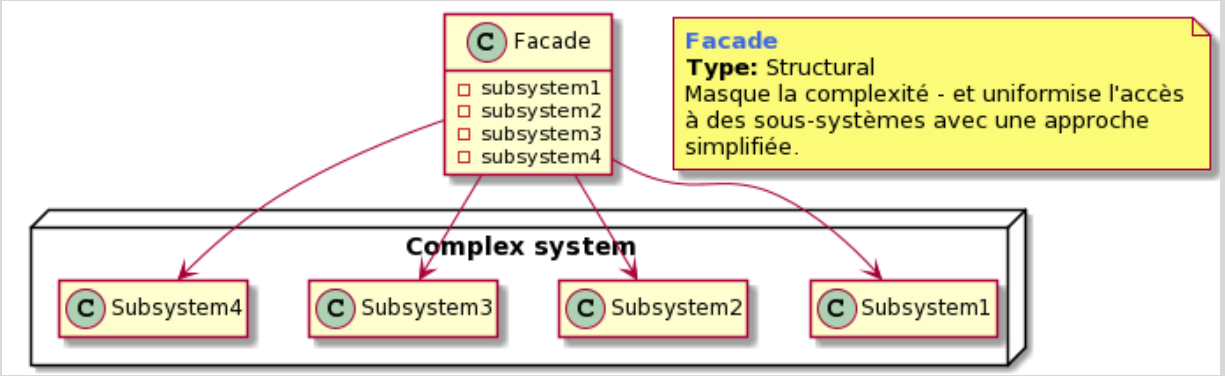


Les 23 patterns en Go

```
// structural package pattern
// 9. Decorator Pattern -- attache des responsabilités supplémentaires à un objet de manière dynamique
type Coffee interface {
        Cost() int
        Description() string
}

type SimpleCoffee struct{}

func (s *SimpleCoffee) Cost() int          { return 5 }
func (s *SimpleCoffee) Description() string { return "Simple coffee" }

type MilkDecorator struct {
        Coffee Coffee
}

func (m *MilkDecorator) Cost() int          { return m.Coffee.Cost() + 2 }
func (m *MilkDecorator) Description() string { return m.Coffee.Description() + ", milk" }
```

Les 23 patterns en Go

```
// structural package pattern
// 10. Facade Pattern -- regroupe plusieurs interfaces en une seule interface
type SubsystemA struct{}
func (s *SubsystemA) OperationA() string { return "SubA" }

type SubsystemB struct{}
func (s *SubsystemB) OperationB() string { return "SubB" }

type Facade struct {
        a *SubsystemA
        b *SubsystemB
}
func NewFacade() *Facade {
        return &Facade{&SubsystemA{}, &SubsystemB{}}
}

func (f *Facade) Operation() string {
        return f.a.OperationA() + " + " + f.b.OperationB()
}
```

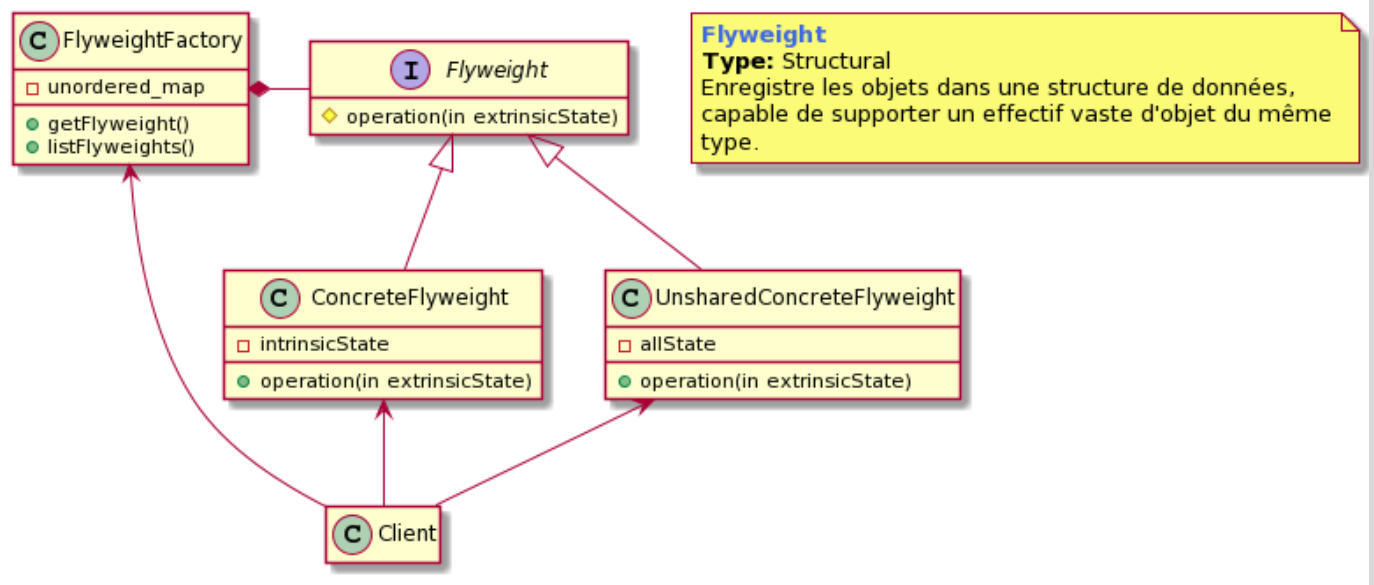


Les 23 patterns en Go

// structural package pattern

// 11. Flyweight Pattern -- partage de l'état

```
type CharacterFlyweight struct {
        Char rune
}

type CharacterFactory struct {
        chars map[rune]*CharacterFlyweight
}

func NewCharacterFactory() *CharacterFactory {
        return &CharacterFactory{chars: make(map[rune]*CharacterFlyweight)}
}

func (f *CharacterFactory) GetCharacter(c rune) *CharacterFlyweight {
        if _, ok := f.chars[c]; !ok {
                f.chars[c] = &CharacterFlyweight{Char: c}
        }
        return f.chars[c]
}
```



Les 23 patterns en Go

// structural package pattern

```
// 12. Proxy Pattern -- effet miroir
type Subject interface {
	Request() string
}

type RealSubject struct{}

func (s *RealSubject) Request() string { return "RealSubject active" }

type Proxy struct {
	realSubject *RealSubject
}

func (p *Proxy) Request() string {
	if p.realSubject == nil {
		p.realSubject = &RealSubject{}
	}
	return "Proxy: " + p.realSubject.Request()
}
```

Les 23 patterns en Go

```
// behavioral package pattern
// 13. Chain of Responsibility Pattern -- répond à une demande avec découplage
type Handler interface {
        SetNext(handler Handler)
        Handle(request string) string
}
type BaseHandler struct {
        next Handler
}
func (b *BaseHandler) SetNext(h Handler) { b.next = h }
type ConcreteHandler1 struct{ BaseHandler }
func (h *ConcreteHandler1) Handle(req string) string {
        if req == "one" { return "Handled by 1" }
        if h.next != nil { return h.next.Handle(req) }
        return "Not handled"
}
type ConcreteHandler2 struct{ BaseHandler }
func (h *ConcreteHandler2) Handle(req string) string {
        if req == "two" { return "Handled by 2" }
        if h.next != nil { return h.next.Handle(req) }
        return "Not handled"
}
```

Les 23 patterns en Go

// behavioral package pattern

// 14. Command Pattern -- encapsuler une requête sous la forme d'un objet, permettant la paramétrisation des clients avec différentes requêtes

```
type Command interface { Execute() string }

type Light struct{}
func (l *Light) On() string { return "Light On" }

type LightOnCommand struct { Light *Light }
func (c *LightOnCommand) Execute() string { return c.Light.On() }
```

Les 23 patterns en Go

// behavioral package pattern

// 15. Interpreter Pattern -- grammaire dans l'objet

```
type Expression interface { Interpret() bool }
type TerminalExpression struct { Data string }
func (t *TerminalExpression) Interpret() bool { return len(t.Data) > 0 }
```

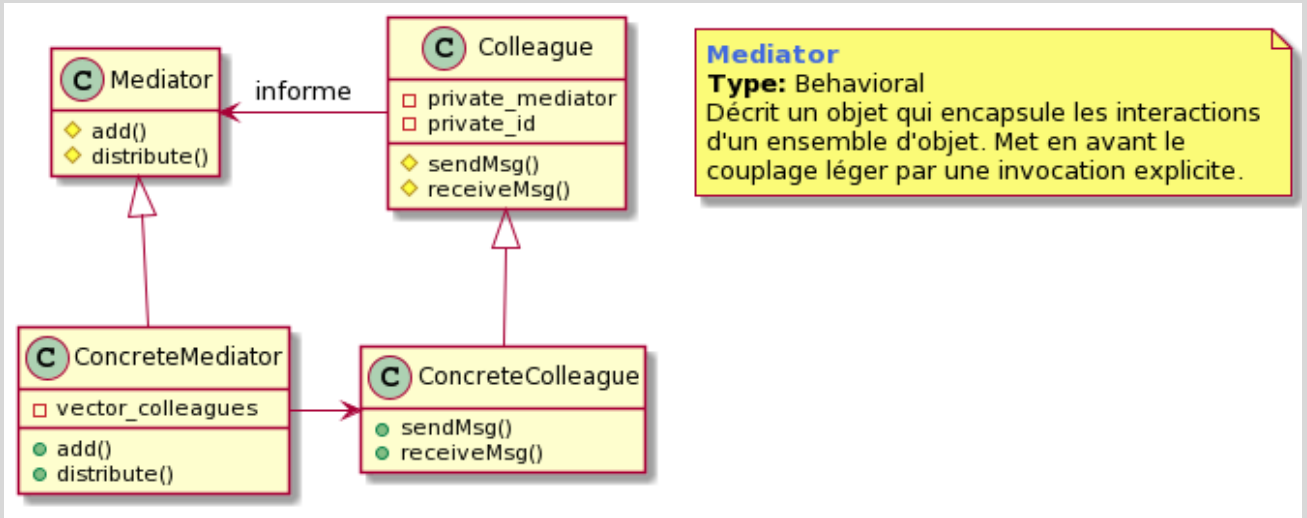
Les 23 patterns en Go

```
// behavioral package pattern
// 16. Iterator Pattern -- se déplace sans connaitre le détail de l'implémentation
type Iterator interface {
        HasNext() bool
        Next() interface{}
}
type Collection struct { Items []interface{} }
func (c *Collection) CreateIterator() Iterator {
        return &CollectionIterator{collection: c}
}
type CollectionIterator struct {
        collection *Collection
        index      int
}
func (i *CollectionIterator) HasNext() bool { return i.index < len(i.collection.Items) }
func (i *CollectionIterator) Next() interface{} {
        item := i.collection.Items[i.index]
        i.index++
        return item
}
```

Les 23 patterns en Go

// behavorial package pattern

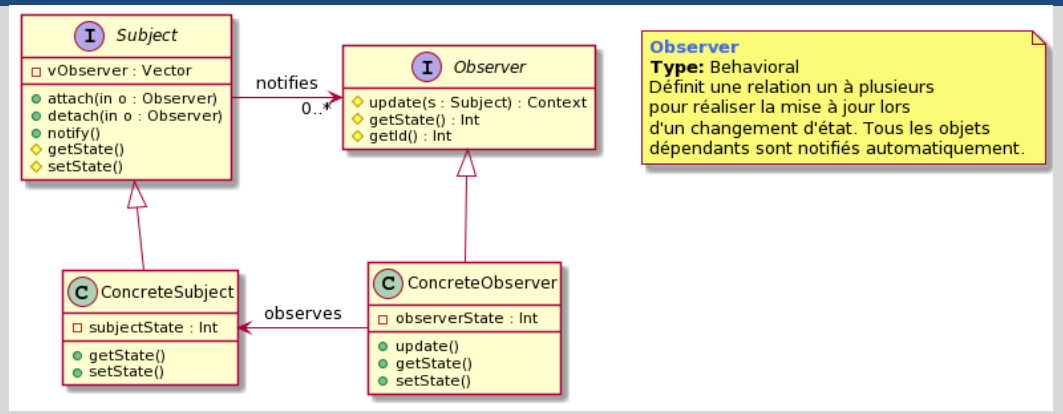
```
// 17. Mediator Pattern -- spécifie les interactions entre objets sans définir leurs classes concrètes
type Mediator interface { Notify(sender interface{}, event string) }

type ConcreteMediator struct {
        c1 *Component1
}

func (m *ConcreteMediator) SetComponent1(c *Component1) { m.c1 = c }
func (m *ConcreteMediator) Notify(s interface{}, e string) {
        fmt.Printf("Mediator: reacting to %s\n", e)
}

type Component1 struct { Mediator Mediator }
func (c *Component1) Trigger() { c.Mediator.Notify(c, "click") }
```



// behavioral package pattern

```
// 18. Memento Pattern -- informations privées pour « backup » de 1 état
type Memento struct { State string }
type Originator struct { State string }
func (o *Originator) CreateMemento() *Memento { return &Memento{State: o.State} }
func (o *Originator) RestoreMemento(m *Memento) { o.State = m.State }
```

// behavioral package pattern



// 19. Observer Pattern -- définit une dépendance entre objets afin qu'un changement de l'état d'un objet entraîne une mise à jour automatique

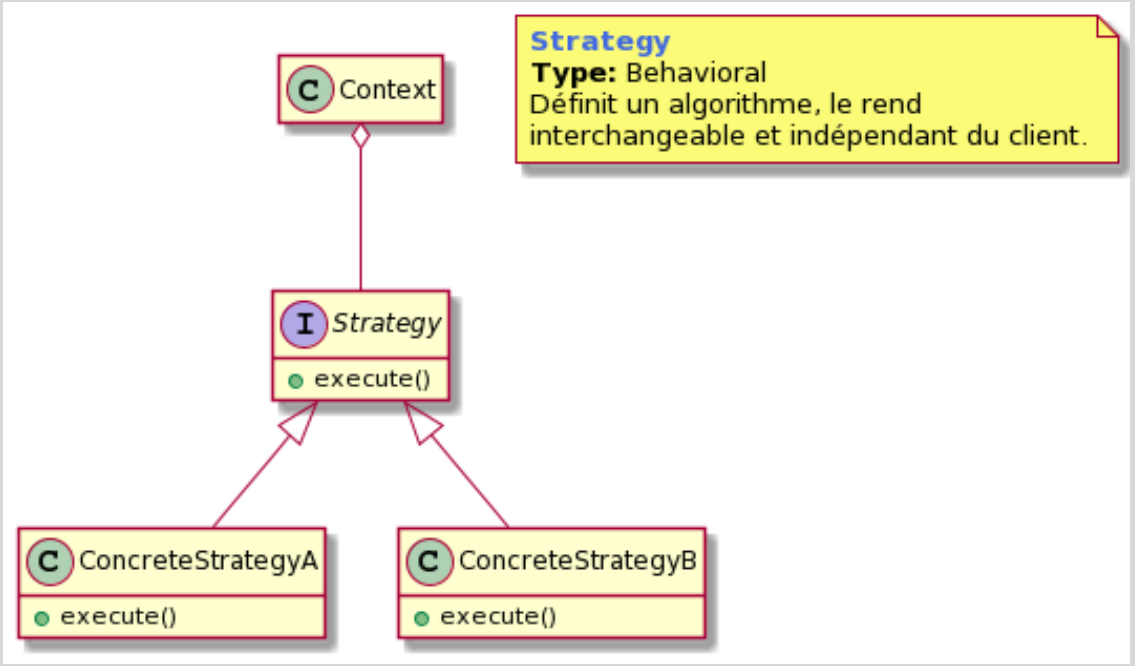
```
type Observer interface { Update(string) }
type Subject2 struct {
        observers []Observer
        state     string
}

func (s *Subject2) Attach(o Observer) { s.observers = append(s.observers, o) }
func (s *Subject2) SetState(st string) {
        s.state = st
        for _, o := range s.observers { o.Update(st) }
}

type ConcreteObserver struct{ ID int }
func (o *ConcreteObserver) Update(s string) { fmt.Printf("Observer %d: state is %s\n", o.ID, s) }
```

// behavioral package pattern

```
// 20. State Pattern -- les états sont gérés par un handle
type State interface { Handle() string }
type Context struct { State State }
func (c *Context) Request() string { return c.State.Handle() }

type ConcreteStateA struct{}
func (s *ConcreteStateA) Handle() string { return "State A" }
}
```

Les 23 patterns en Go

```
// behavioral package pattern
// 21. Strategy Pattern -- algorithme
type Strategy interface { Execute() string }
type ConcreteStrategyA struct{}
func (s *ConcreteStrategyA) Execute() string { return "Strategy A" }

type ContextStrat struct { Strategy Strategy }
func (c *ContextStrat) Execute() string { return c.Strategy.Execute() }
```



Les 23 patterns en Go

// behavioral package pattern

```
// 22. Template Method Pattern -- squelette
type WorkerInterface interface {
        Work() string
}

type TemplateWorker struct {
        provider WorkerInterface
}

func NewWorker(p WorkerInterface) *TemplateWorker { return &TemplateWorker{provider: p} }
func (t *TemplateWorker) Execute() string {
        return "Prep -> " + t.provider.Work() + " -> Clean"
}

type ConcreteWorker struct{}
func (w *ConcreteWorker) Work() string { return "Specific Task" }
```

Les 23 patterns en Go

// behavioral package pattern

```
// 23. Visitor Pattern -- pas de modification de la classe
type Visitor interface {
        VisitA(*ConcreteElementA)
}

type Element interface { Accept(Visitor) }

type ConcreteElementA struct{}
func (e *ConcreteElementA) Accept(v Visitor) { v.VisitA(e) }

type ConcreteVisitor struct{}
func (v *ConcreteVisitor) VisitA(e *ConcreteElementA) { fmt.Println("Visited Element A") }
```