Design patterns

References

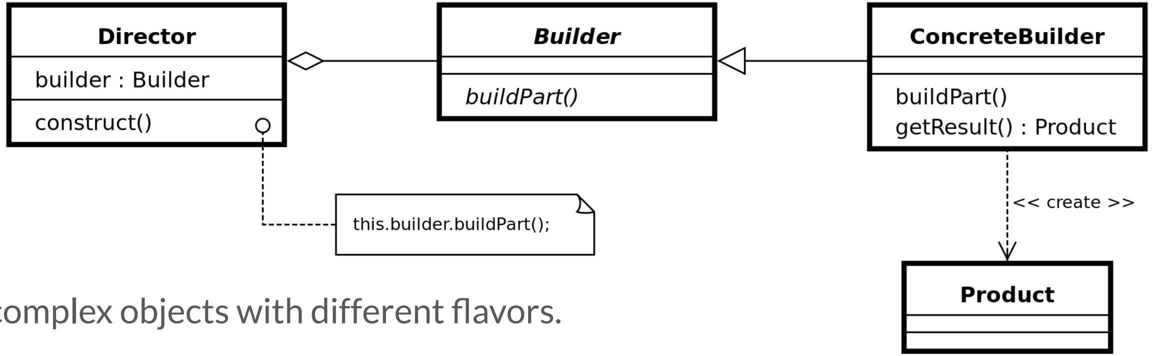
Design Patterns: Elements of Reusable Object-Oriented Software (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

https://github.com/tmrts/go-patterns

en.wikipedia.org

Creational patterns

Builder

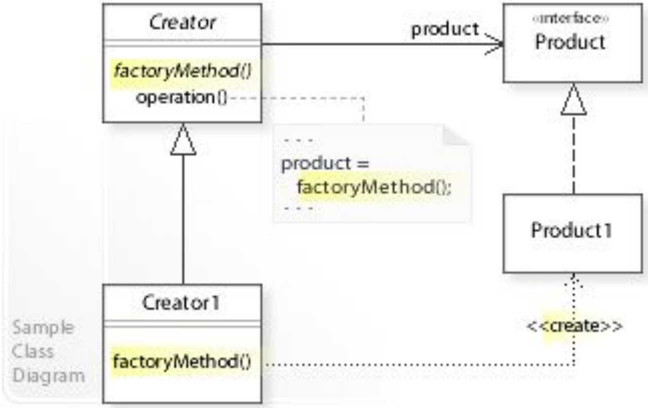


Promote the construction of complex objects with different flavors.

Use when:

- Use when you have objects that needs to be built in several steps and want to delay object finalisation until all steps are completed
- You have multiple flavors of a same object to construct.
- You want to decouple the object creation from its constructor.

Factory method



Decouple construction from the object itself.

Use when:

- You will only know the concrete type to instantiate at runtime.

Singleton

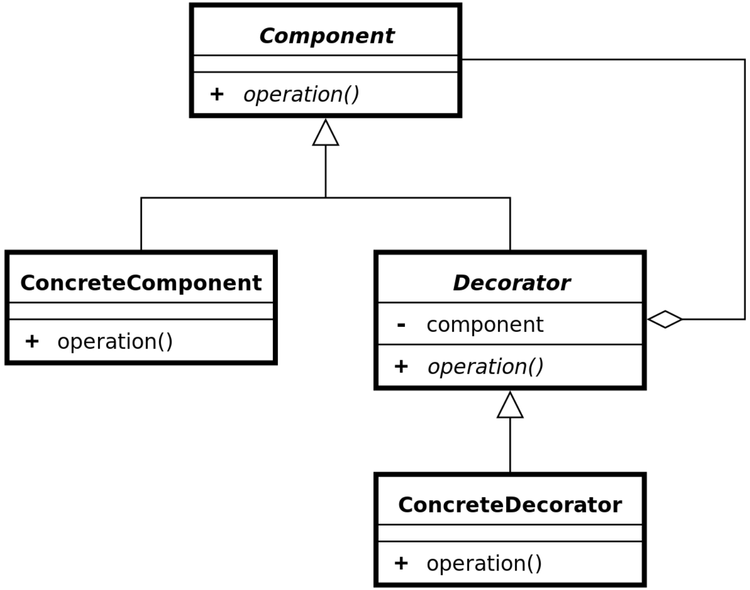
Limits the number of instance to 1.

Often consider to be an anti pattern!

Structural patterns

Decorator

Allows to dynamically add behavior to an object.

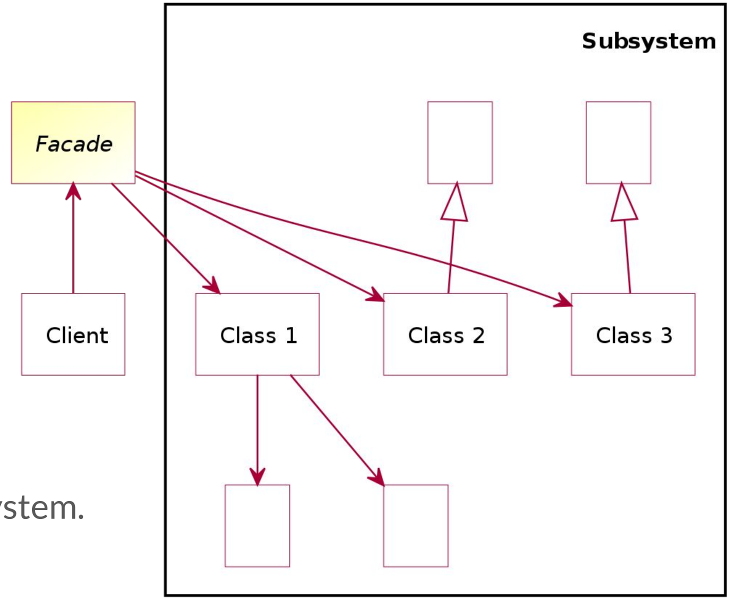


Facade

The Facade presents an interface to a subsystem.

Use when:

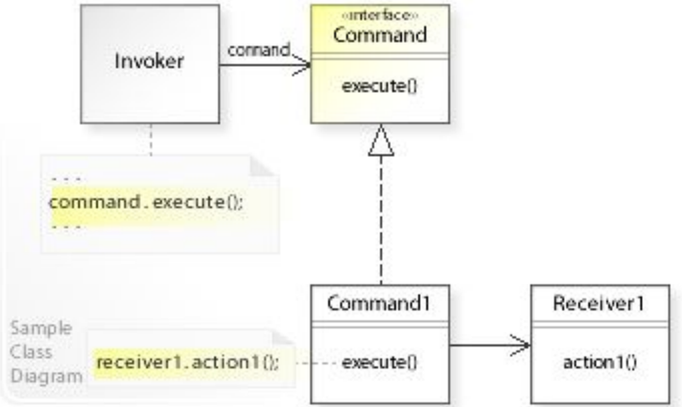
- You need to offer a simplified interface to a complex system.

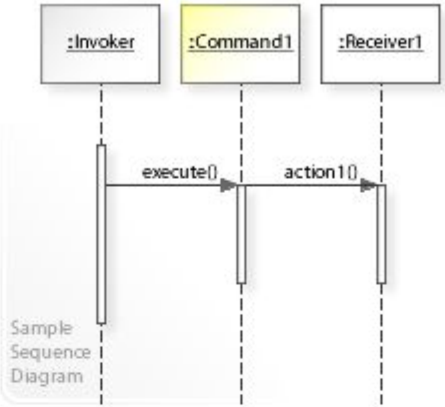


Sample class diagram

Behavioral patterns

Command





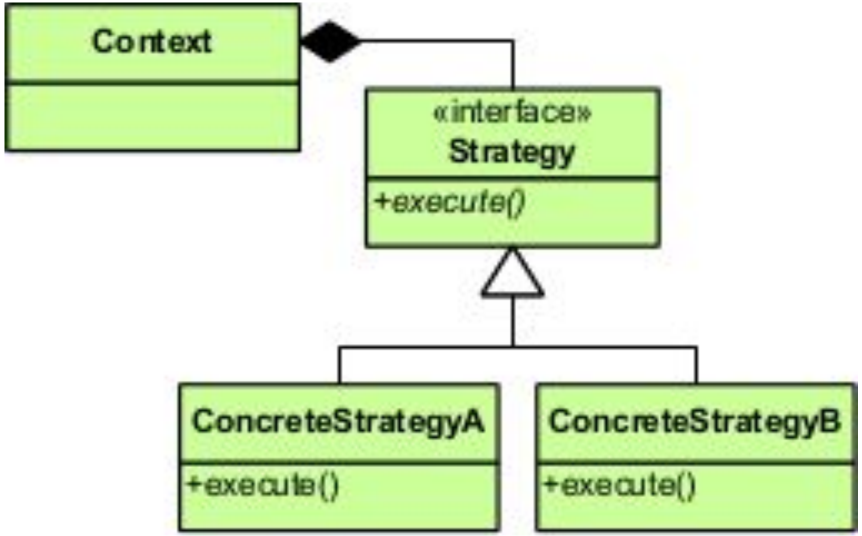
Bundles a command and an argument to be called later.

Use when:

- You need to prepare an "action" to be run later.

Strategy

Allow to select an algorithm at runtime.

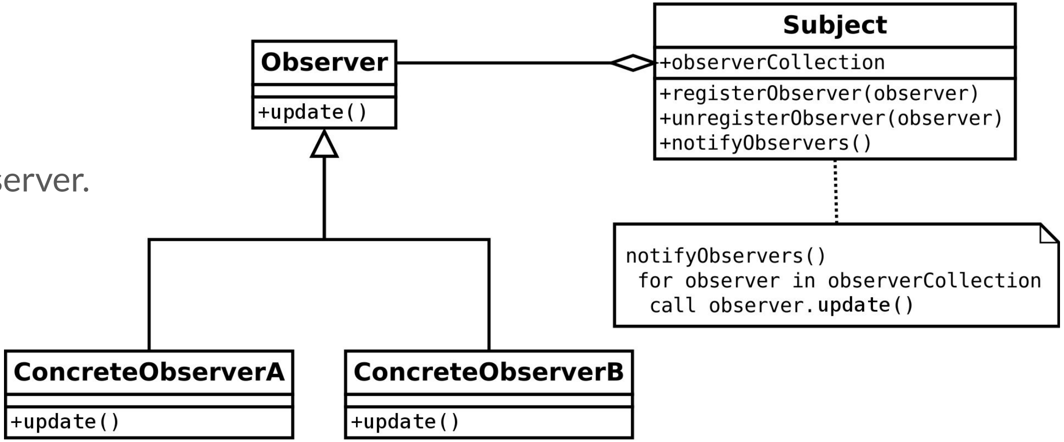


Observer

A “Subject” notifies registered “Observers” of certain events.

Use when:

- You need to react to some “events”.
- To decouple the subject and the observer.



Functional patterns

Lambda

Simply an anonymous functions.

Because sometimes you don’t need to put names on everything, including functions.

Typical use cases: short-live callbacks. (The observer patterns may then become obsolete)

Closure

A block of code (usually a function) that captures (part of) its surrounding scope (usually variables).

Use when you need to represent a process with its associated state.

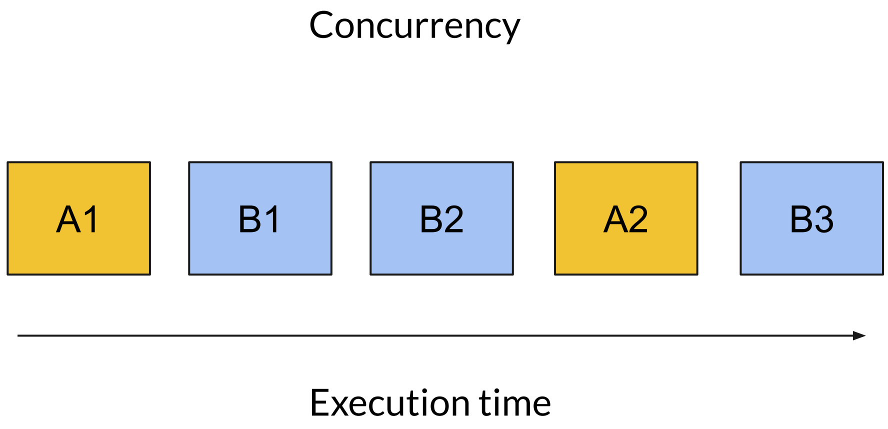
Memoization (with closure)

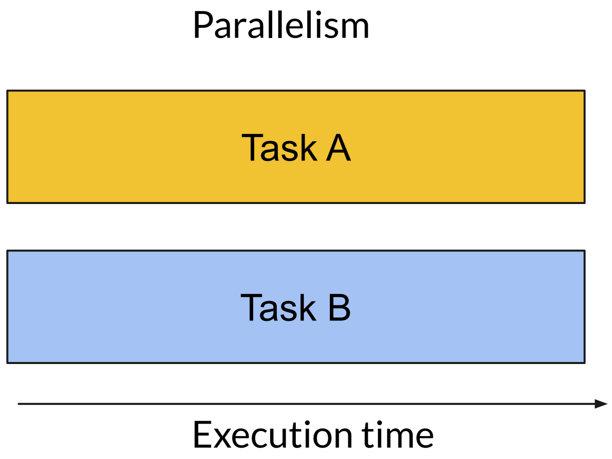
Remembering the result of an idempotent function.

Use on expensive function calls. May also be use on things that are not strictly idempotent (Http requests)

Concurrency patterns

Concurrency is not parallelism





Processes, threads and green threads

Terminology might get mixed up! (speaking of processes as a task for example)

- Processes are handled by the OS
- Executions is scheduled on physical threads by OS (hardware might intervene)
- Green threads are scheduled by the language runtime and mapped to OS threads

```
go someFunction()
```

The problem: Data Race

… and also deadlocks

Caused by:

- Asynchronicity of your architecture
- Modern CPU caching architecture
- Compiler optimizations

Memory Model

A memory model states what behaviour is guaranteed, or is not guaranteed considering sharing of data between multiple threads.

Memory Model : “Happens before relationship

If Thread1 states:

```
a = 1
```

```
b = 2
```

Thread2 might see:

```
b = 2
```

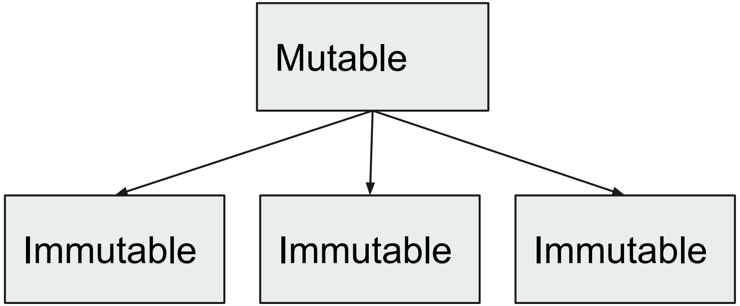
```
a = 1
```

The “Happens before” relationship is only guaranteed within the originating thread.

A property: Immutability

Data races originate from memory writing, aka modification in mutable entities.

Immutable entities are by nature thread safe.

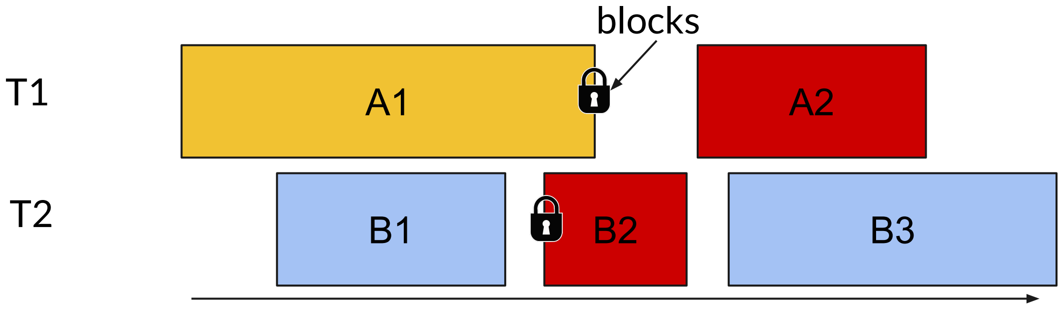


*Segregate immutability

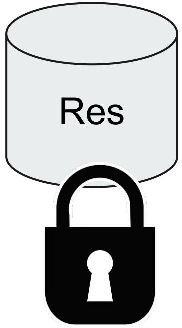
Mutex/Locks

A Mutex guards a critical section of your code.

Only 1 thread should execute the critical sections at once.



Execution time

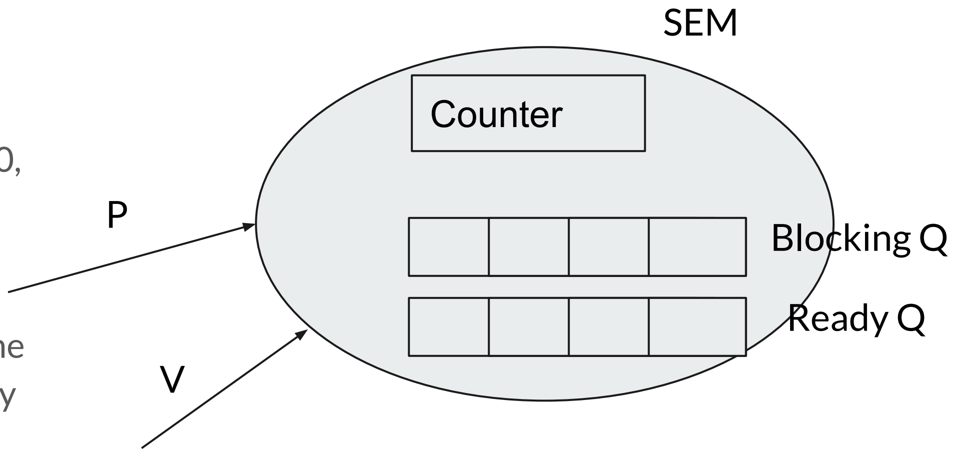


Semaphore/Signaling

Can be seen of a generalisation of Mutexes.

P (wait): decrement counter by 1. If counter < 0, blocks and is added to blocking queue.

V (signal): increments counter by 1. If pre-increment value was negative, transfers the first element in the blocking queue to the ready queue.



Producer-Consumer problem

One process emits items, another process consumes them. Items are put on a shared buffer queue.

- Consumer blocks if buffer queue is empty.
- Producer blocks if buffer queue is full.

```
produce:
    P(useQueue)
    putItemIntoQueue(item)
    V(useQueue)
```

```
consume:
    P(useQueue)
    item ← getItemFromQueue()
    V(useQueue)
```

Can scale to multiple Producer and Consumers

Communication Sequential Process (CSP)

- Design each task (process) in sequential execution.
- Data is **communicated** through channels. No shared states.

Channels

“Don’t **communicate** by **sharing memory**, **share memory** by **communicating**”

Build a channel:

`c := make(chan int)`

Send to a channel: `c <- 1`

Receive from a channel: `msg := <-c`

Buffered and unbuffered

Unbuffered

```
c := make(chan int)
```

Sender blocks until receiver consumes.

Receiver blocks until sender sends.

Buffered

```
c := make(chan int, 1)
```

Sender blocks if channel is full.

Receiver blocks if channel is empty.

Close channels

To signal receivers not to wait anymore. Most of the time, the sender should be responsible for closing.

```
    c := make(chan int)
    close(c)
    _, ok := <-c
    fmt.Println(ok) // Print false
```

Select

Analog to switch statement for channels operations.

Choses the first non-blocking case, or default.

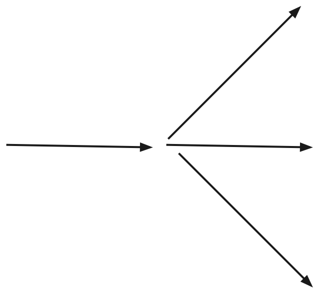
Avoid blocking on channels

```go
func tryReceive(c <-chan int) (data int, more, ok bool) {
    select {
    case data, more = <-c:
        return data, more, true
    default:
        return 0, true, false
    }
}
```

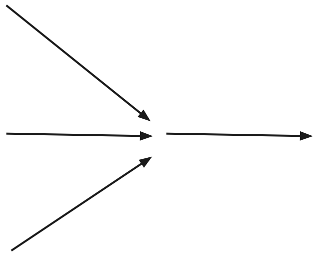
Avoid blocking on channels (timeout)

```go
func tryReceiveWithTimeout(c <-chan int, duration time.Duration) (data int, more, ok bool) {
    select {
    case data, more = <-c:
        return data, more, true
    case <-time.After(duration): // returns a channel
        return 0, true, false
    }
}
```

Shaping the execution flow



Fan-out



Fan-in

Fanout

```go
func fanOut(In <-chan int, OutA, OutB chan int) {
    for data := range In { // receives until closes
        select {
        case OutA <- data:
            //TODO
            break
        case OutB <- data:
            //TODO
            break
        }
    }
}
```

That’s it!