

DESIGN PATTERNS

Mubarak



Architecture Design vs Implementation Design



Quality

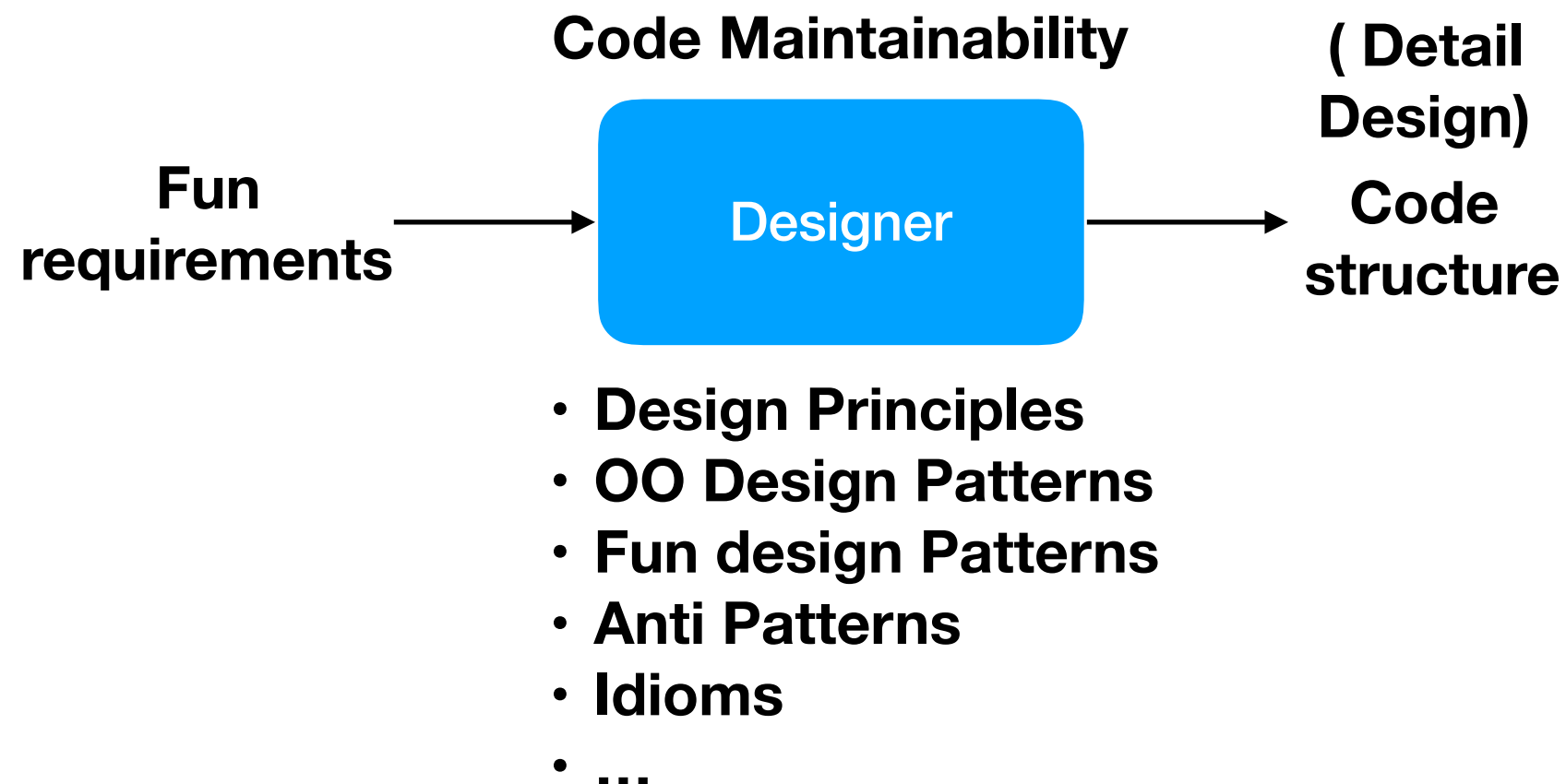
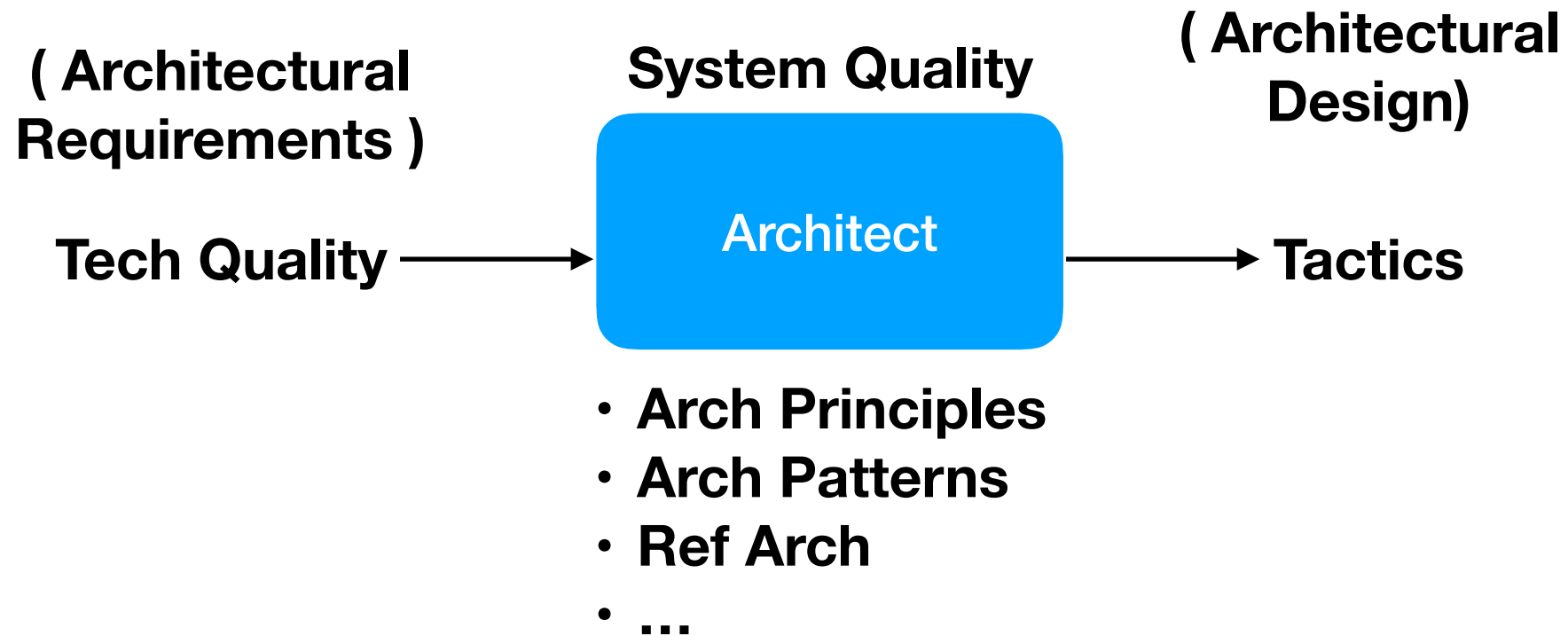
1. Cost
2. Time

Tech Quality

1. Performance (cpu, memory, I/O, ...)
2. Maintainability
3. Scalability (volume- cpu, memory, I/O, ...)
4. Security (Trustability)
5. Usability
6. Reliability (Trustability)
7. Availability
8. Robustness (Rugud)
9. Portability
10. Interoperability

Tactics

1. Reduce memory foot print
2. Extensible, readability, log, Testability
3. Authentication, Audit
4. ACID - Transaction
5. Input validation
6. Parallel
7. Caching
8. Lazy loading
- 9.



Java / py/ C++/ JS/

		Interface	Lamda
	Procedural	OO	Functional
Performance	n/a	n/a	3
Security	n/a	n/a	n/a
Testability	1	2	3
Manage code Complexity	1	3	2
Learning Curve	3	1	2
Time to develop	3	1	2
Immutability	No	No	Yes

OO => Manage Code Complexity

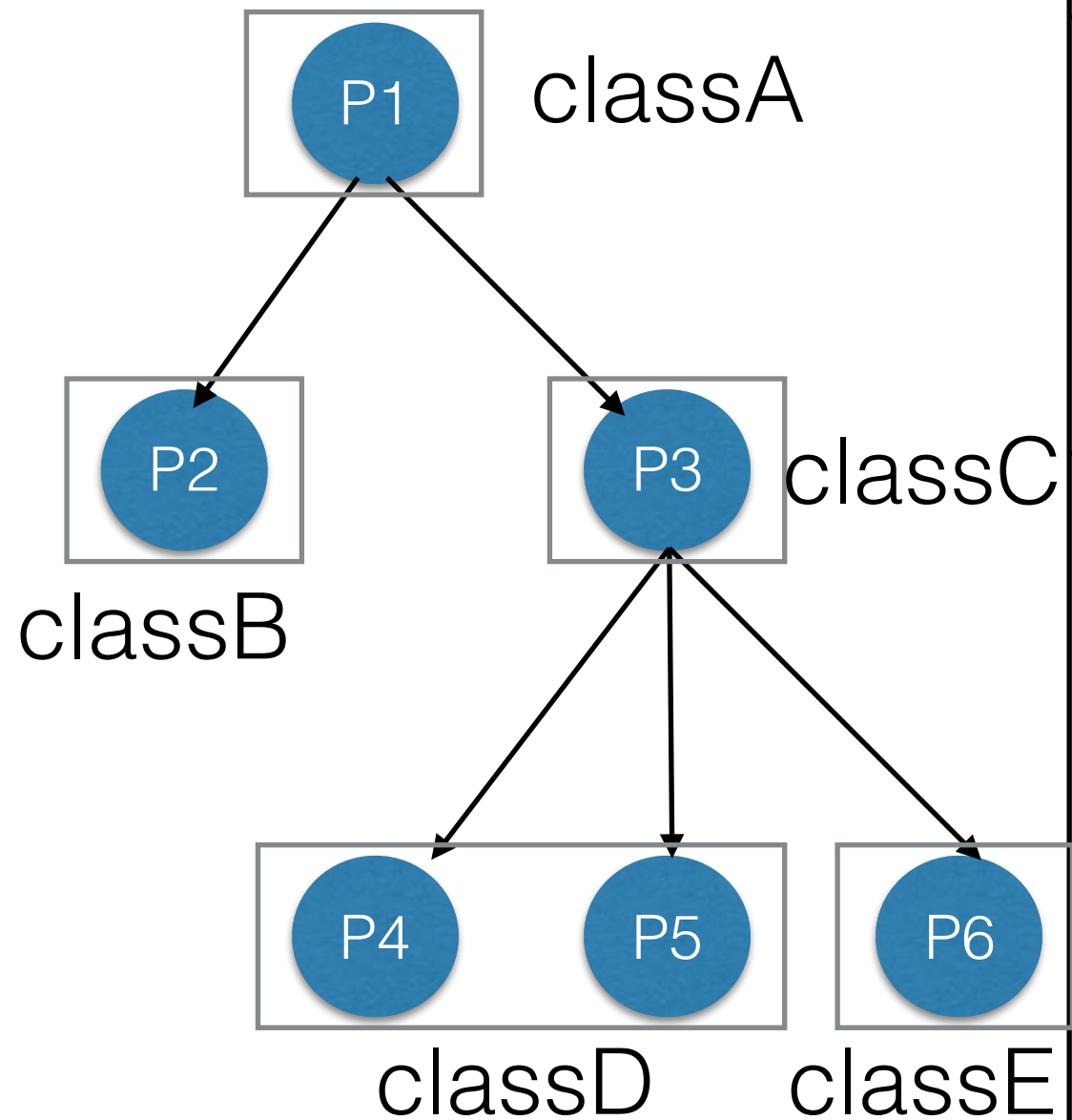
```
Interface Bird  
{  
    fly();  
    buildNest();  
    layEggs();  
    sing();  
}
```

```
Interface Bird  
{  
    eat()  
}
```

```
fun(Bird bird)  
{  
    //logic  
}
```

Procedural Prog

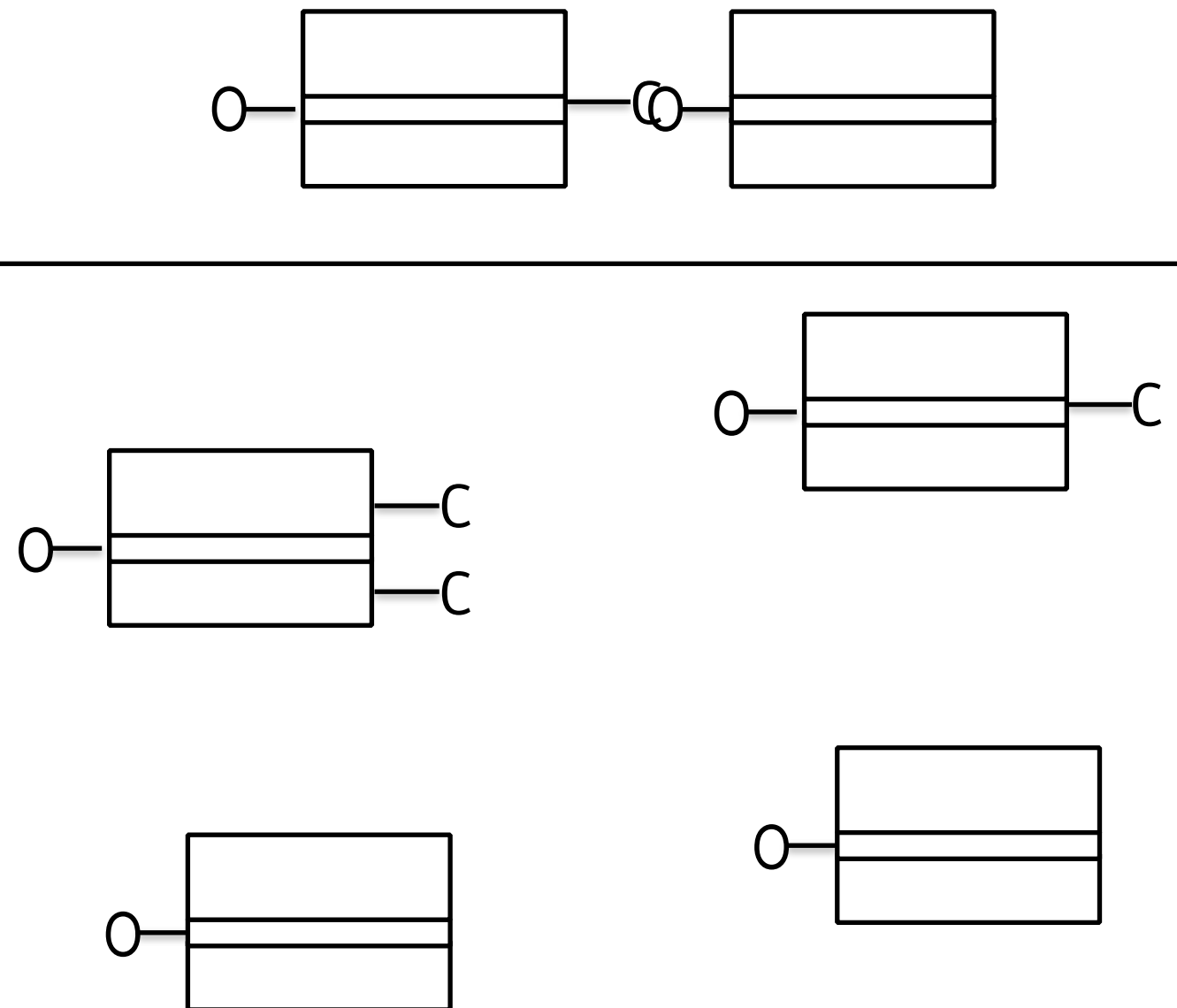
(tree)



(top down)

OO Prog

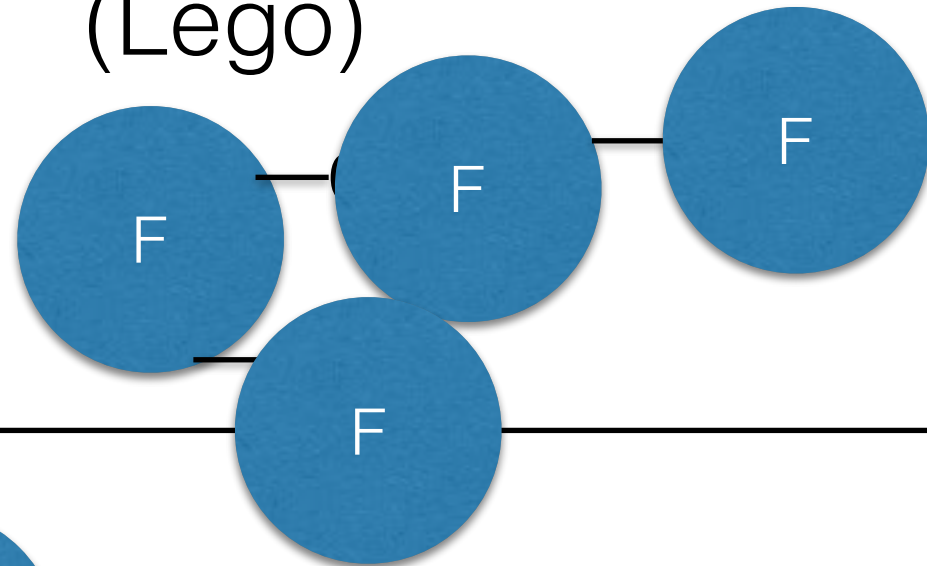
(Lego)



(bottom up)

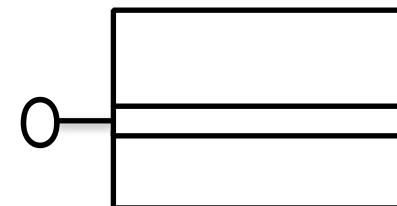
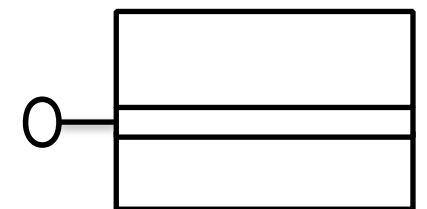
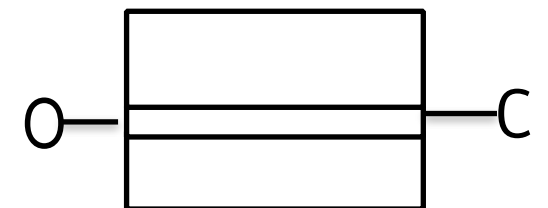
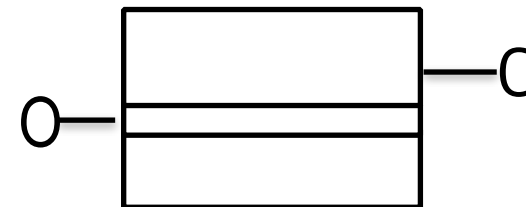
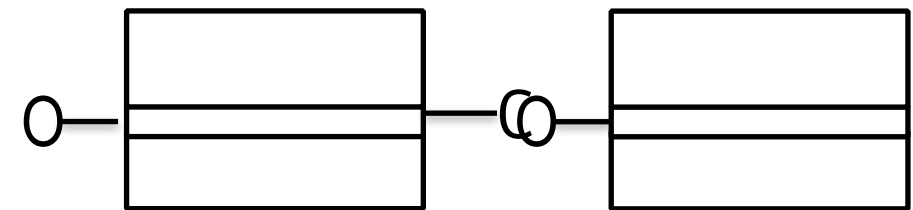
Functional Prog

(Lego)



OO Prog

(Lego)



==

If/switch ==> EH

Error

```
res = fun();  
if(res == true)  
{  
    ...  
}
```

==

If/switch ==> interface

Flow

```
Status = MakePayment();  
if(status == 1)  
{  
    ...  
}  
if(status == 2)  
{  
    ...  
}
```

< > <= >= ==

If/switch ==> ?

Domain rule

```
if( salary> 5000 && age < 32)  
{  
    ...  
}
```

obj.f1();

Method Call

coupling ==> interface typing

Coupling ==> function Objects

Coupling ==> duck typing

//*** interface**

interface Brid{

fly()

}

void do(Bird bird)

{

bird.fly();

}

//*** duck**

void do(bird)

{

bird.fly();

}

//*** lambda**

void do(fly)

{

fly();

}

new CA();

Instantiation

coupling ==> DI

coupling ==> factory

Abstraction

******* interface**

```
interface Bird{  
    fly()  
}  
void do(Bird bird)  
{  
    bird.fly();  
}
```

//*** duck**

```
void do(bird)  
{  
    bird.fly();  
}
```

//*** lambda**

```
void do(fly)  
{  
    fly();  
}
```

class Parrot implements Bird{
 public void fly(){

 }
}

do(new Parrot());

class Parrot {
 public void fly(){

 }
}

do(new Parrot());

class Parrot {
 public void flyHard(){

 }
}

do(()=> flyHard());

High order Functions

```
Lamda fun1(int x)
{
    z = x + 5;
    return (y)=> {
        return z+ y;
    };
}
```

```
Lamda fo1 = fun1(10);
Lamda fo2 = fun1(20);
```

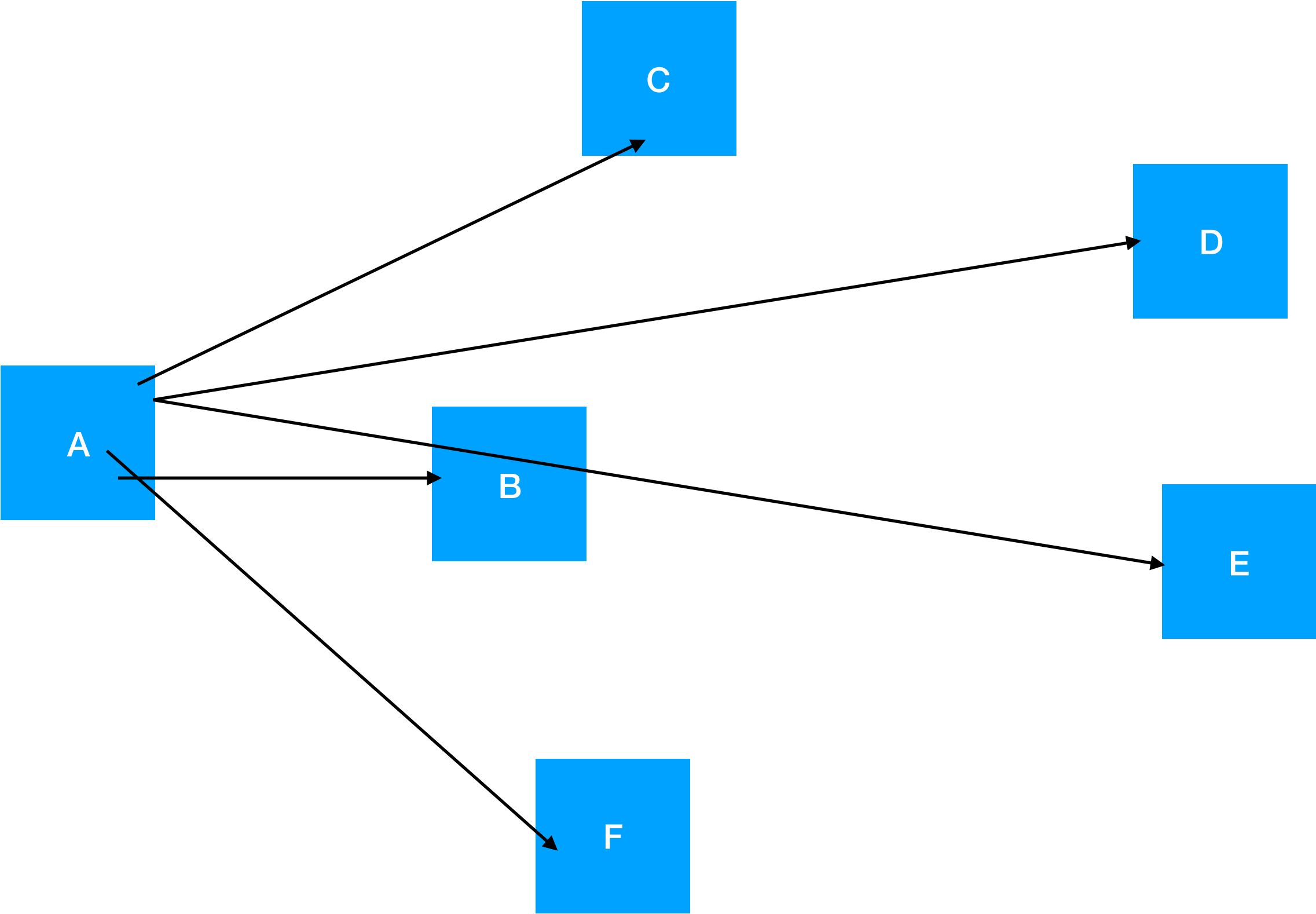
```
int i1 = fo1(5);
int i2 = fo2(5);
```

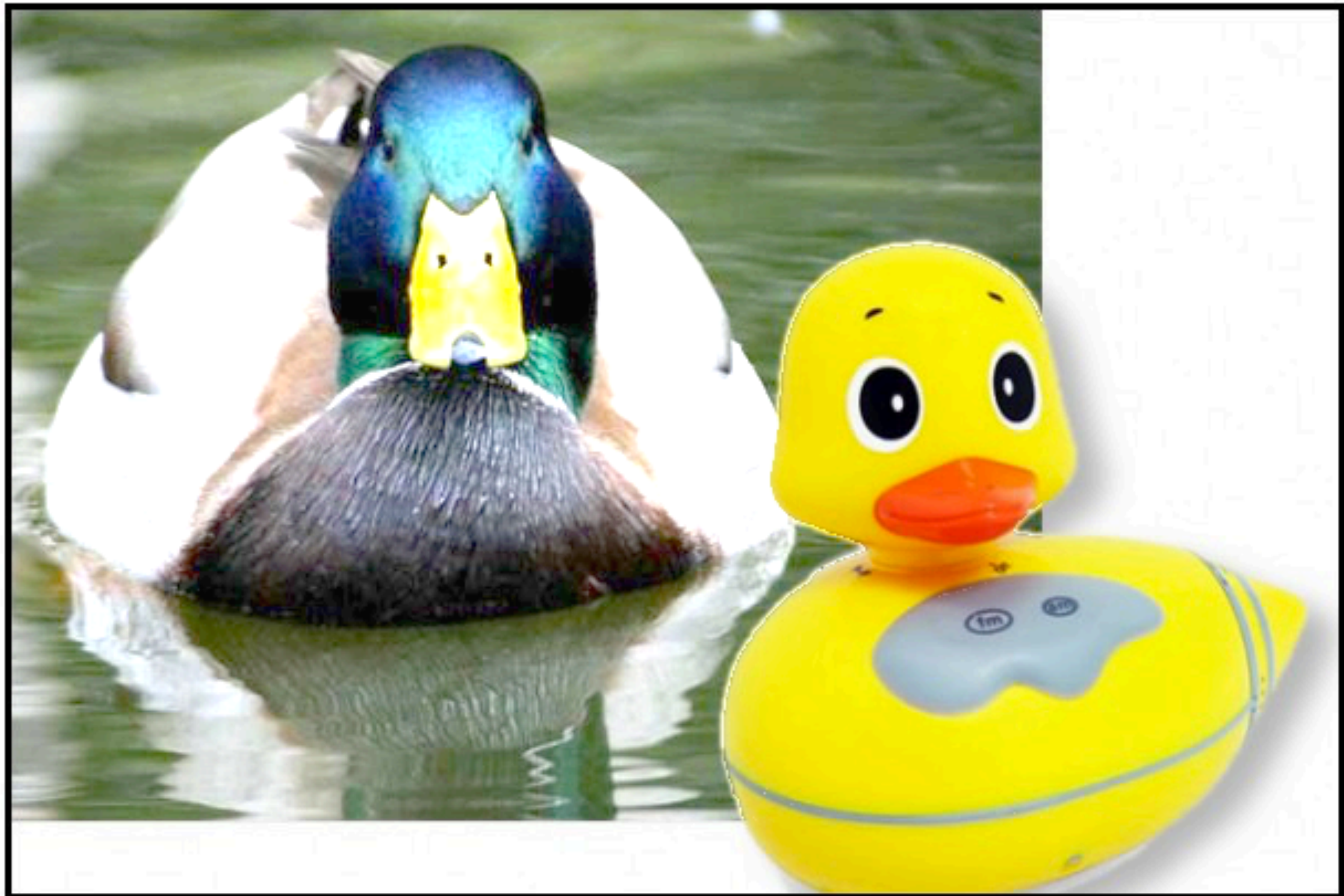
No variables
Only constants
No for
No while
No do

- for vs foreach
- $a+b$ - 3 cpu cycles
- Create thread - 200,000 cpu cycles
- Destroy thread - 100,000 cpu cycles
- I/O operations
- Exe Db command - 45,00,000 cpu cycles
-

Design Check list

- + LSP
- + SRP (*)
 - # things which don't change together
 - #fun size
 - \$ Avg: 5 loc
 - \$ Max: fit screen
 - #class size
 - \$ Avg: 5 interface methods
 - \$ Max: 12
- + Low Coupling (*)
- + Exceptions
- + DRY (*)
- + DIP
- + OCP (open for add, closed for change)
- + Program to an Interface
- + Cyclomatic Complexity < 10
- Flag
- Throws NotImplemented
- bool/null/int for error handling
- Static Methods
- Swiss Knife/ God Class
(Util,Controller, Helper, Provider, Handler,Activity, Manager, Processor, Module, ...)
- Functional Interface
- default methods
- Runtime Type Identification





LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.


```
1 class Repeat
2   def print_message
3     puts "I Will Not Repeat My Code"
4     puts "I Will Not Repeat My Code"
5     puts "I Will Not Repeat My Code"
6     puts "I Will Not Repeat My Code"
7     puts "I Will Not Repeat My Code"
8     puts "I Will Not Repeat My Code"
9     puts "I Will Not Repeat My Code"
10  end
11 end
```

Software Engineering v/s Tuning



Quality

**# Performance
Engineering**

Threat Modeling

**# Performance
Tuning**

Ethical hacking

