Mubarak

Architecture Design vs Implementation Design


Quality
Excellent
Good
Average
Poor

Quality

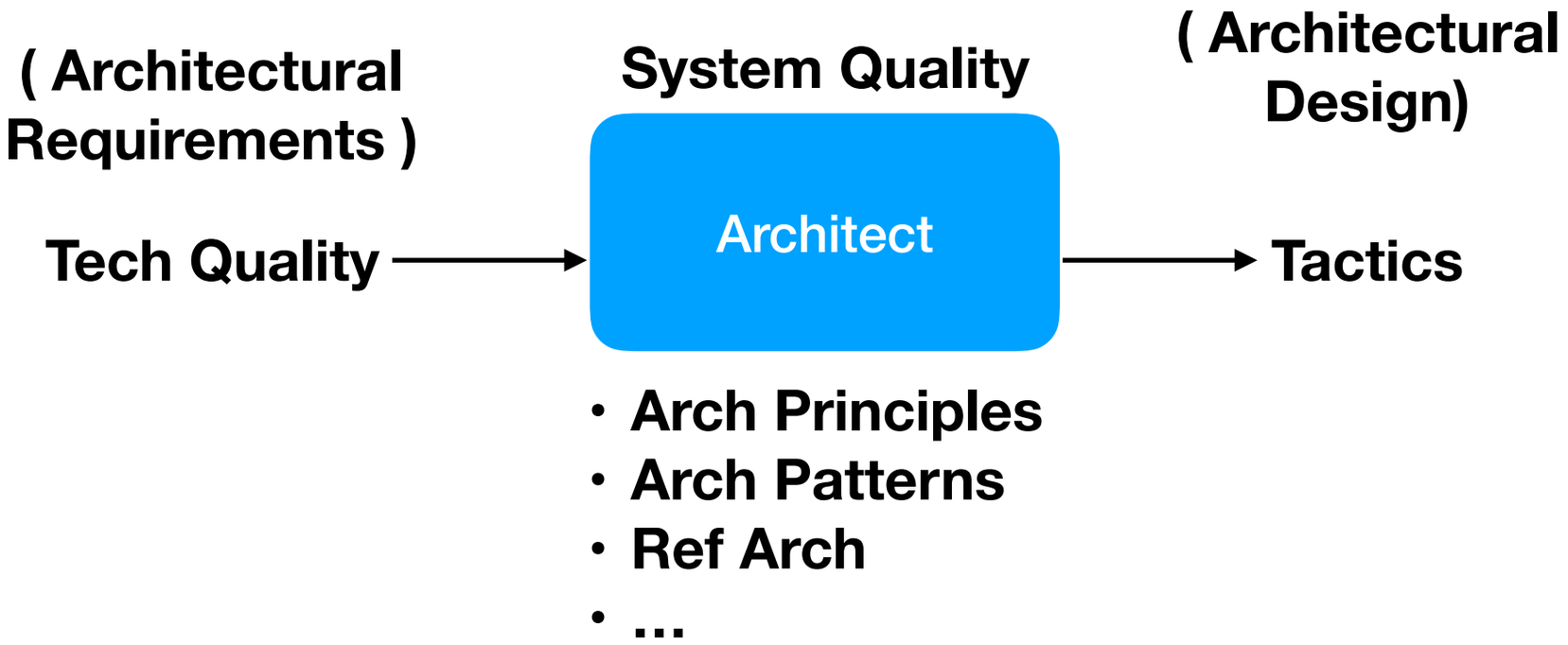
1. Cost
2. Time

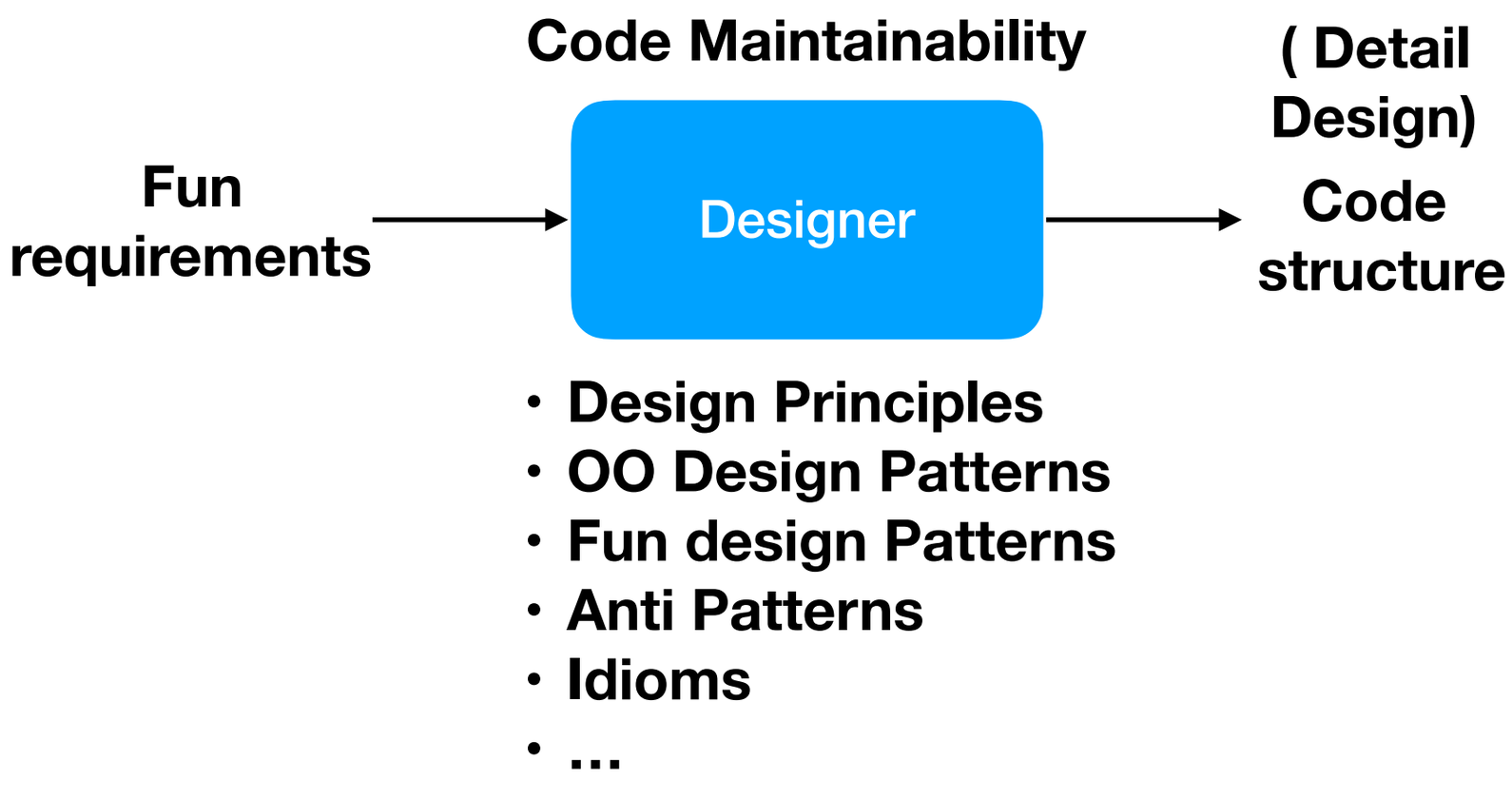
Tech Quality

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

Tactics

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


(Architectural Requirements)
System Quality
(Architectural Design)
Tech Quality
Architect
Tactics
• Arch Principles
• Arch Patterns
• Ref Arch
• ...


Code Maintainability
(Detail Design)
Fun requirements
Designer
Code structure
• Design Principles
• OO Design Patterns
• Fun design Patterns
• Anti Patterns
• Idioms
• ...

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

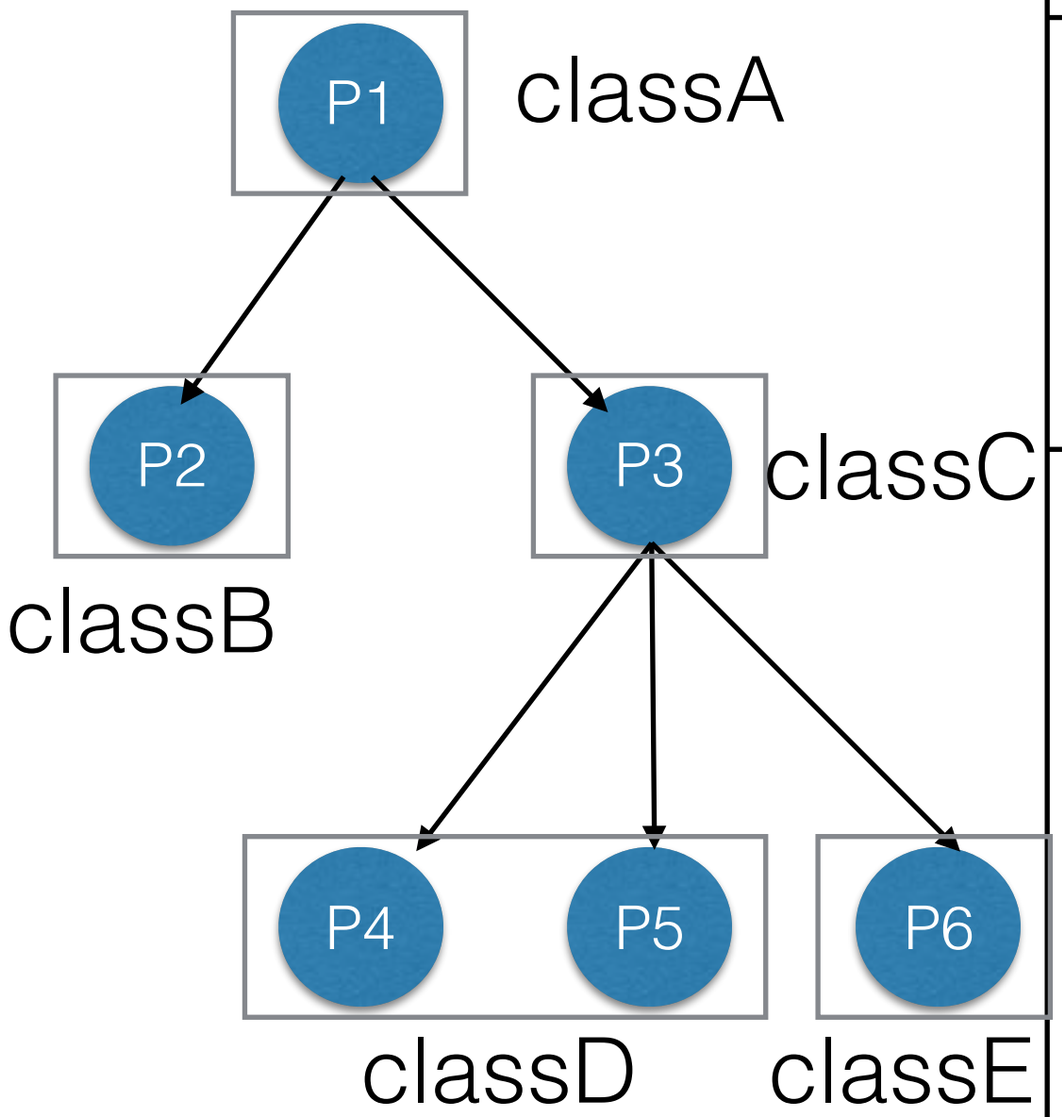
```
Interface  Bird
{
   fly();
   buildNest();
   layEggs();
   sing();
}
```

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

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

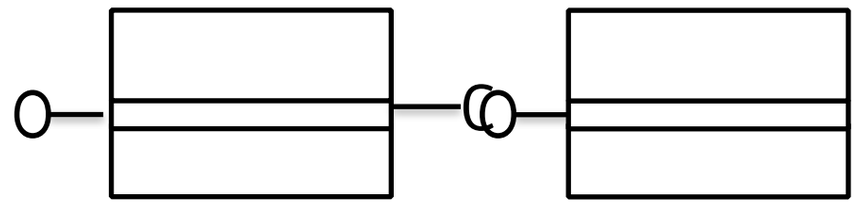
Procedural Prog

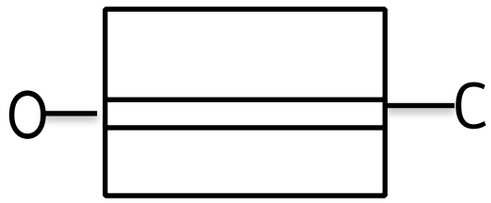
(tree)

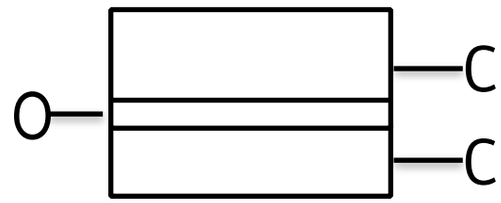


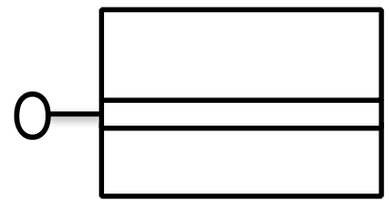
OO Prog

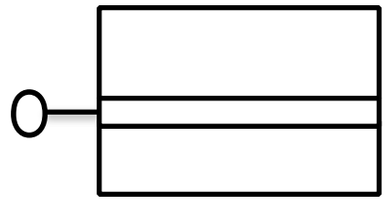
(Lego)



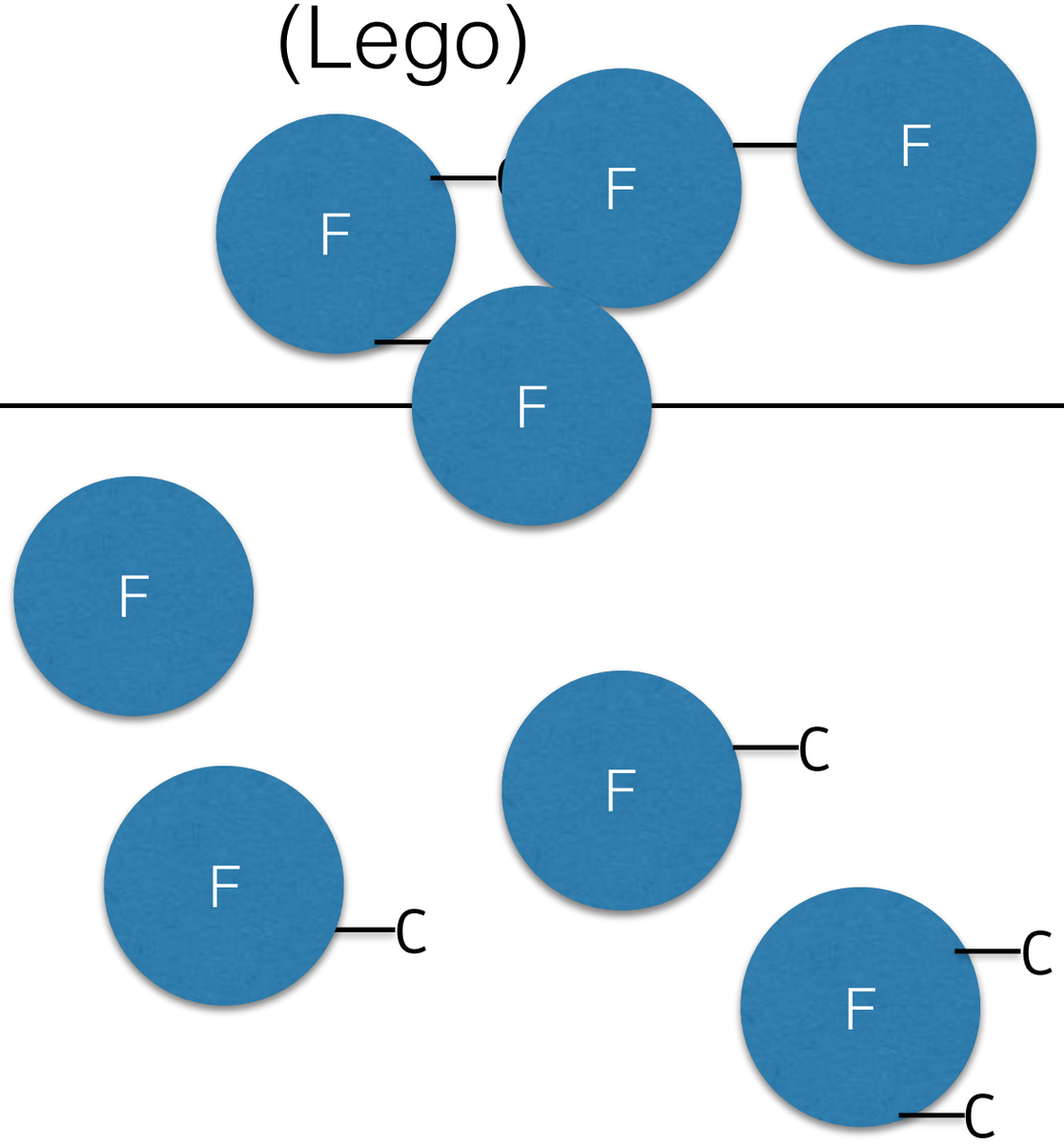




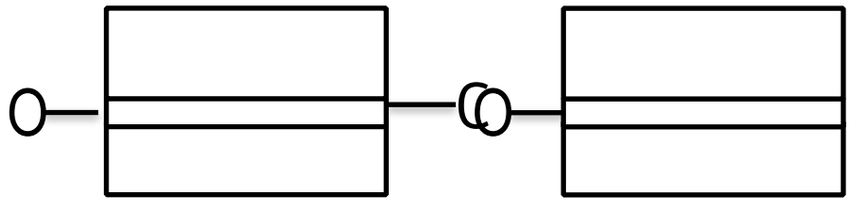


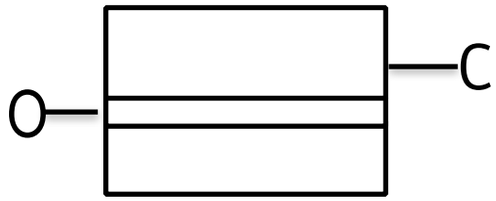


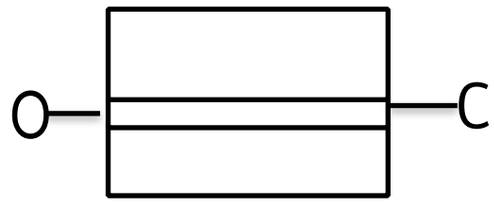
Functional Prog (Lego)

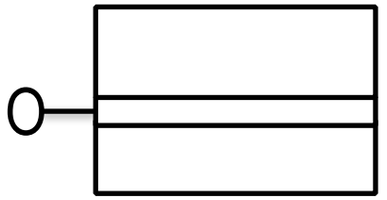


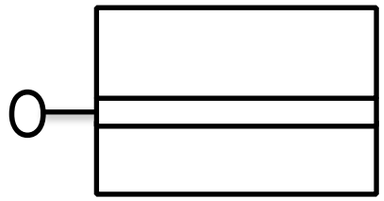
OO Prog (Lego)











==

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

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);
```

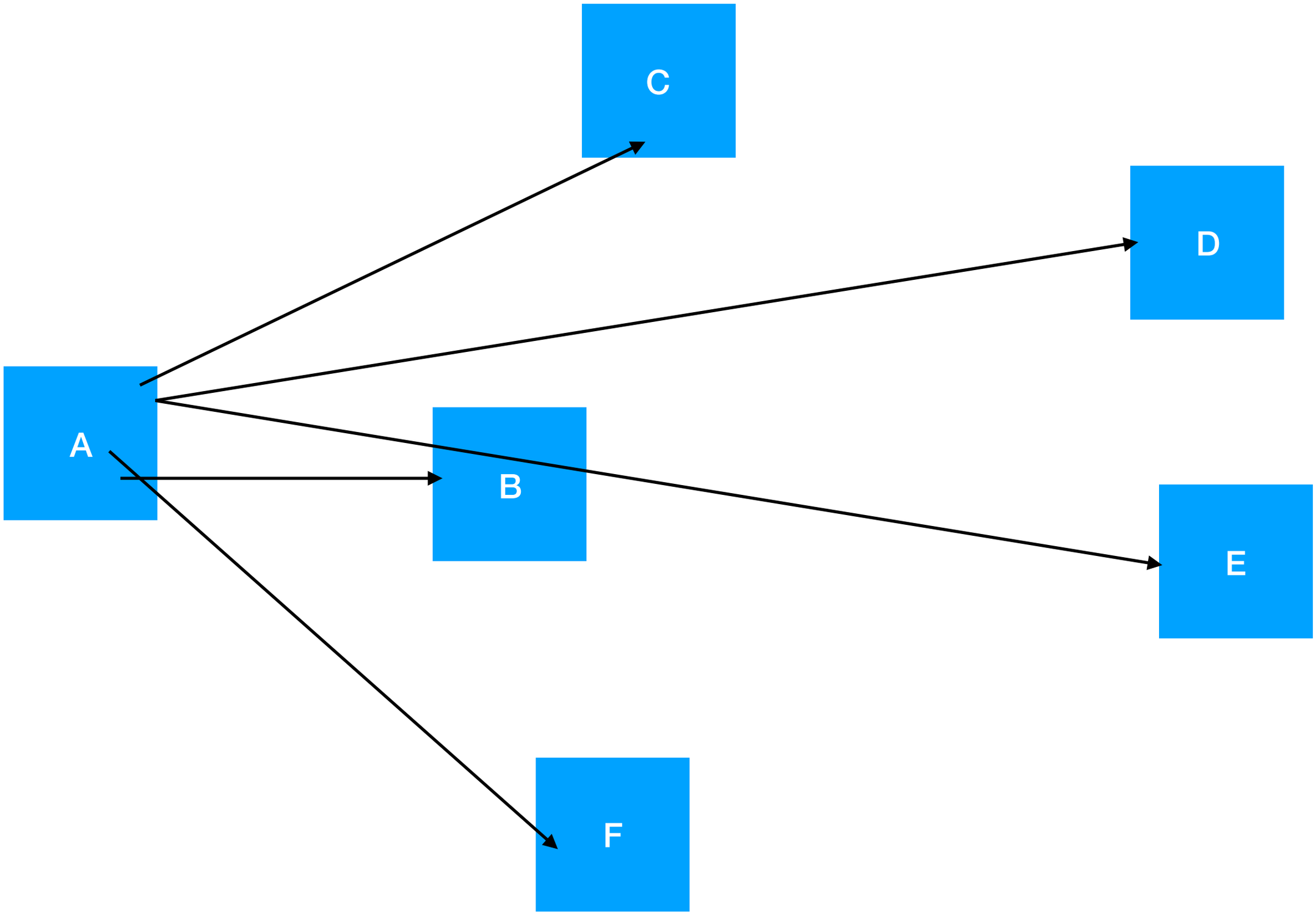
No variables
Only constants
No for
No while
No do

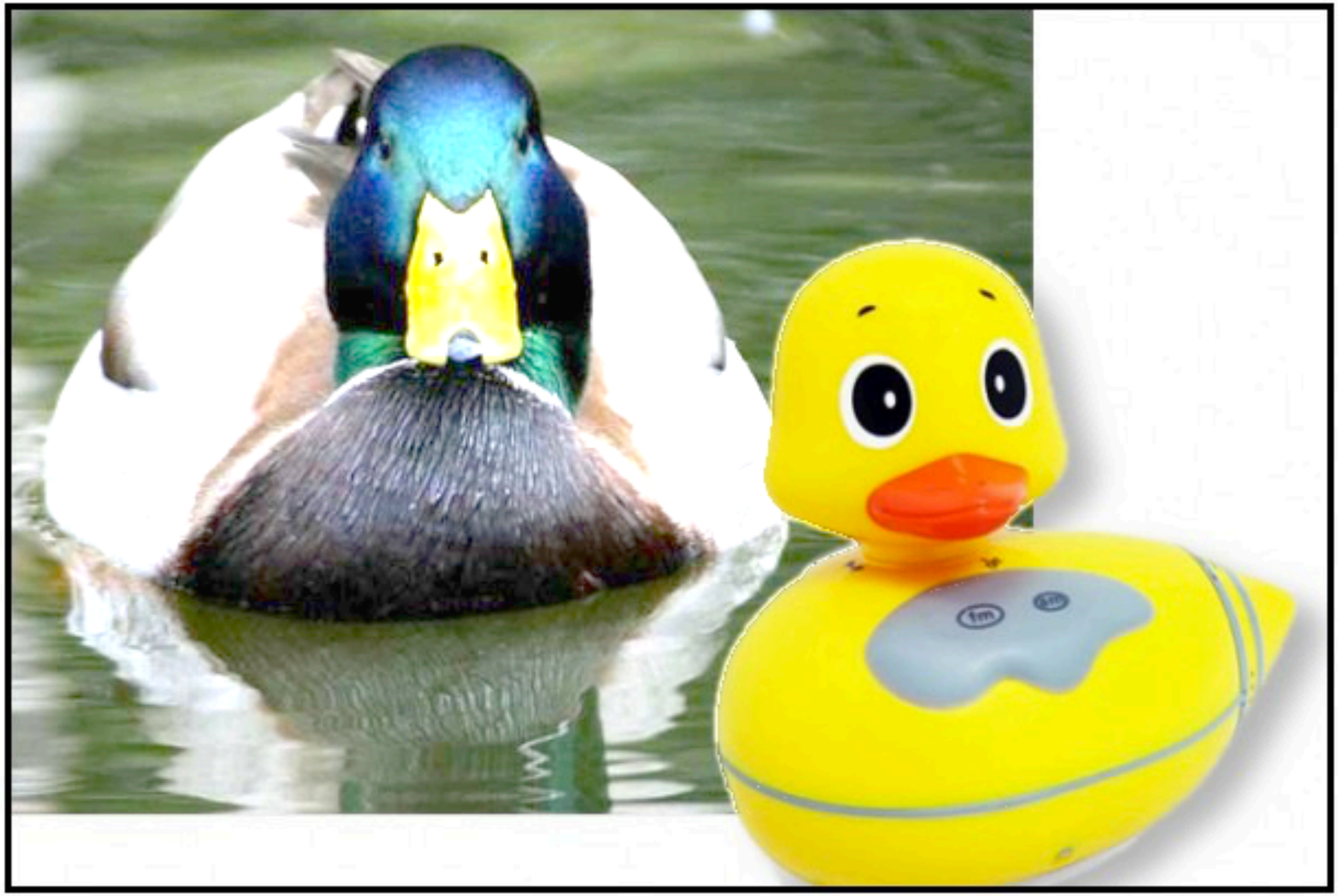
- for vs foreach
- a+b - 3 cpu cycles
- Create thread - 200,000 cpu cycles
- Destry thread - 100,000 cpu cycles
- I/O operations
- Exe Db command - 45,00,000 cpu cycles
-

Design Check list

+ LSP
+ SRP (*)
    # things which don’t change together
    #class size
        $ Avg: 5 interface methods
        $ Max: 12
+ Low Coupling (*)
+ Exceptions
+ DRY (*)
+ Program to an Interface

- 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


C
D
A
B
E
F



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.

```ruby
class Repeat
  def print_message
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
    puts "I Will Not Repeat My Code"
  end
end
```

Software Engineering v/s Tuning

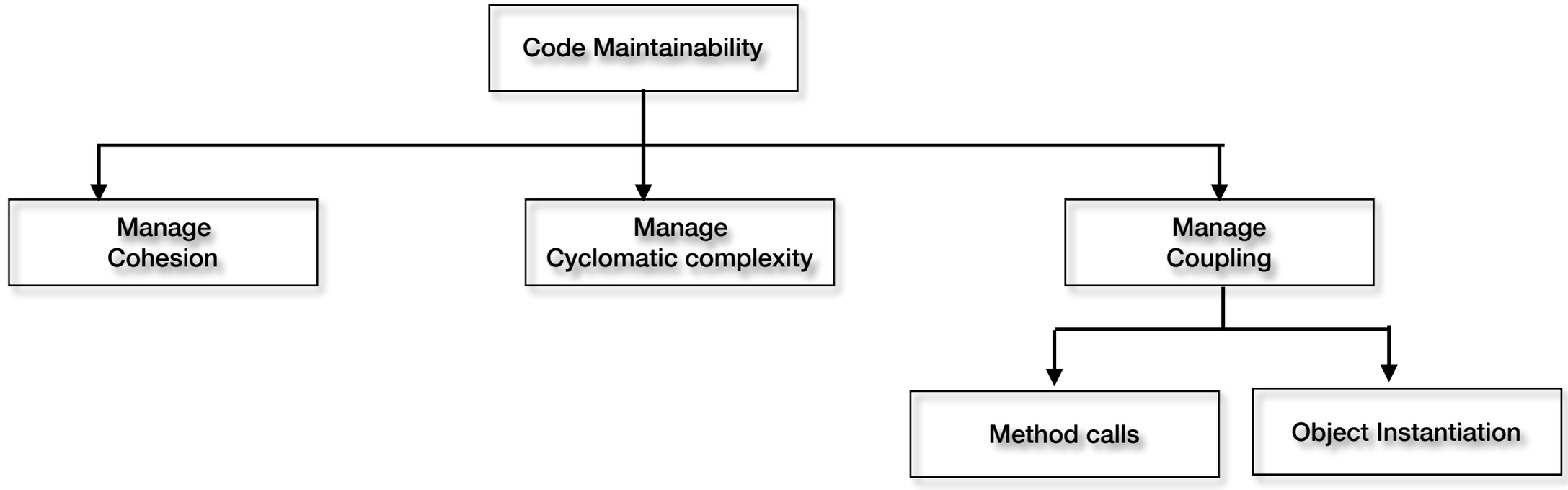
Quality

 # Performance Engineering

 # Threat Modeling

 # Performance Tuning

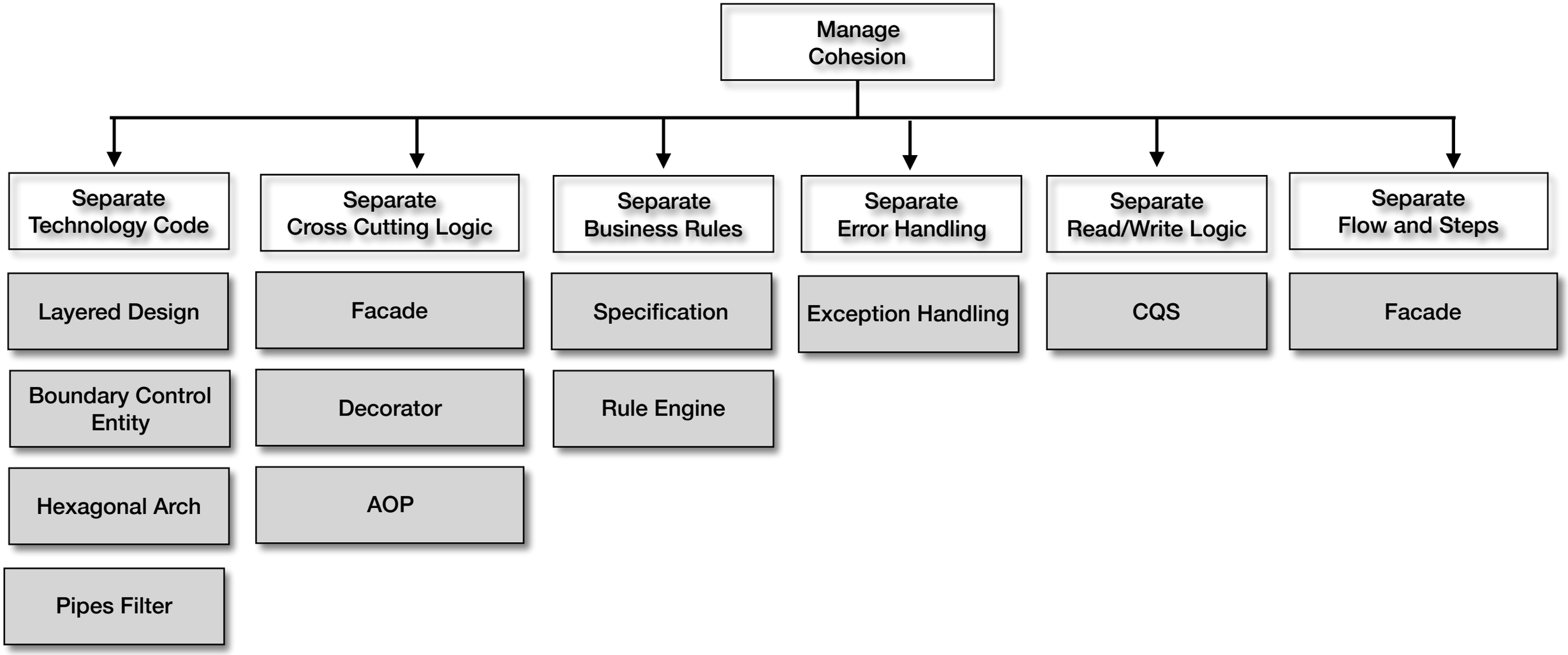
 # Ethical hacking


Code Maintainability
Manage
Cohesion
Manage
Cyclomatic complexity
Manage
Coupling
Method calls
Object Instantiation

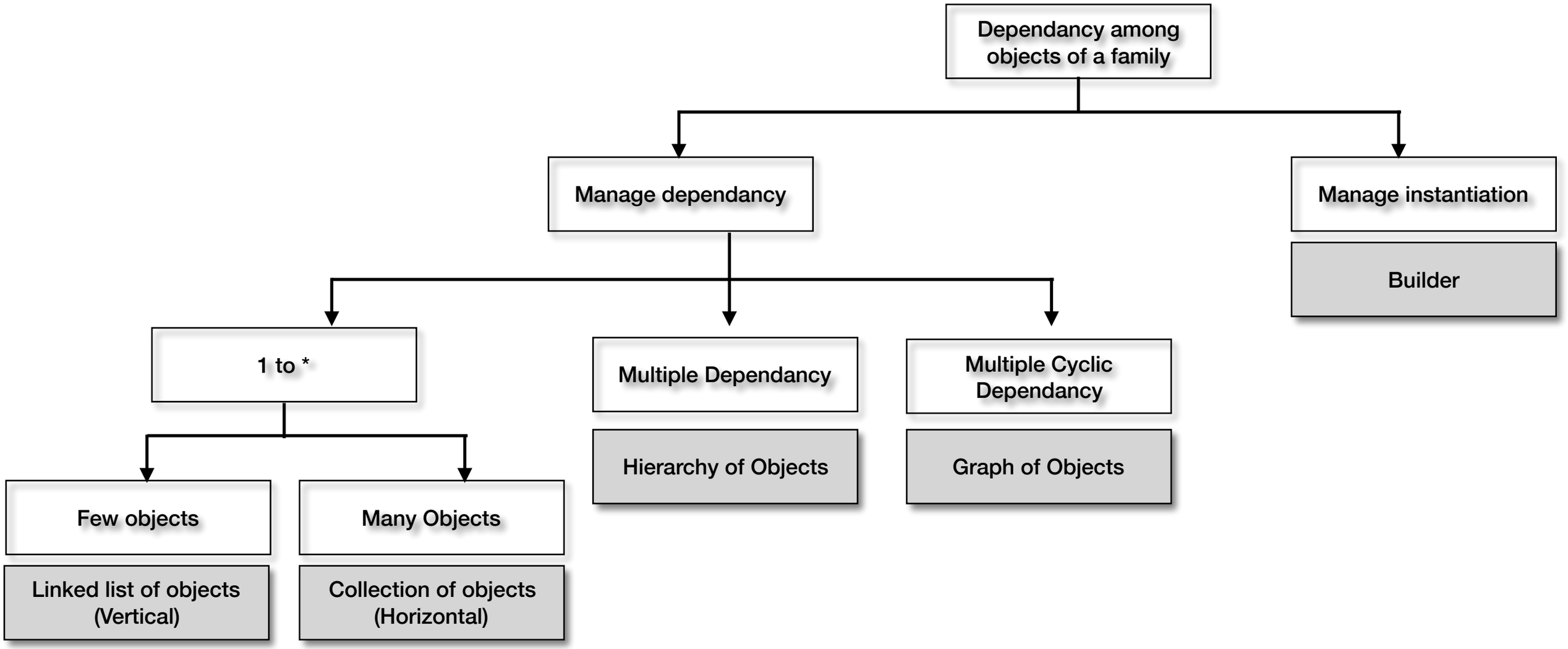
Manage Cyclomatic complexity

- Error Handling Flow
  - Exception Handling
- Alternate Code Flow
  - Only Data changes In Paths
    - Single class, Object per Path
  - Logic changes In Paths
    - Single dispatch
      - Interface
        - Functional Interface
          - Function Object
        - Breaks SRP
          - Delegated Interface
        - Breaks coupling
          - Visitor
    - dual dispatch
      - 2 objects belonging to different families
        - Visitor
      - 2 objects of same family
        - Use Lookup
    - multi dispatch
      - Decision Table
- Business Rules Flow
  - Specification

1. Convert the flag/enum/ type check to a Interface
2. Add a method for every use of flag/enum
3. Convert each flag/enum value to a interface implementation


Manage Cohesion
Separate Technology Code
Separate Cross Cutting Logic
Separate Business Rules
Separate Error Handling
Separate Read/Write Logic
Separate Flow and Steps
Layered Design
Facade
Specification
Exception Handling
CQS
Facade
Boundary Control Entity
Decorator
Rule Engine
Hexagonal Arch
AOP
Pipes Filter

- Manage Coupling
  - Tightly coupled Method call
    - Heterogeneous Interface
      - Adapter
    - Homogenous Interface
      - Static Language
        - Interface Typing
      - Dynamic Language
        - Duck Typing
      - Single Method
        - Function Object
    - * to *
      - Mediator
  - Tightly Coupled Instantiation
    - Delegate Responsibility
      - Dependancy Injection
    - Control instantiation
      - Class factory
        - Flags cannot removed (comes from boundary)
          - Factory with Lookup
        - Factory contains flag/enum
          - Factory interface


Dependancy among objects of a family
Manage dependancy
Manage instantiation
Builder
1 to *
Multiple Dependancy
Hierarchy of Objects
Multiple Cyclic Dependancy
Graph of Objects
Few objects
Many Objects
Linked list of objects (Vertical)
Collection of objects (Horizontal)