Mubarak

Architecture Design vs Implementation Design

Quality
☑ Excellent
□ Good
□ Averag
□ Poor

Quality

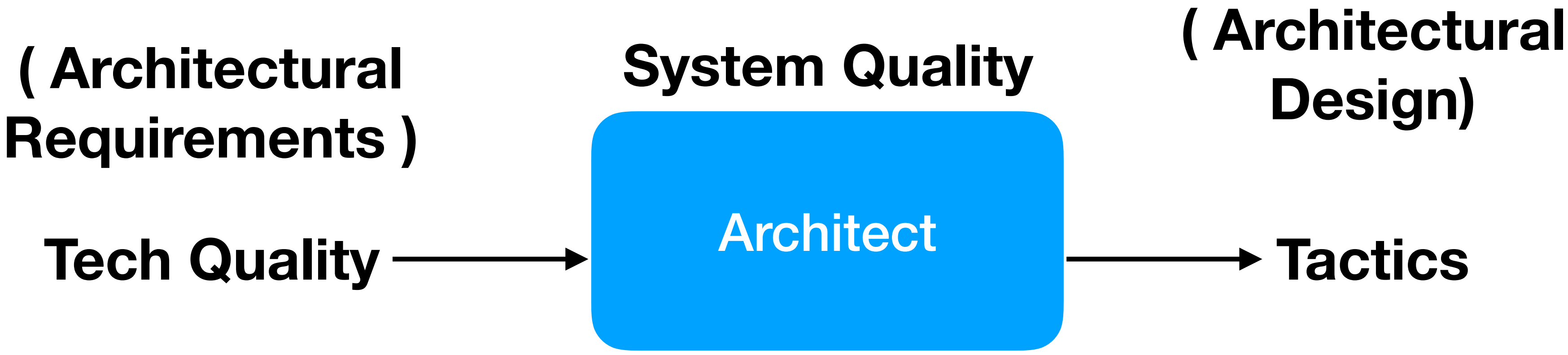
1. Cost
2. Time

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)

Tech Quality

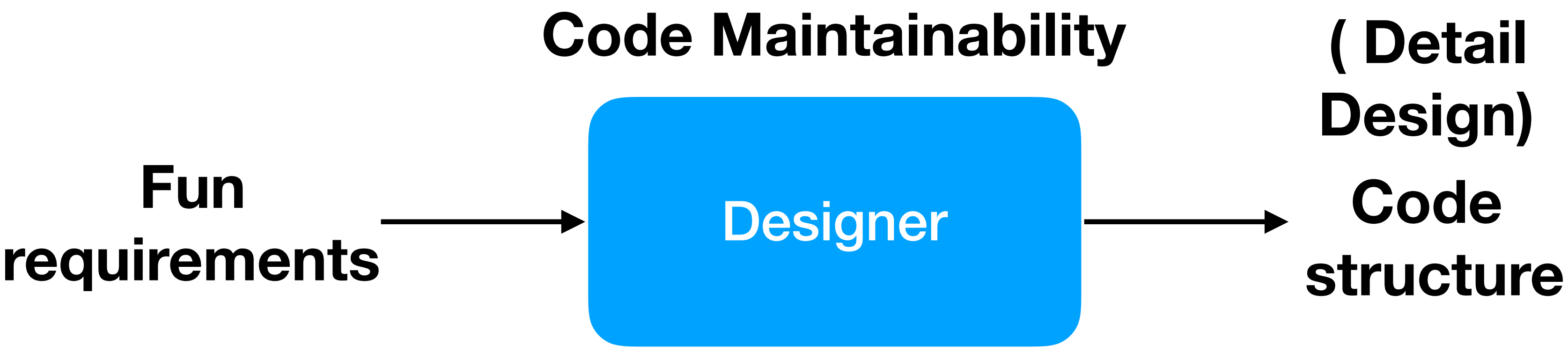
System Quality

Architect

(Architectural Design)

Tactics

- Arch Principles
- Arch Patterns
- Ref Arch
- …



Code Maintainability

Fun requirements

Designer

(Detail Design)

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()
}
```

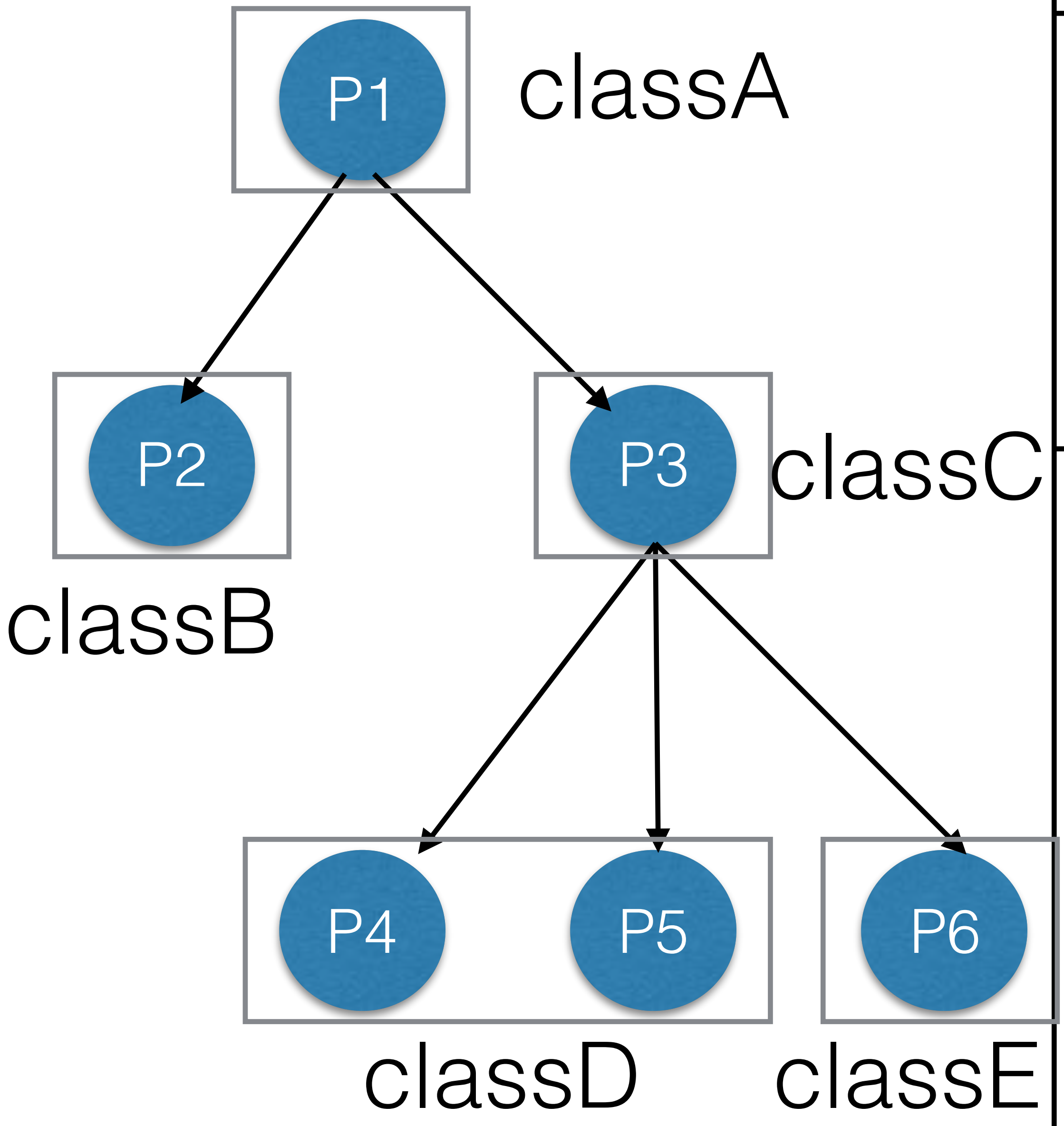
```
fun(Bird bird)
{
   //logic
}
```

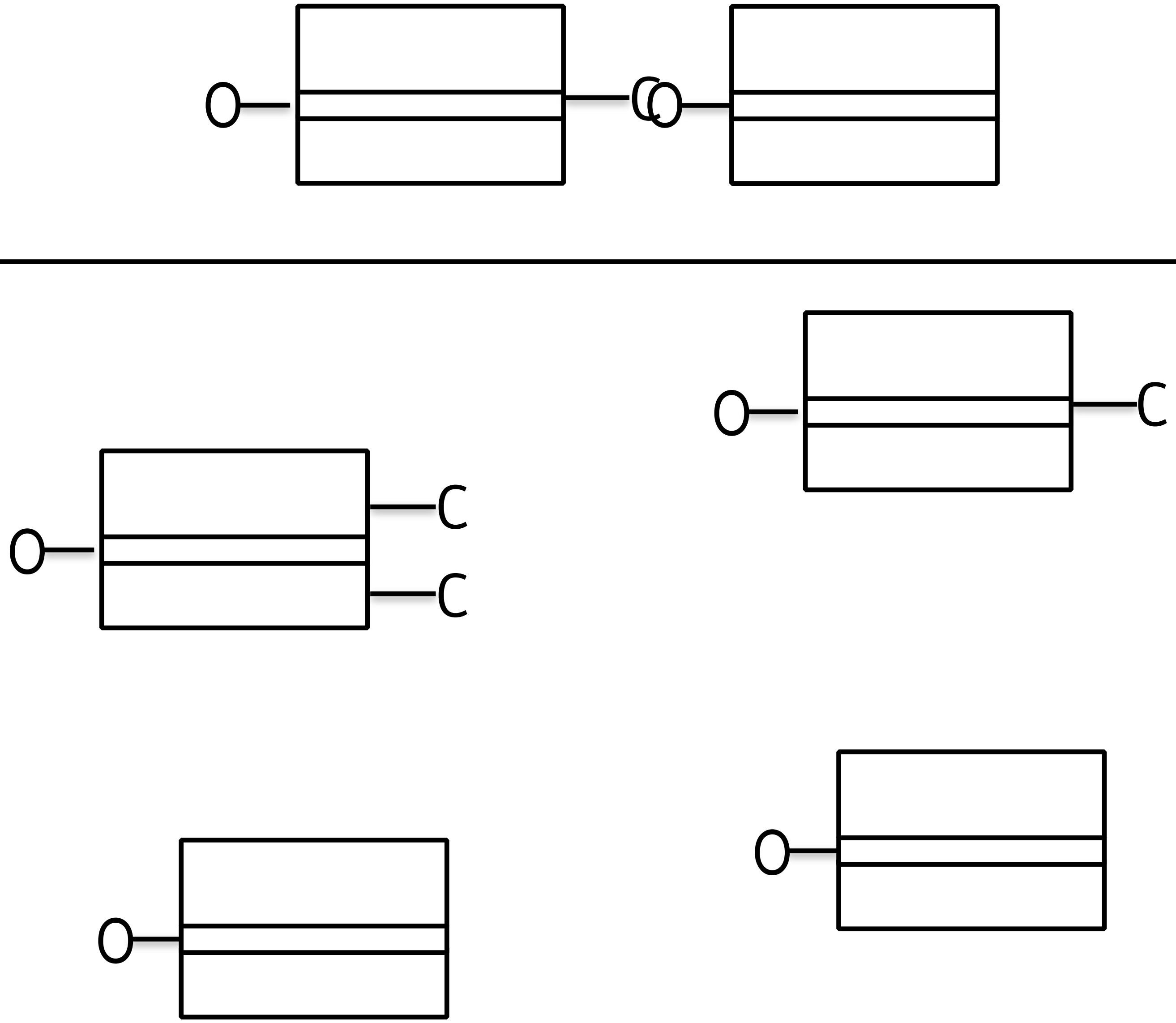
Procedural Prog

(tree)

OO Prog

(Lego)



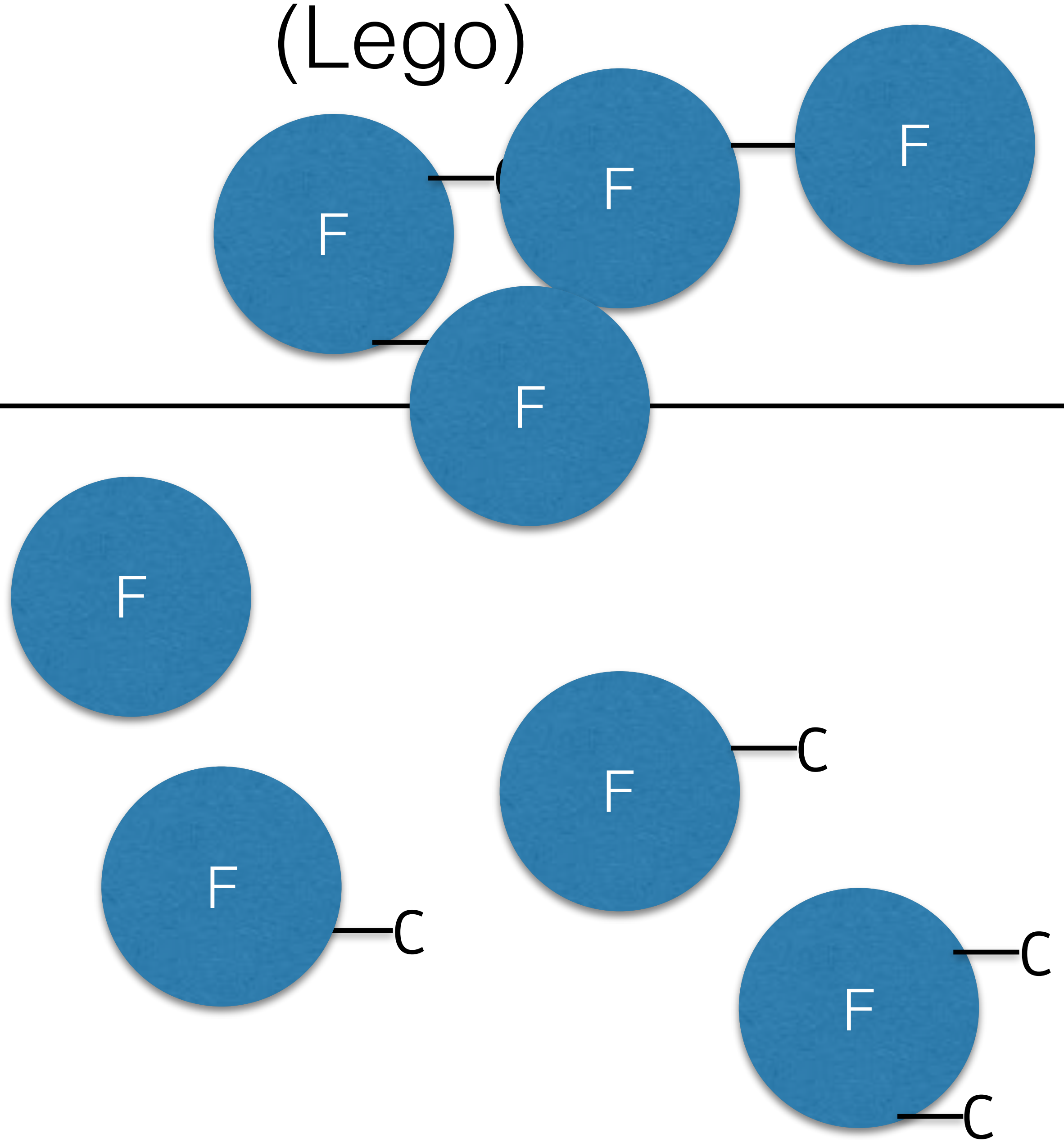


(top down)

(bottom up)

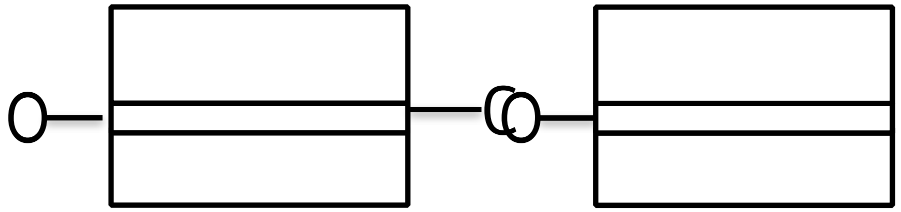
Functional Prog

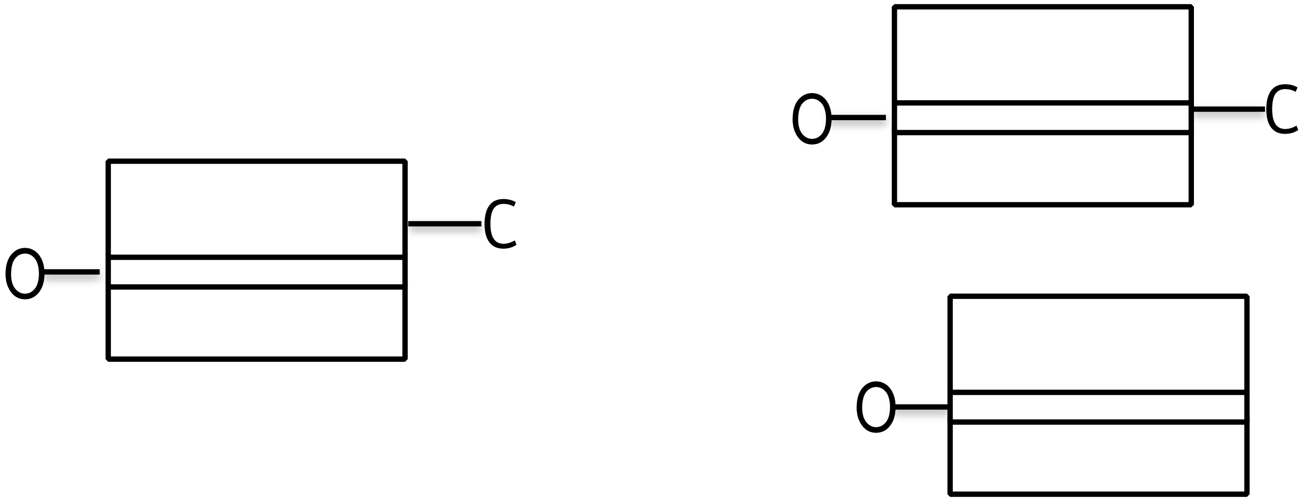
(Lego)

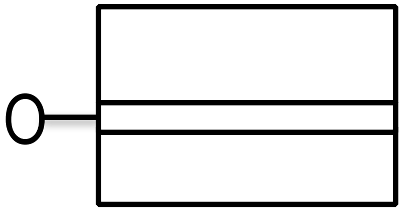


OO Prog

(Lego)







==

If/switch ==> EH

Error

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

==

If/switch ==> interface

Flow

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

< > <= >= ==

If/switch ==> ?

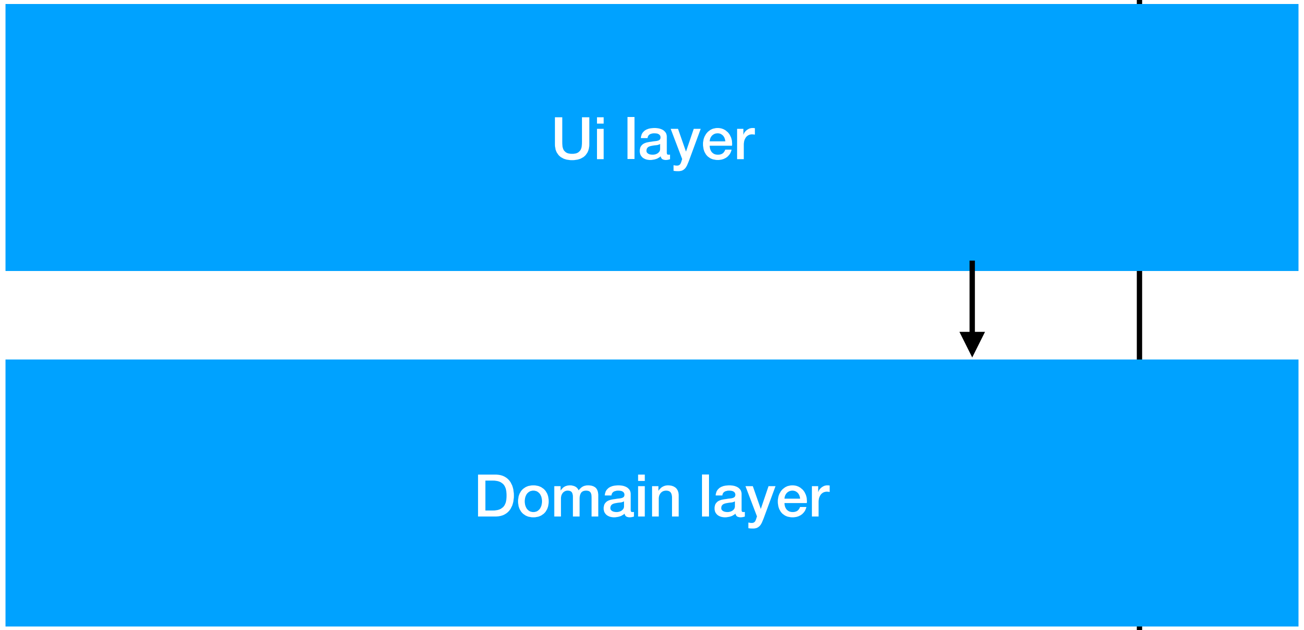
Domain rule

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

obj.f1();

Method Call

coupling ==> interface typing
Coupling ==> function Objects
Coupling ==> duck typing



new CA();

Instantiation

coupling ==> DI
coupling ==> factory

Abstraction

```
********** interface

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

```
class Parrot implements Bird{
  public void fly(){
    ....
  }
}

do(new Parrot());
```

```
//************** duck

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

```
class Parrot {
  public void fly(){
    ....
  }
}

do(new Parrot());
```

```
//********* lambda

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

```
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
- Destry thread - 100,000 cpu cycles
- I/O operations
- Exe Db command - 45,00,000 cpu cycles
-

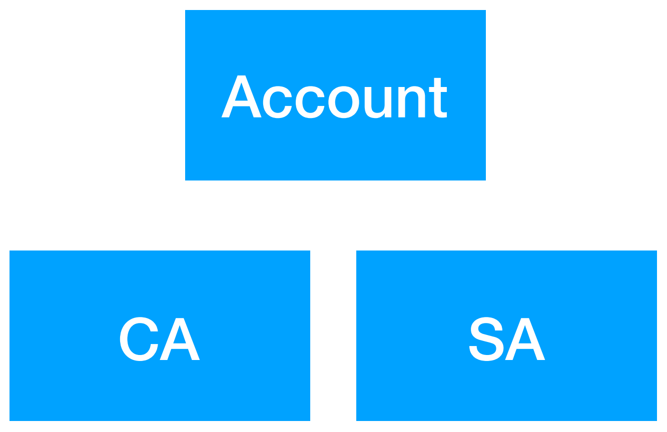
Design Check list

SOLID principles

+ **L**SP
+ **I**SP
+ **S**RP (*)
    # 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 (*)
+ **D**IP
+ **O**CP (open for add, closed for change)
+ Program to an Interface
+ Cyclomatic Complexity < 10
+ Prefer composition over Inheritance
+ Design By Contract (DBC)

- Flag
- Overloading Polymorphic Types
- 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
- Bi Directional / Cyclic Coupling
- Runtime Type Identification
- Downcasting

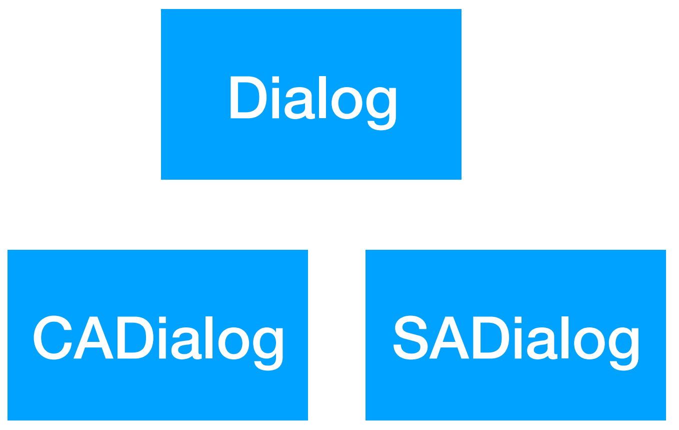
	Inheritance / extends	**Composition / Aggregation / Association**
Reuse	Within the sub classes	Any where
Coupling	High	Low (DI)
Change Parent at runtime	No (compile time)	Yes
Lazy Load Parent	No	Yes
Add Parent at runtime	No	Yes



Account

CA

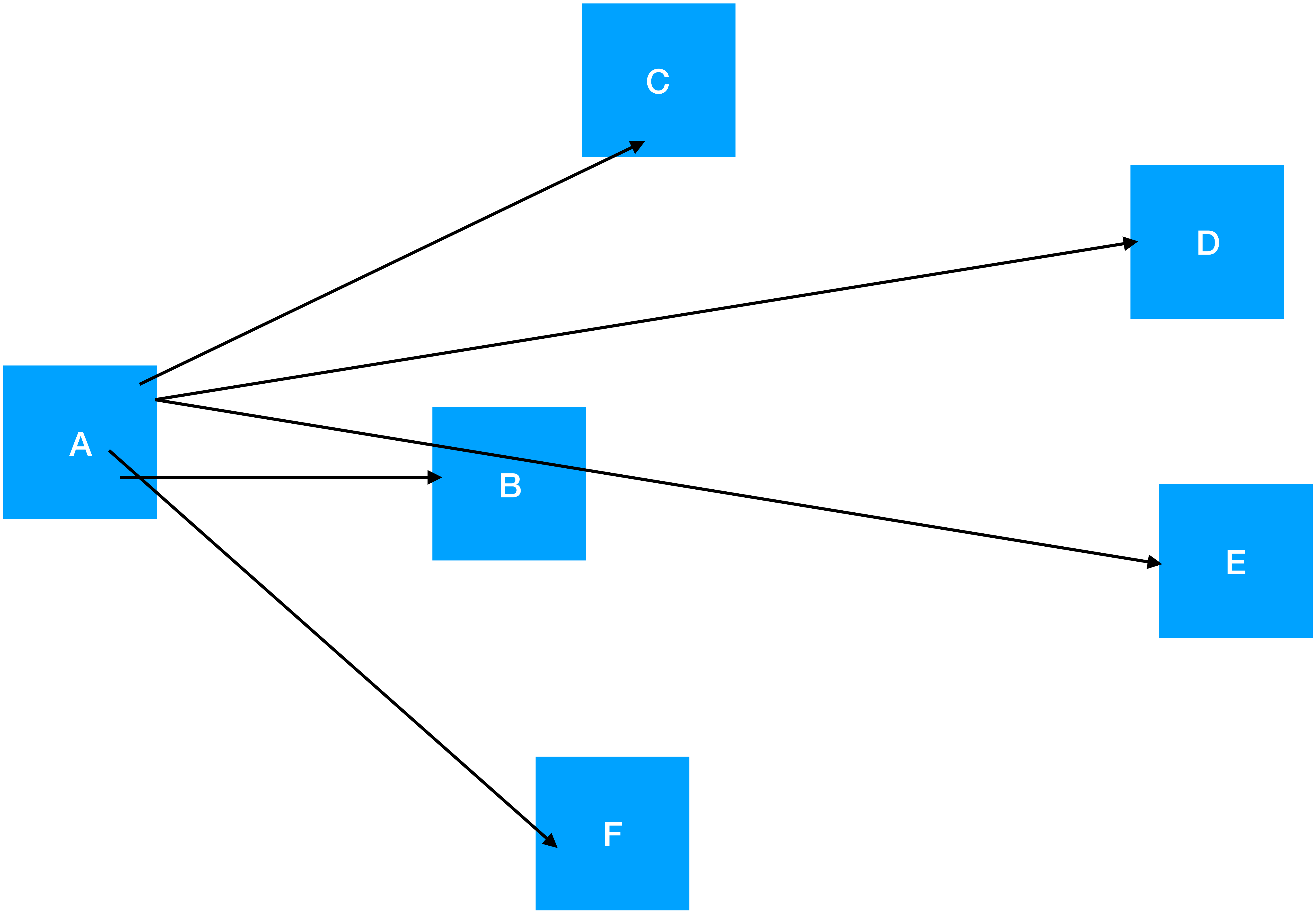
SA

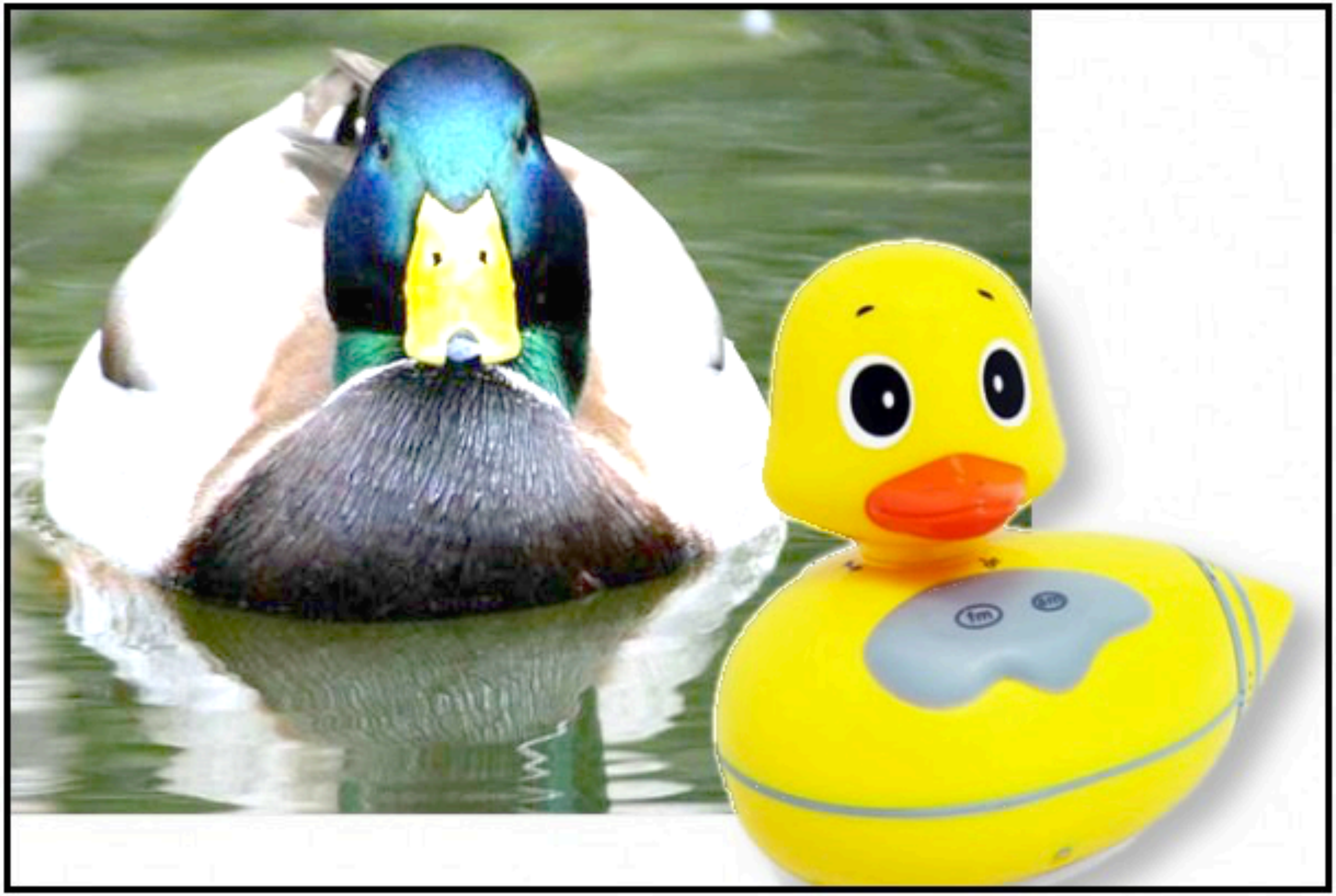


Dialog

CADialog

SADialog


C
D
A
B
E
F

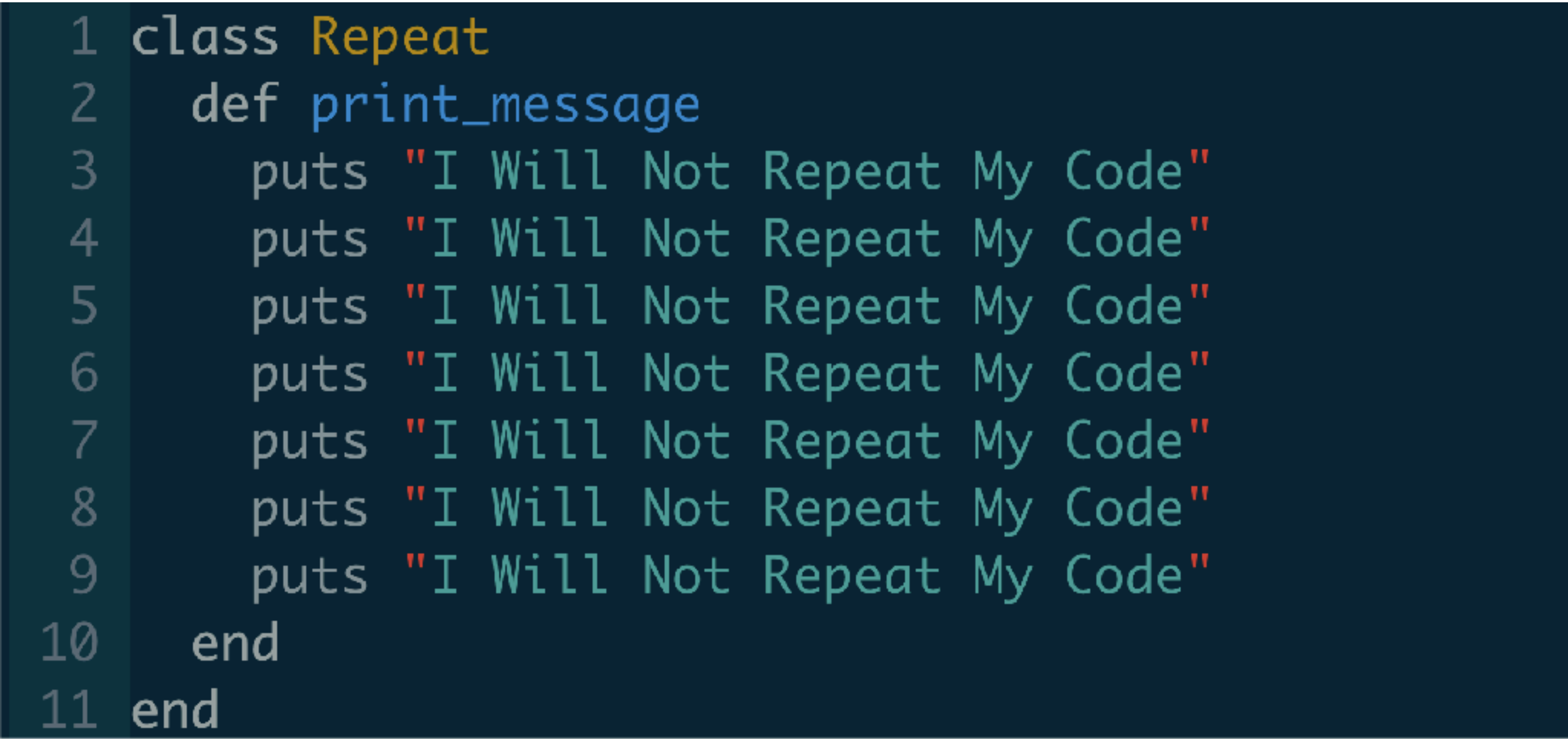


LISKOV SUBSTITUTION PRINCIPLE

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"
  end
end
```

Software Engineering v/s Tuning

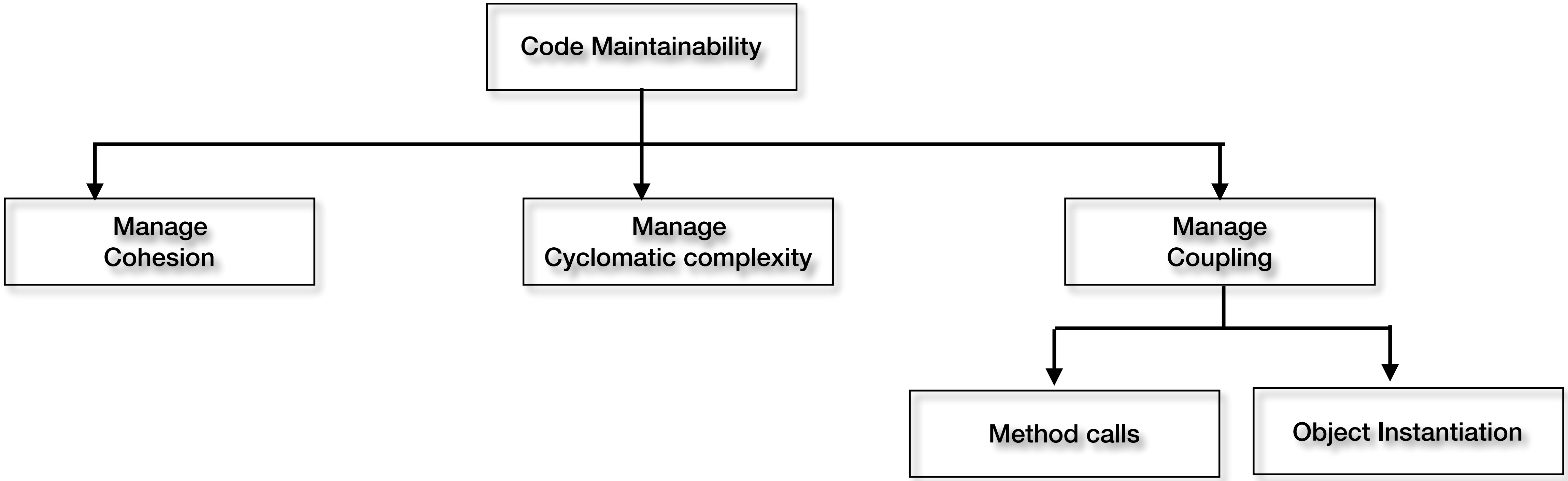
Quality

# Performance Engineering

# Performance Tuning

# Threat Modeling

Ethical hacking



Code Maintainability

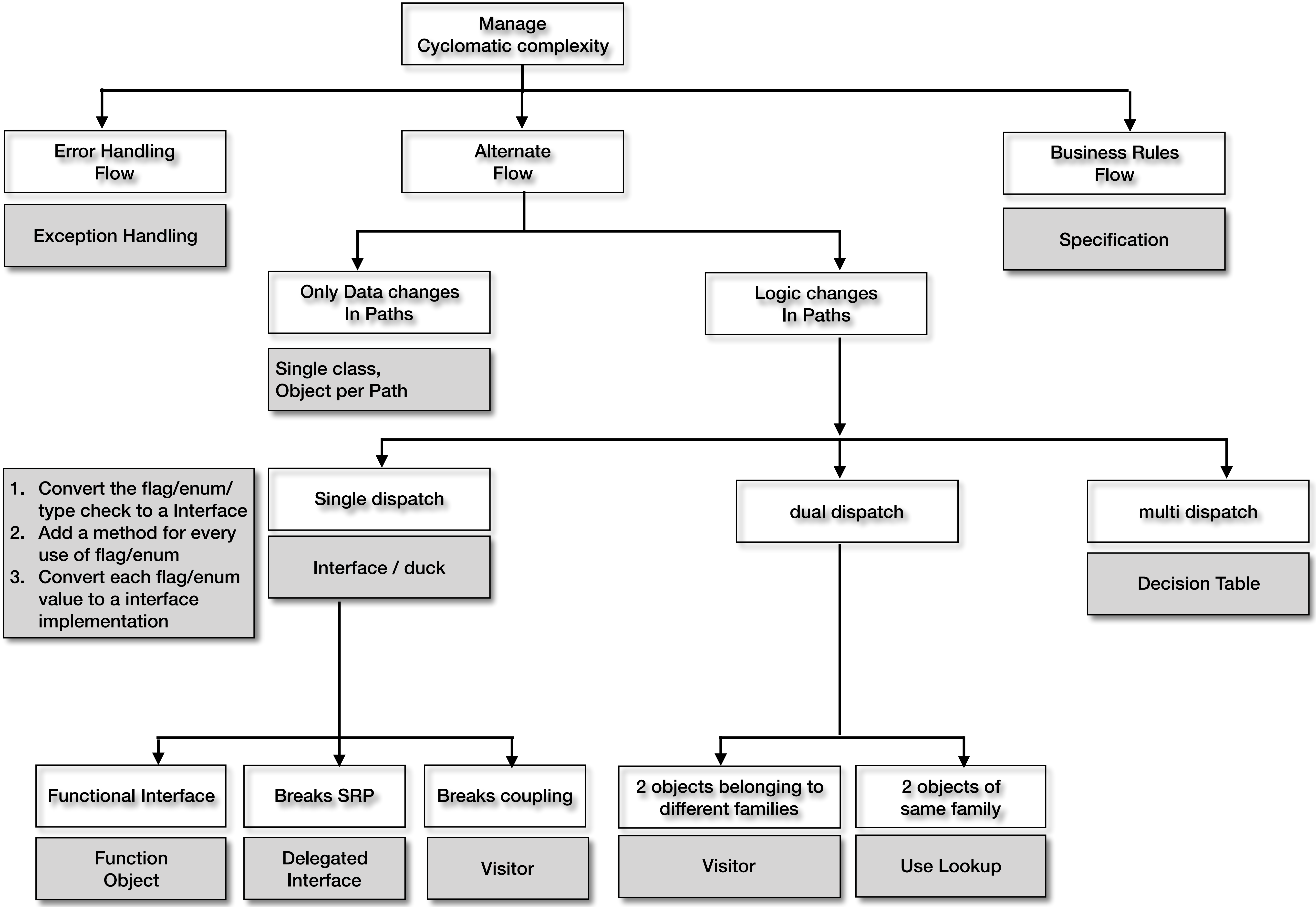
Manage
Cohesion

Manage
Cyclomatic complexity

Manage
Coupling

Method calls

Object Instantiation



Manage Cyclomatic complexity

Error Handling Flow

Exception Handling

Alternate Flow

Only Data changes In Paths

Single class, Object per Path

Logic changes In Paths

Single dispatch

Interface / duck

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

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

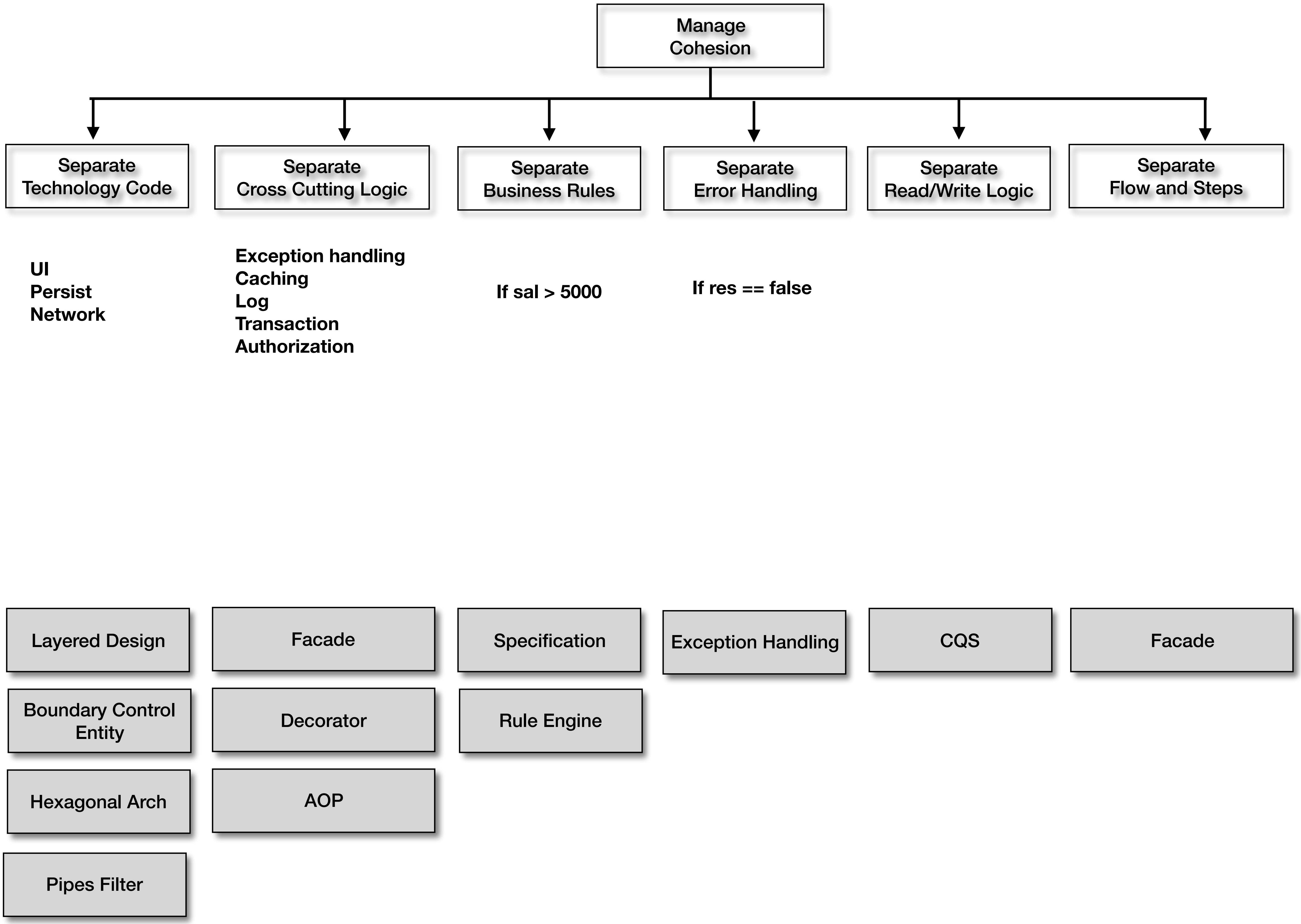
Use Lookup

multi dispatch

Decision Table

Business Rules Flow

Specification



Manage Cohesion

Separate Technology Code

UI
Persist
Network

Layered Design

Boundary Control Entity

Hexagonal Arch

Pipes Filter

Separate Cross Cutting Logic

Exception handling
Caching
Log
Transaction
Authorization

Facade

Decorator

AOP

Separate Business Rules

If sal > 5000

Specification

Rule Engine

Separate Error Handling

If res == false

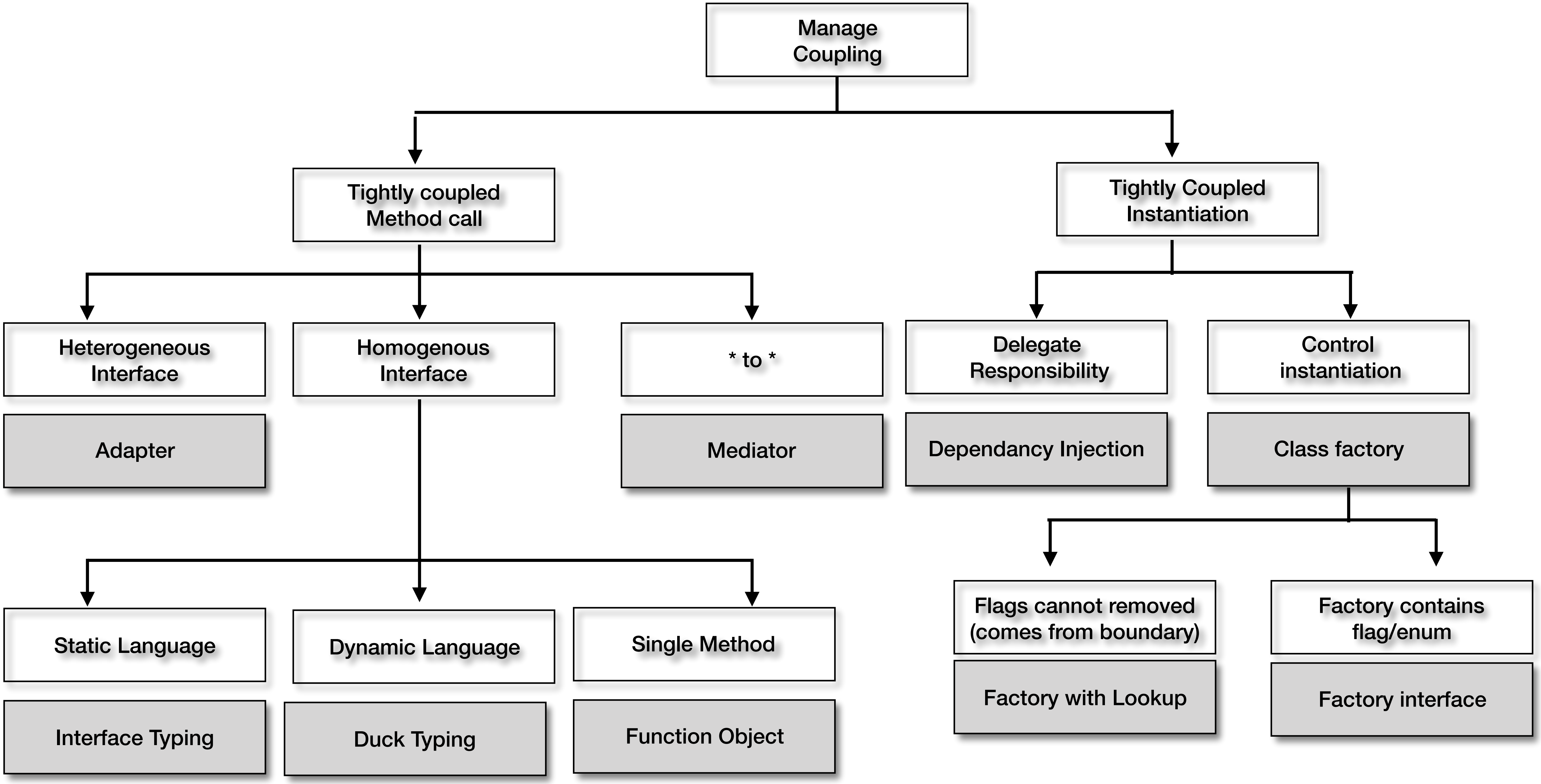
Exception Handling

Separate Read/Write Logic

CQS

Separate Flow and Steps

Facade



Manage Coupling

Tightly coupled Method call

Tightly Coupled Instantiation

Heterogeneous Interface

Adapter

Homogenous Interface

* to *

Mediator

Delegate Responsibility

Dependancy Injection

Control instantiation

Class factory

Static Language

Interface Typing

Dynamic Language

Duck Typing

Single Method

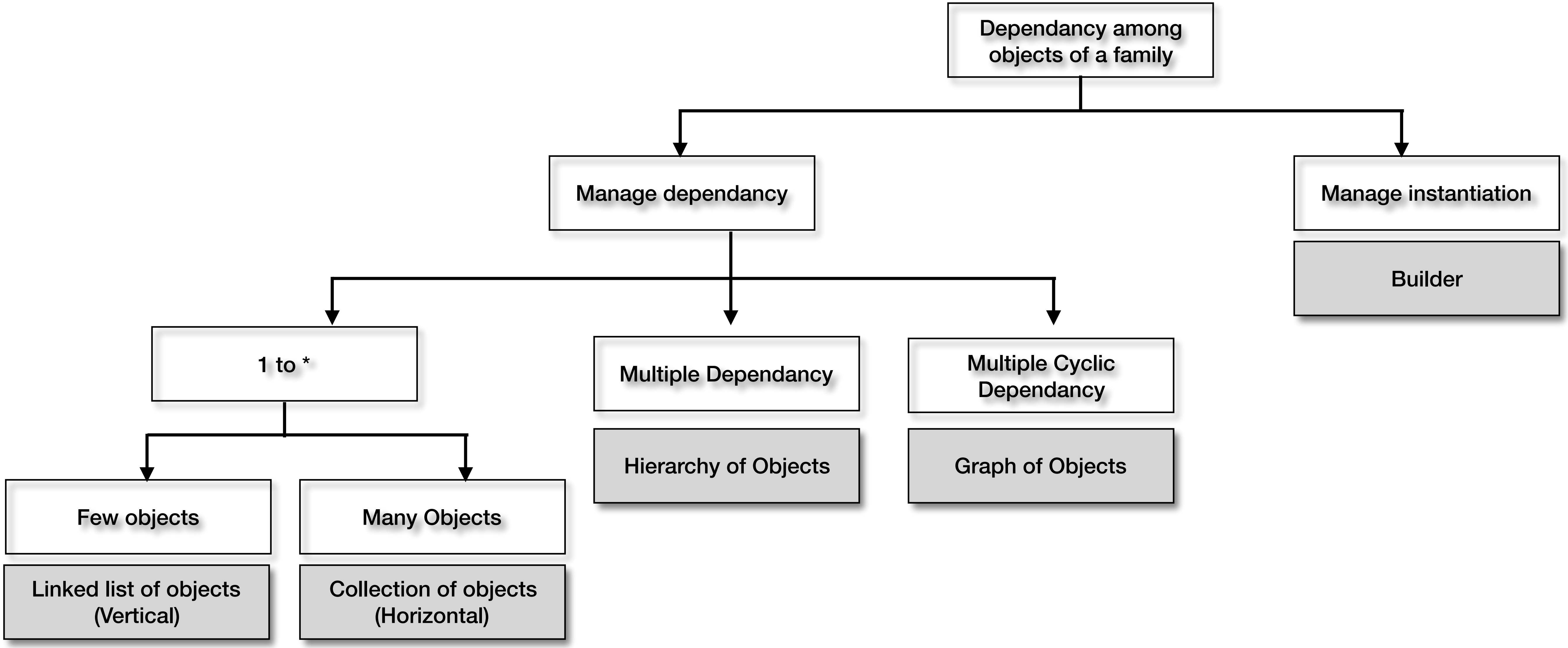
Function Object

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

Multiple Cyclic Dependancy

Hierarchy of Objects

Graph of Objects

Few objects

Many Objects

Linked list of objects (Vertical)

Collection of objects (Horizontal)