Mubarak

About me

- skan.ai
- ai.robotics
- Welldoc

Agenda

- Cyclomatic Complexity
    - Interface, duck, Lamda, EH
- Coupling
    - Interface, duck, Lamda,Adapter, Mediator
- Composition
- High Cohesion
    - Delegation

Architecture vs Design

+

Boundary Control Entity
KISS
Visitor Pattern
LSP
High Cohesion
SRP (***)
Cyclomatic complexity < 10
DIP (Dependancy inversion)
DRY (***)
ISP
Program to an interface
Interface size
max methods : 12
Avg methods : 5
OCP
Prefer association over inheritance
Null Pattern

-

Magic numbers/string
Int Flags
Bool flags
Null flags
Functional interface/ lilliput
Tight coupling
Cyclic/bi-directional Coupling
God Class
Command Pattern
Type check is sinful
Down cast
Overloading Ploymorphic Type
Static Methods
Inheritance
Boolean/ Optional/ nullable fun parameter
Boolean/ Null / int return for error

```
void fun(bool b,Emp e, Optional ){

  2 things
  Breaks SRP

}
```

Code segment

```
void f(CA a) {} //1
 void f(CB b) {} //2
 void f(CC c) {} //3

 void f(CA a) {} //4
 void f(CB b) {} //5
 void f(CC c) {} //6

 void f(CA a) {} //7
 void f(CB b) {} //8
 void f(CC c) {} //9
```

Data segment

CX vtbl

CA - 1,
CB - 2,
CC - 3

CY vtbl

CA - 4,
CB - 5,
CC - 6

CZ vtbl

CA - 7,
CB - 8,
CC - 9

Heap

CC

CZ/CY/CX

vptr

?

Runtime

Stack

```
do(new CX(),new CA());
do(new CX(), new CB());
do(new CX(),new CC());

do(new CY(),new CA());
do(new CY(), new CB());
do(new CY(),new CC());

do(new CZ(),new CA());
do(new CZ(), new CB());
do(new CZ(),new CC());
```

a

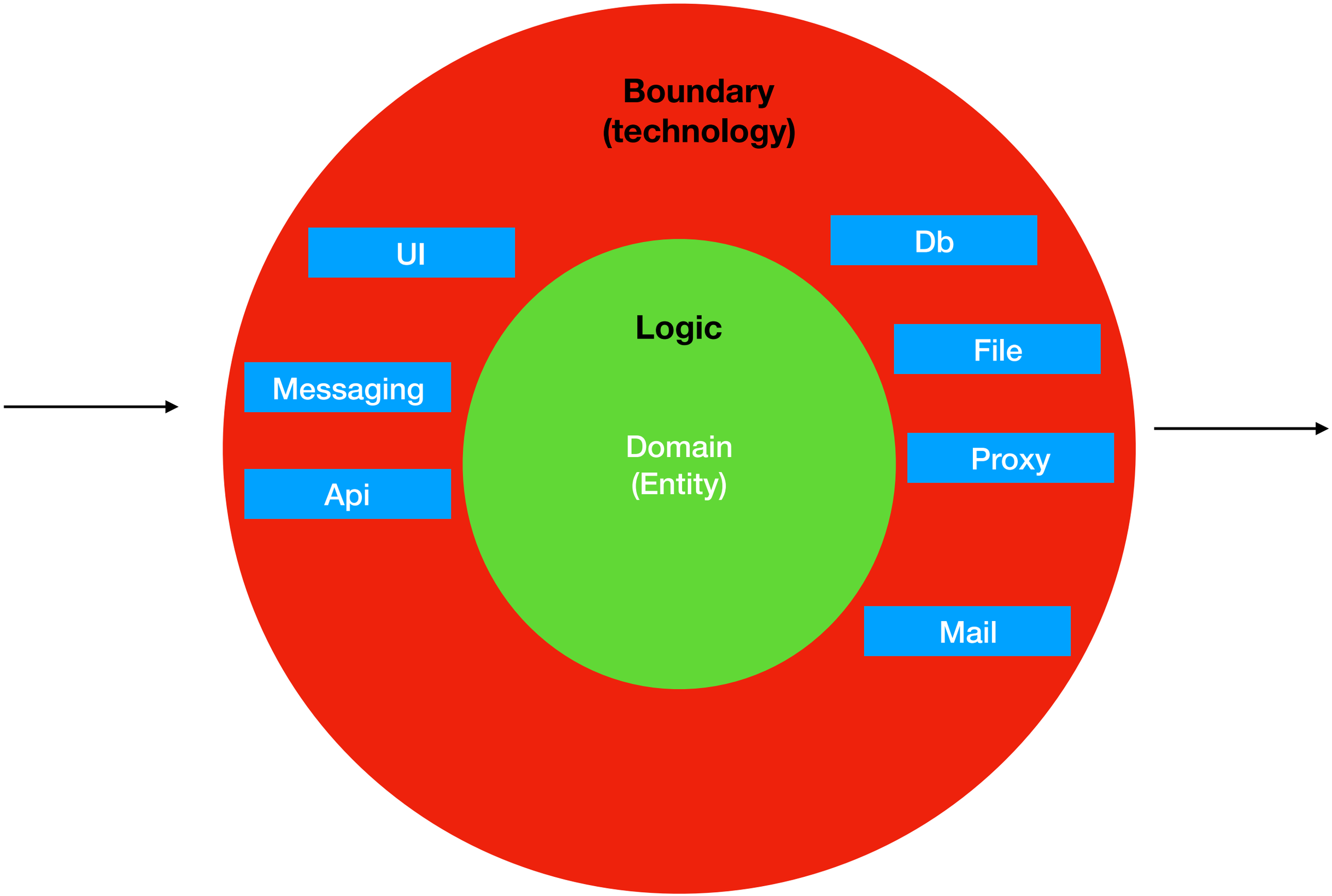
x

```
do(CX x,CA a )
{
      x.f(a)
      x.vptr.vtbl[0](a)
}
```

Compile time

Static

- Boundary and Domain logic is mixed
- Error Handling is mixed with domain logic
- Flow is mixed with steps
- Logic is mixed with rules
- Cyclomatic Complexity
-


Boundary
(technology)
UI
Db
Logic
File
Messaging
Domain
(Entity)
Proxy
Api
Mail

Proc	OO
Flags	Lamda
Enum	Interface / duck
	Delegated Interface / duck
	Exception Handling

Int flag

```
if(flag ==1)
 …a
if(flag == 2)
 .. b
if(flag == 3)
 .. c
```

Flag flag

flag.fa();

```
if(flag ==1)
 …x
if(flag == 2)
 .. y
if(flag == 3)
 .. z
```

flag.fx();

```
# Every flag transforms to an interface
# each group of logic is transformed to a method
# one implementation for each flag value
```

```
Interface Flag
{
   fa();
   fx();
}

Class CA implements Flag
{
   fa() { … a …}
   fx() { … x …}
 }
 Class CB implements Flag
 {
   fa() { … b …}
   fx() { … y …}
 }
Class CC implements Flag
{
   fa() { … c …}
   fx() { … z …}
}
```

Abstraction

c++, java, C#	Py,js	Java 8, C#, c++ 11,py, Haskel
Interface typing	**Duck typing**	**Lamda**

Interface Bird
{
fly()
}
Explicit
	Implicit	**Implicit**

do(Brid bird)
{
bird.fly();
}
	do(bird)
	{
	bird.fly();
	}
		do(fly)
		{
		fly();
		}

class Parrot implements Bird
{
fly() { …. }
}
	class Parrot
	{
	fly() { …. }
	}
		class Parrot
		{
		flyHigh() { …. }
		}

do(new Parrot());	do(new Parrot());	p= new Parrot() do(()=>p.flyHigh());

Low Coupling

Method call Obj.fun()	Instantiation new CA()	Deallocation delete pObj
Interface	Di	Virtual destructor
Duck	Factory	
Lamda		
Adapter		
Mediator		

Domain

UI

Account

CA

SA

Dialog

CADialog

SADialog

If

Except Handling

Interface

Class per flag

Object per flag

Error Handling

Alternate Flow Handling

Alternate Data Handling

Business Rules

If res == false

If acc_type == 2

If sal > 5000 and age < 18

If acc_type == 2

- Things which don’t change together should not be kept together

- Domain logic should not be mixed with technology logic

- Error handling logic should not be mixed with domain logic

-


Quality
Excellent
Good
Average
Poor

Quality

- Correctness
- Maintainability
- Availability
- Security
- Performance + Scalability (resource)
- Reliability(Trust)
- Usability
- Robustness (Rugud)
-

Tactic/Statergy

- Authentication
- Readable
- Logging
- Extensible
- **Concurrency**
- **Caching**
- **Compression**
- **Lazy Loading**
- **Object pooling**
- Error handling
- Low coupling
- ACID - transaction
- Input validation
- Modularity

Measure

- Latency
- Response time
- Throughput
- tps
-

	Proc	OO	Fun
	Java, C++, C#, Py, JS	Java, C++, C#, Py, JS	Java, C++, C#, Py, JS
Performance	-	-	
Security	-	-	
Time to develop/ Cost	Winner	Loser	
Learning Curve	Winner	Loser	
Testability	Loser	Winner	
Manage large code complexity	Loser	Winner	

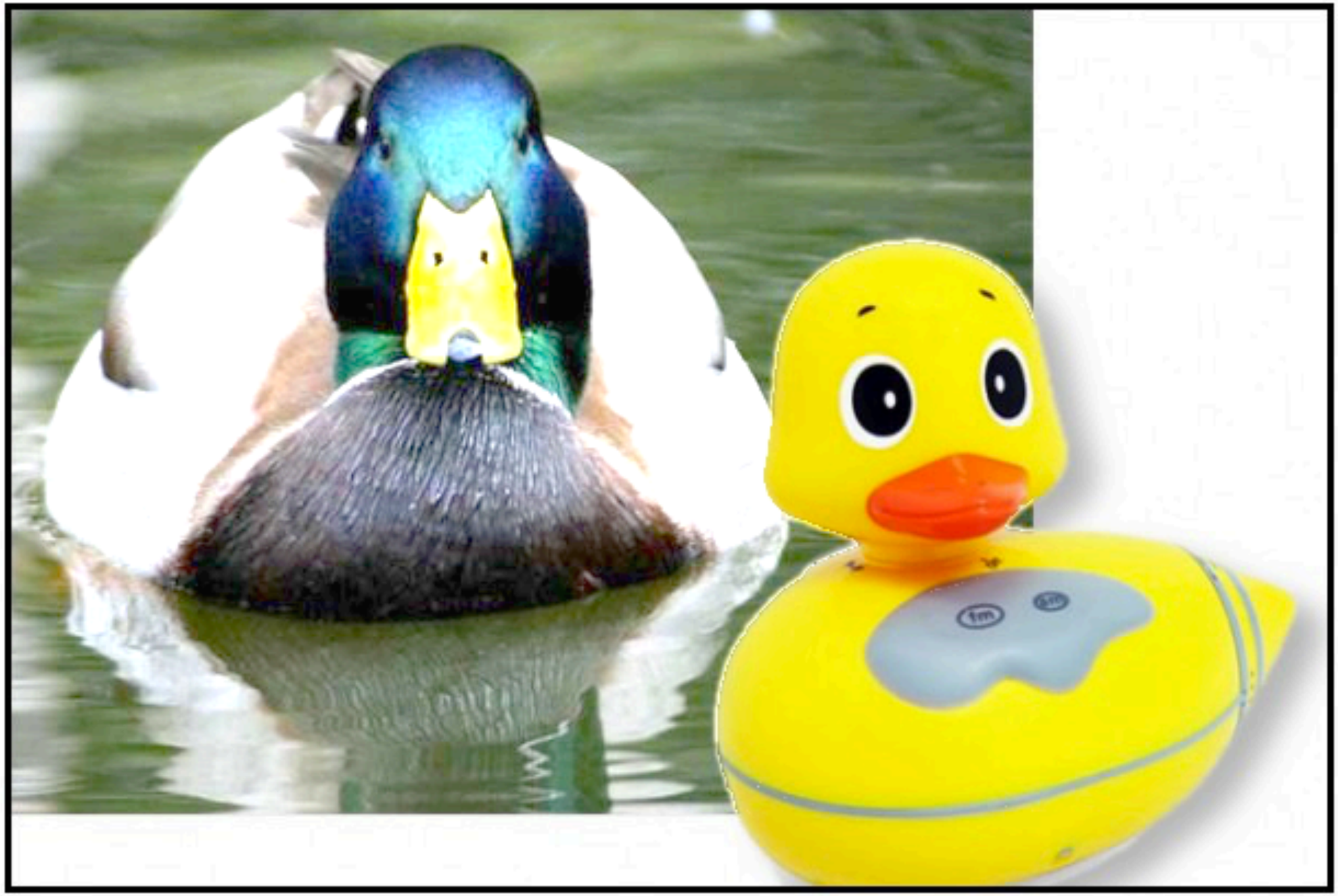
```
interface Bird
{
  fly()
  eat()
  quack()
  swim()
  sleep()
  buildNest()
  layEggs()
}
```

Diabetes -> Parrot
Asma -> Ostritch

```
function(Bird bird)
{
    bird.fly();
}
```

```
interface Bird
{

}
interface FlyingBird
{

}
interface NestingBird
{

}
```

```
Interface Account
{
    public abstract withdraw(double amount);
    public abstract deposit(double amount);
    public abstract void persist();
}
abstract class AccountBase implements Account
{
   public void save()
    {
        ... logic 1
        persist();
        ... logic 2
    }
}
class CA extends AccountBase
{

}
```

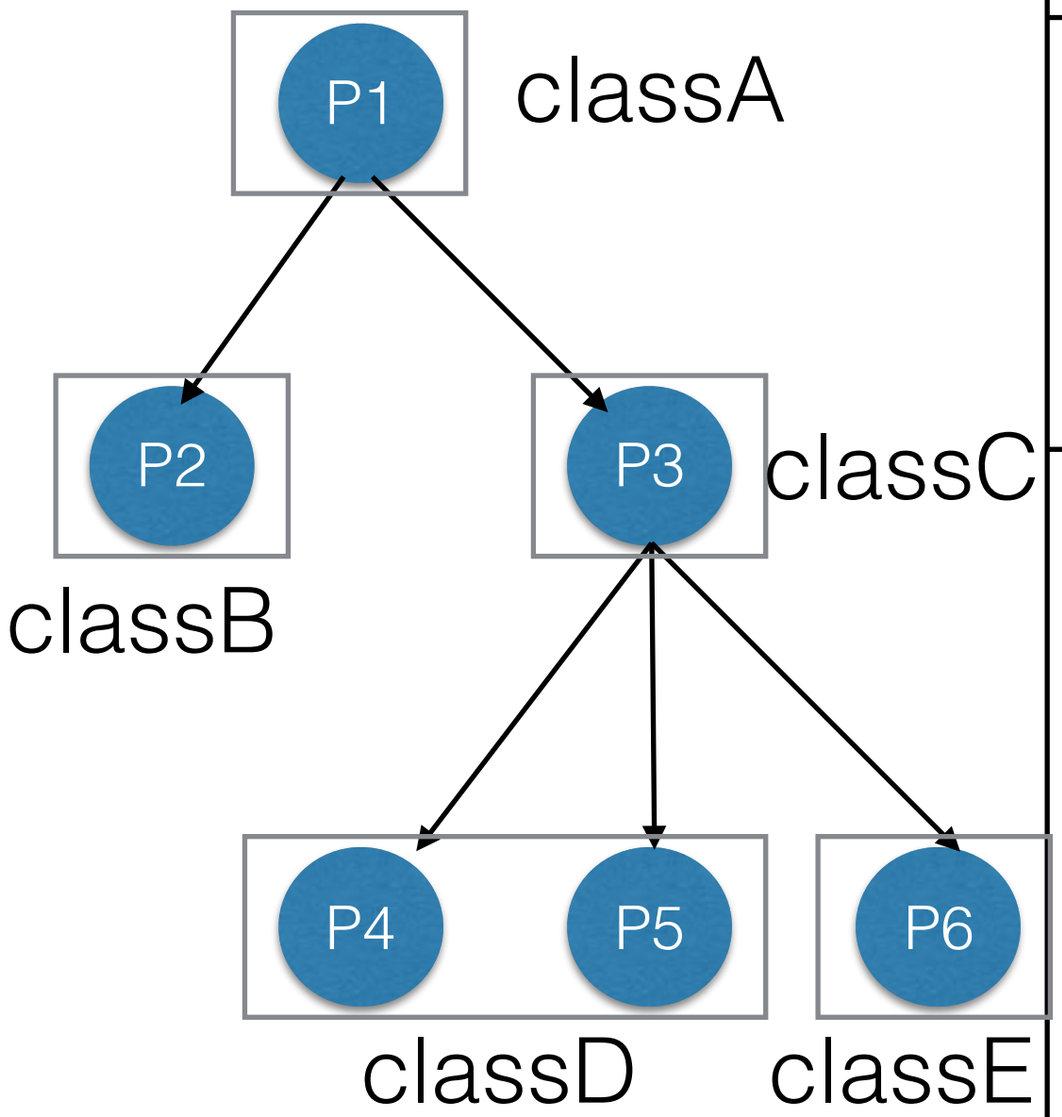


LISKOV SUBSTITUTION PRINCIPLE

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

Procedural Prog

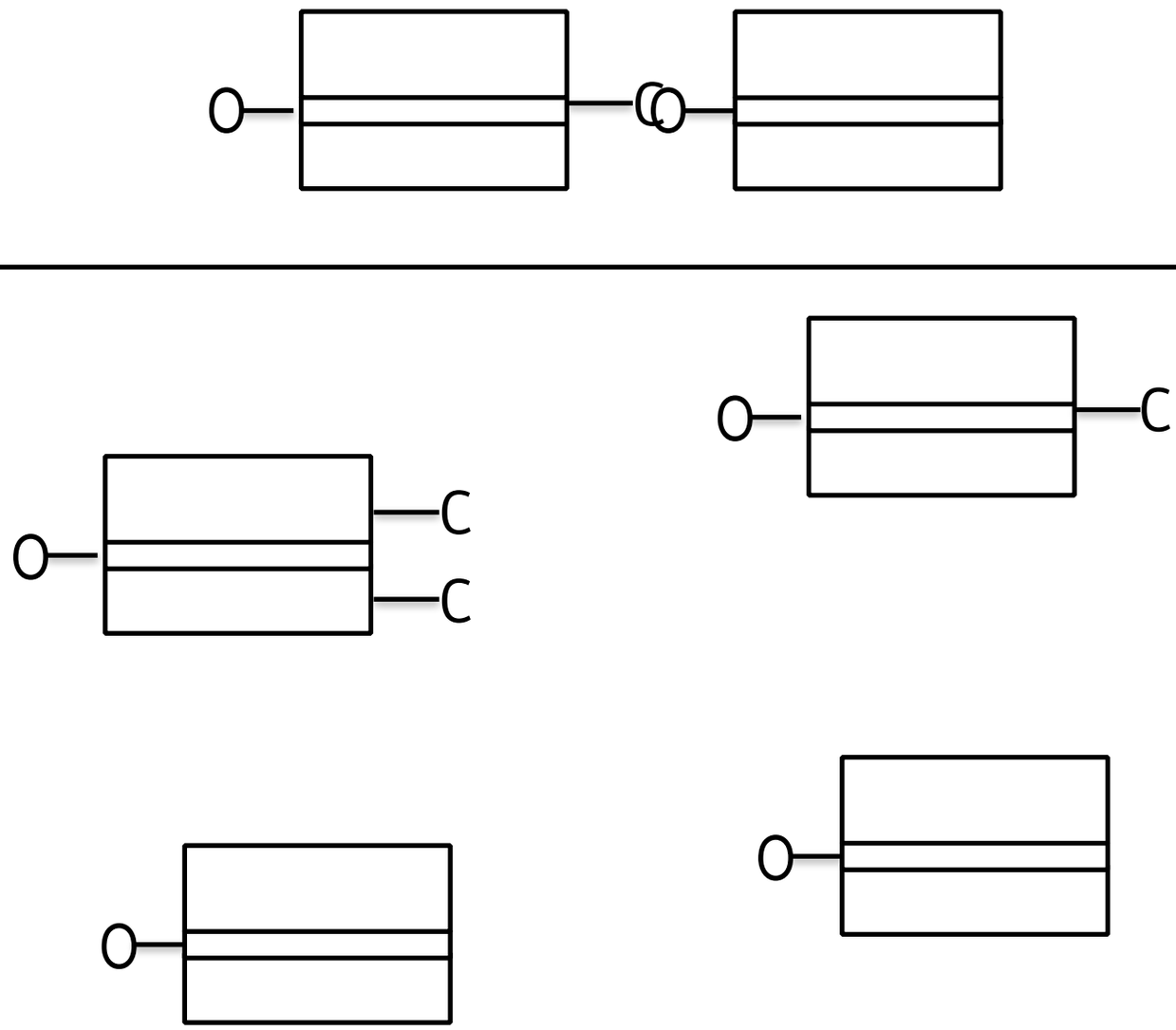
(tree)



(top down)

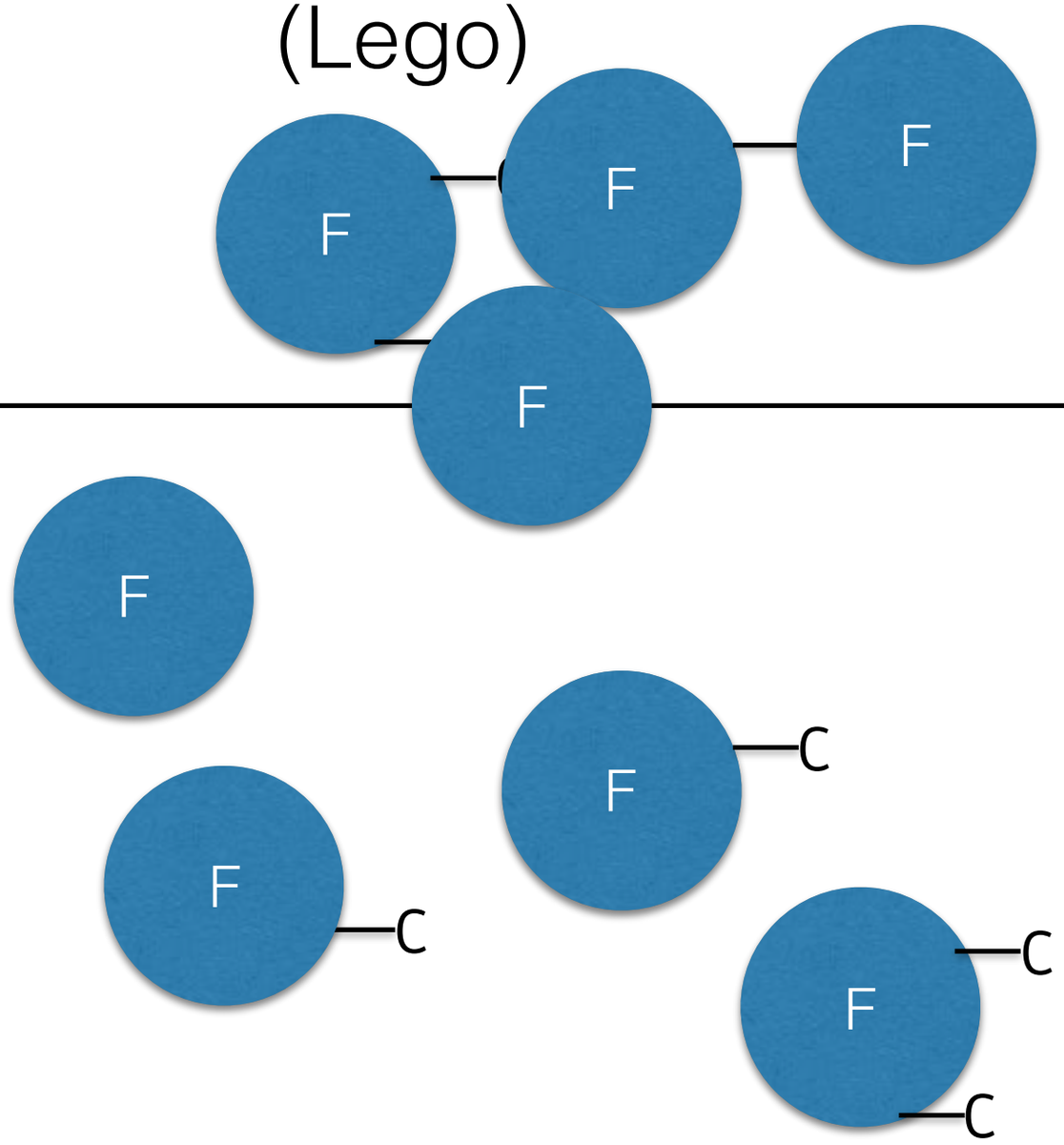
OO Prog

(Lego)

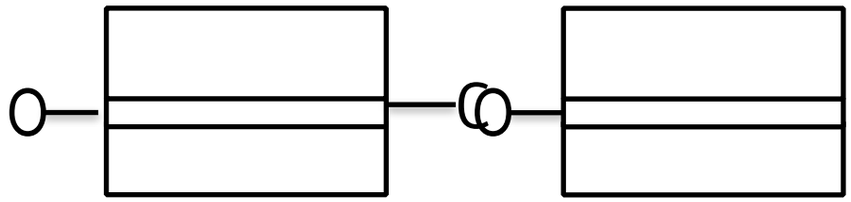


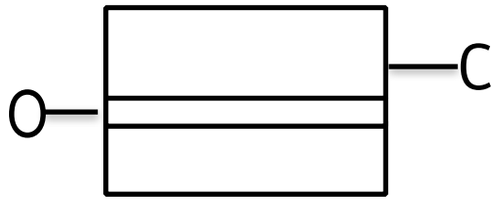
(bottom up)

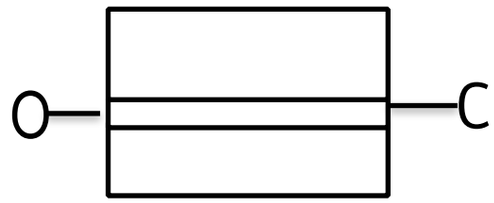
Functional Prog (Lego)

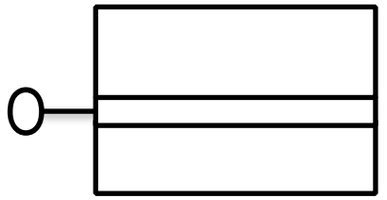


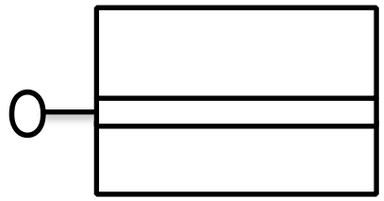
OO Prog (Lego)











Quality

Tactics/statergy (Architecture Design)

Requirement

Skeleton for code (Code design)
(oo/proc/fun)

Code Maintainability

Quality

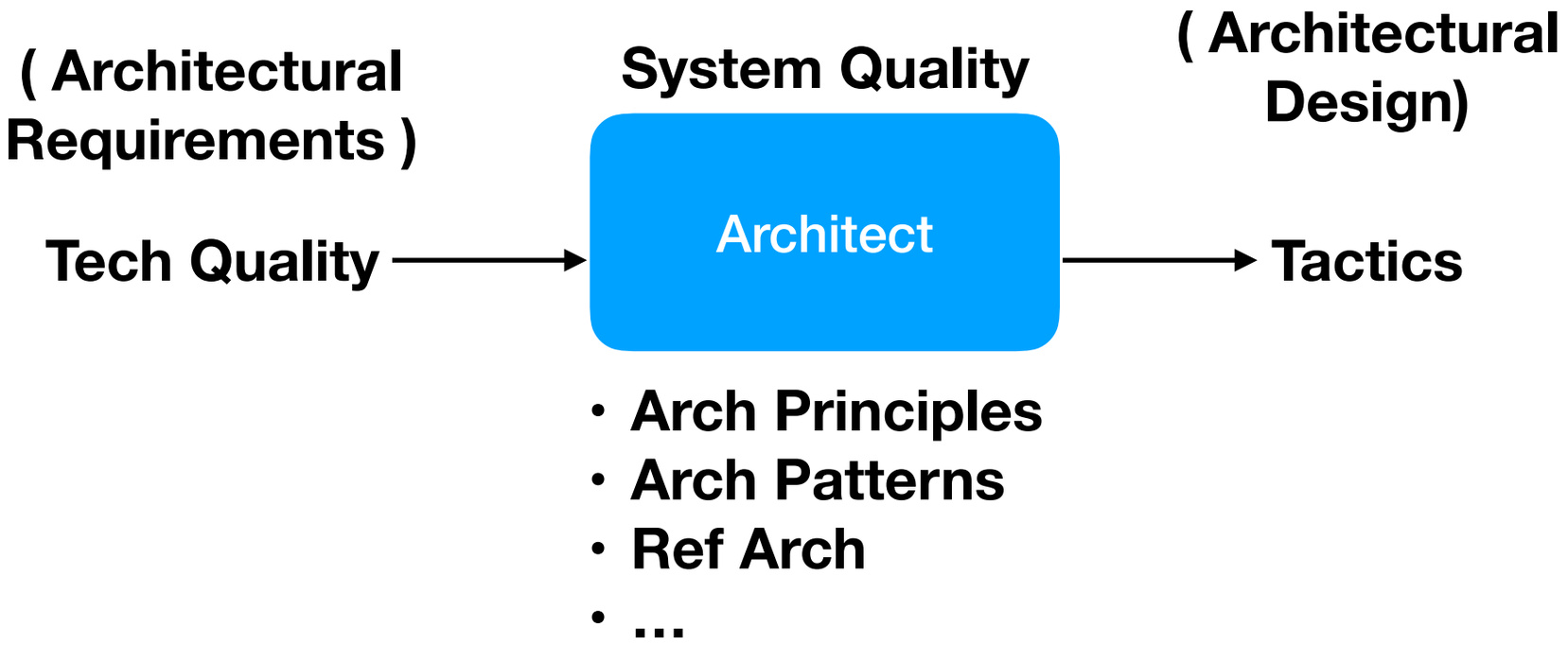
1. Cost
2. Time

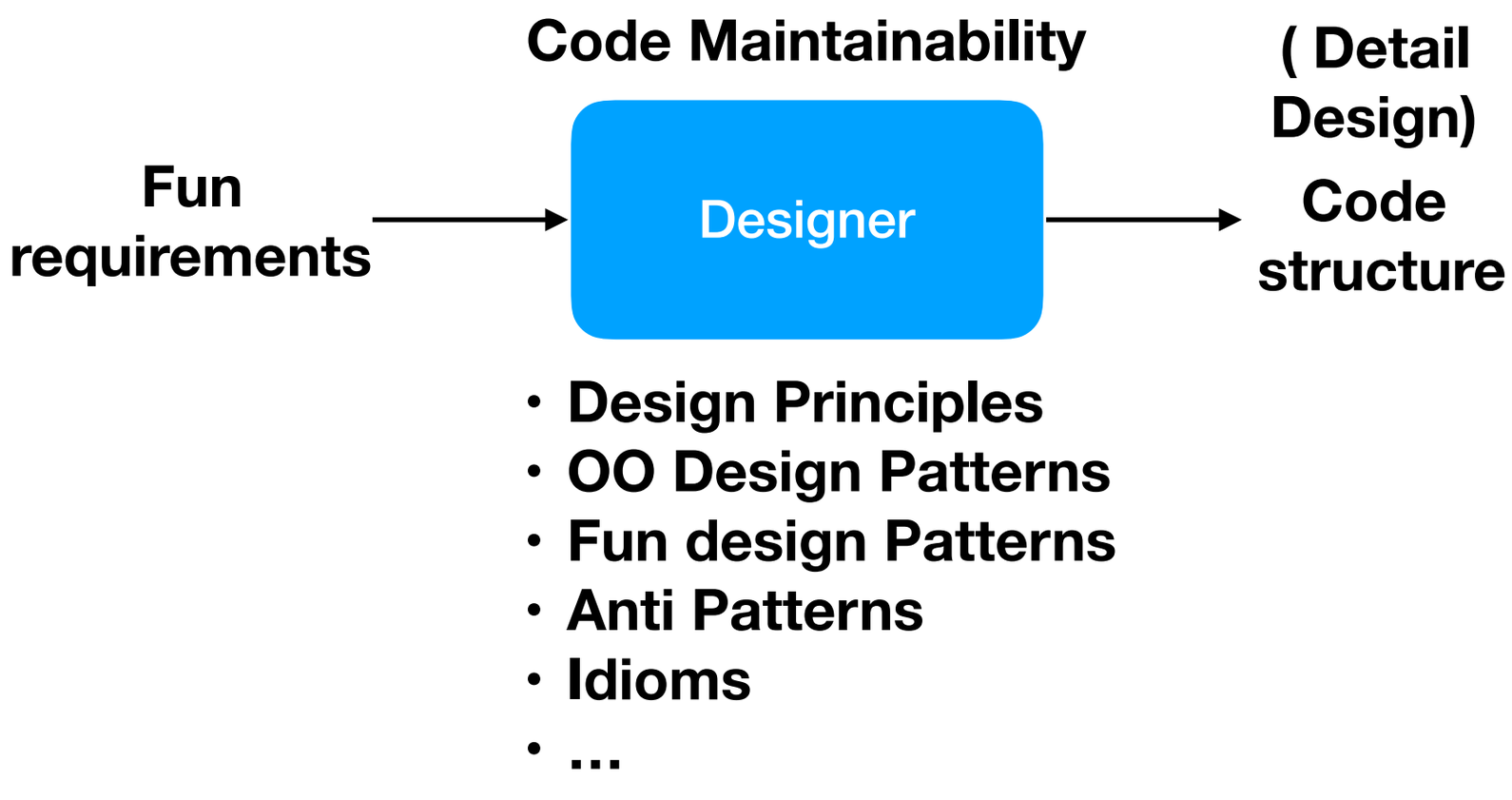
Tech Quality

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
}
```

==

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();

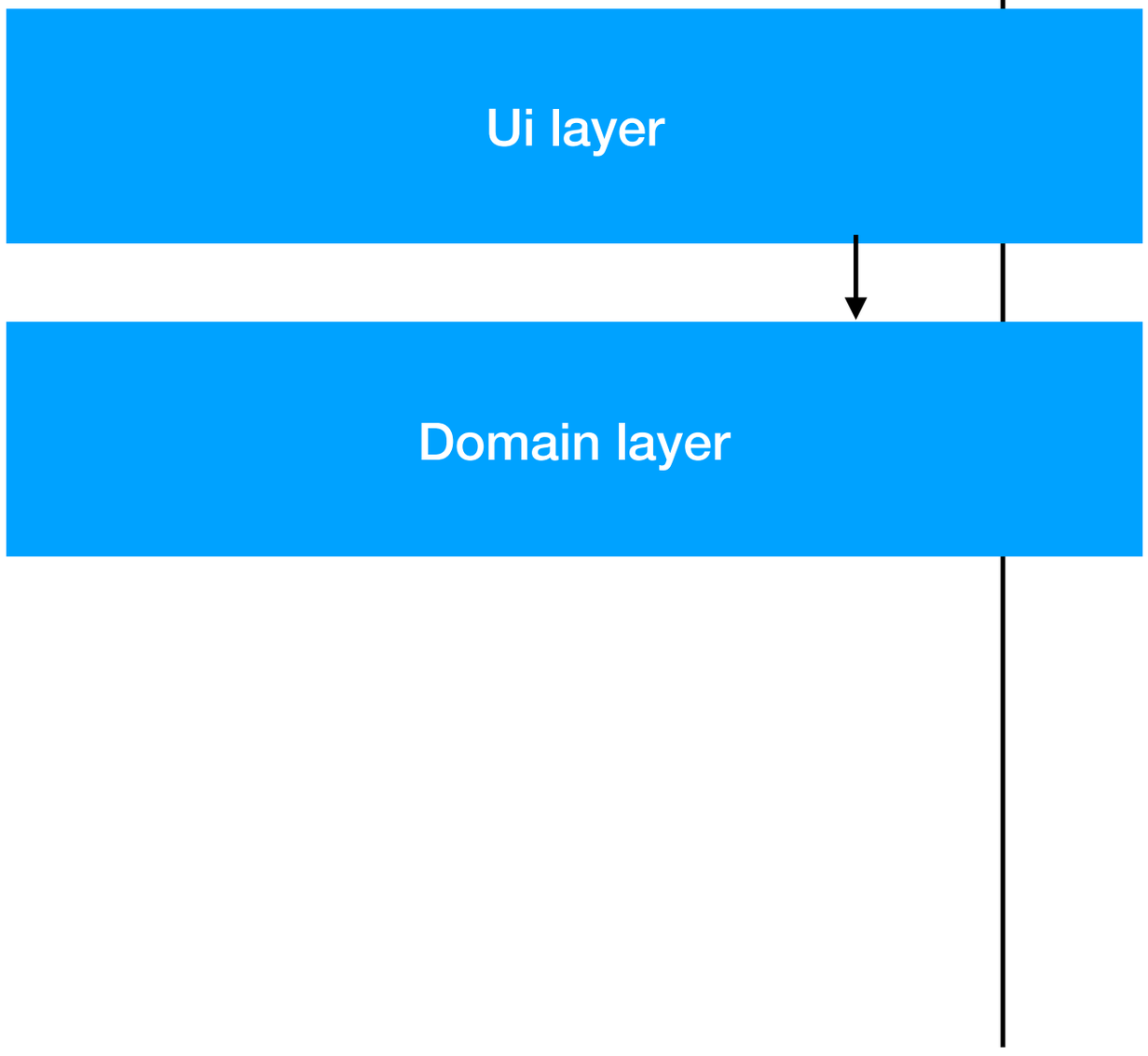
Method Call

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();
    }
```

```
//*************** 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

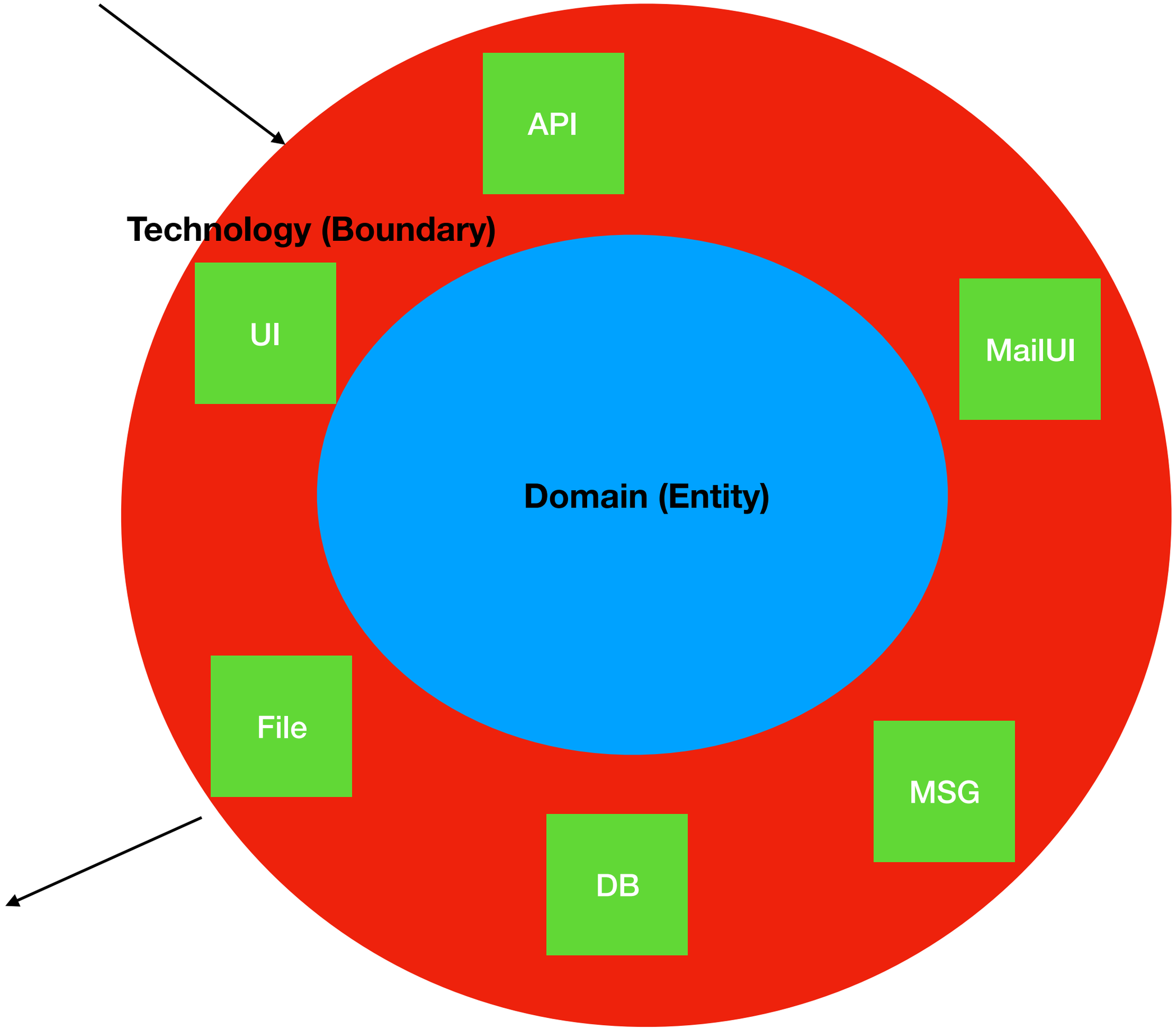
- for vs foreach
- a+b - 3 cpu cycles
- Create thread - 200,000 cpu cycles
- Destry thread - 100,000 cpu cycles
- I/O operations
- Exe Db command - 45,00,000 cpu cycles
-

Design Check list

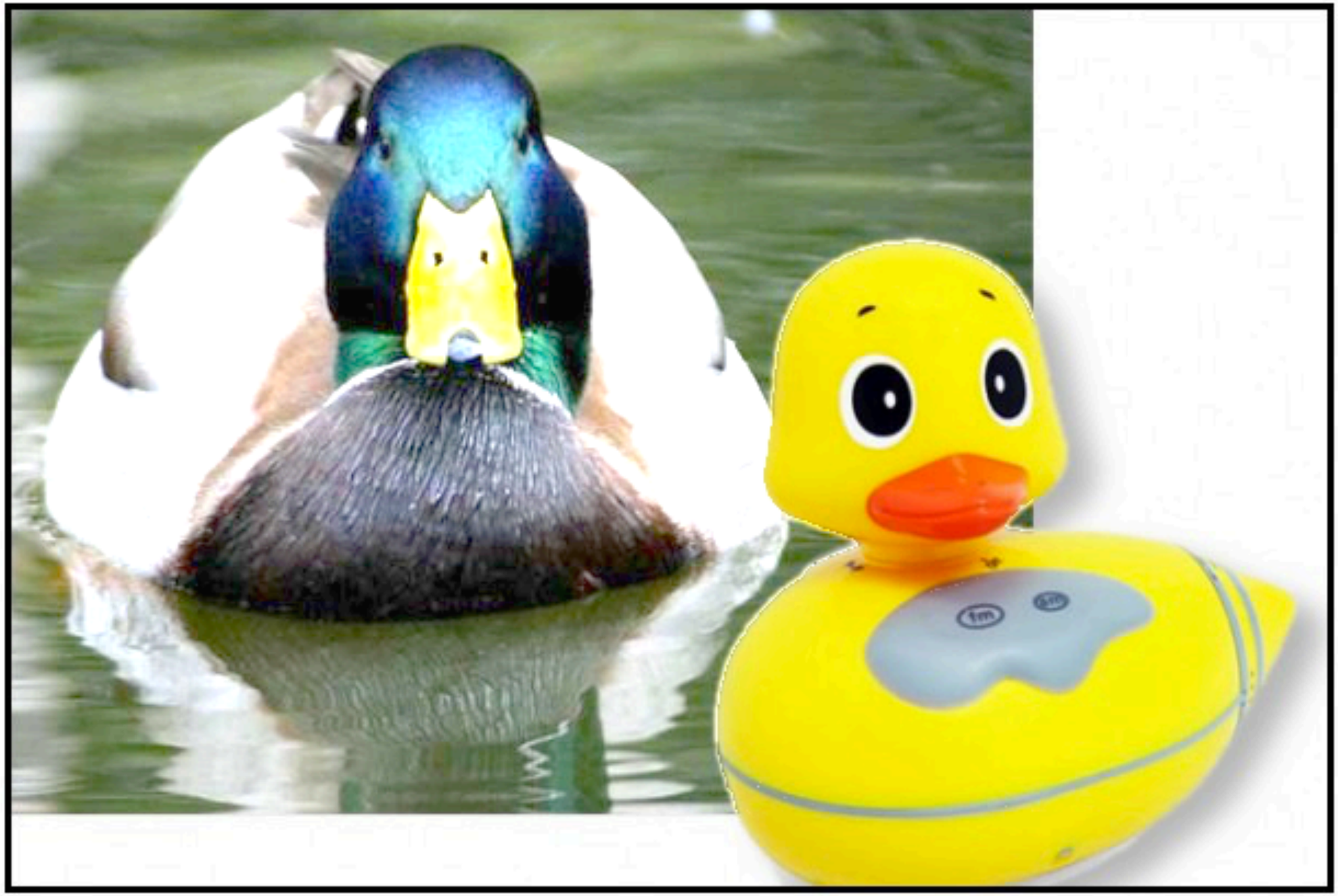
SOLID principles

+ **L**SP
+ KISS
+ **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)
+ Specification Pattern
+ Boundary Control Entity (Hexagonal arch)

- bool/nullable/Optional parameter
- 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
- Singleton Pattern


API
Technology (Boundary)
UI
MailUI
Domain (Entity)
File
MSG
DB

	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



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
```

Polymorphism

Single Dispatch Polymorphism

obj.fun();

—> 1 | 2 | 3

Only with in the family
-> overriding

Outside the family
-> visitor

Dual Dispatch Polymorphism

(obj1, obj2).fun();

—> 1 | 2 | 3

Object of same family
-> Lookup (map)

Object of different family
-> visitor

Multi Dispatch Polymorphism

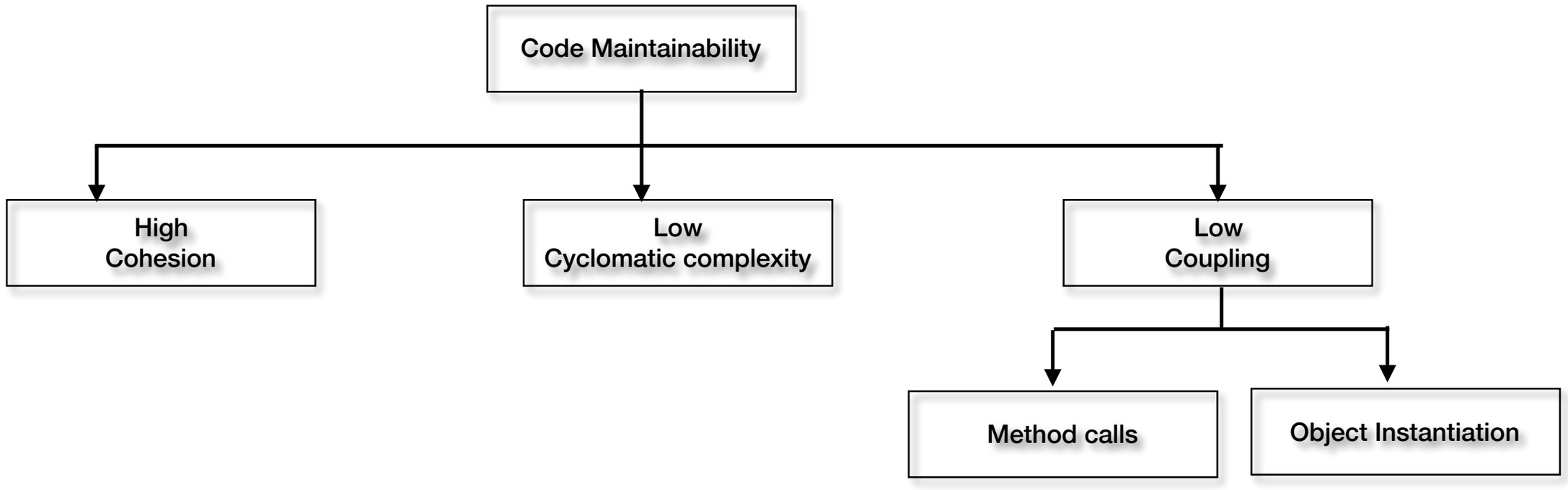
(obj1, obj2,…objn).fun();

—> 1 | 2 | 3

```
class Dept
{
   List<Emp> ref;
}
```

```
class Emp
{                        <— decorator pattern,
  Emp ref;               Chain of responsibility
  }
```

```
class Emp
{
  List<Emp> ref;       <— composite pattern
}
```


Code Maintainability
High
Cohesion
Low
Cyclomatic complexity
Low
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
          - Functional Interface
            - Function Object
          - Breaks SRP
            - Delegated Interface
          - Breaks coupling
            - Visitor
      - Flag from Boundary
        - Use Lookup
      - dual dispatch
        - 2 objects belonging to different families
          - Visitor
        - 2 objects of same family
          - Use Lookup
      - multi dispatch
        - Decision Table
  - Business Rules Flow
    - Specification
    - Rule Engine

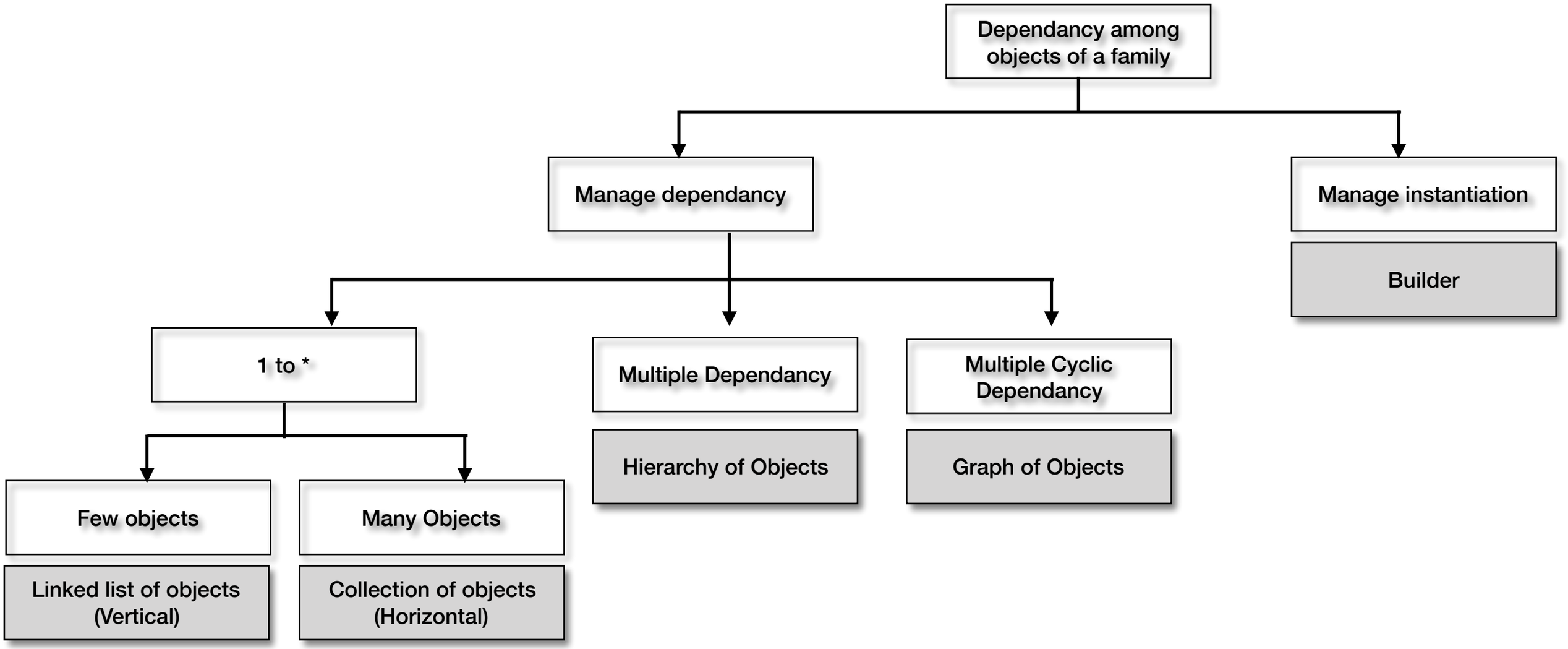
1. Convert the flag/enum/ type check to a Interface
2. Add a method for every use of flag/enum
3. Convert each flag/enum value to a interface implementation

Manage Cohesion

Separate Technology Code	Separate Cross Cutting Logic	Separate Business Rules	Separate Error Handling	Separate Read/Write Logic	Separate Flow and Steps	Unrelated Logic
UI Database File API EMail Messaging	Exception handling Caching Log Transaction Authorization	If sal > 5000	If res == false			Things which do not Change together

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)

emp->emp->emp->emp->emp->emp->emp->ceo

```
class Emp
{
  Emp mgr;
}
```

```
Interface Employee {  }

Class FTE implements Employee {
    Employee mgr;
}

Class CEO implements Employee {

}
```

Decorator pattern
Chain of Responsibility Pattern

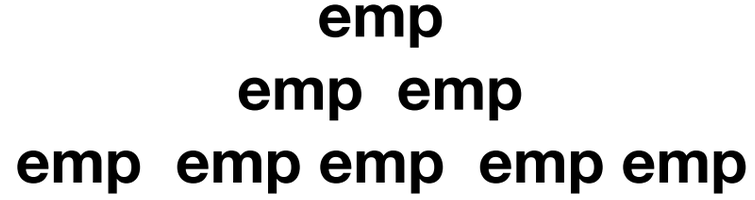
Visitor pattern

Composite pattern

Abstract Factory pattern

23 patterns
GOF design patterns

Command pattern
GOF Singleton pattern




```
class Emp
{
  List<Emp> reportees;
}
```

```
class Emp
{
  Emp left;
  Emp right
}
```

```
Interface Employee {  }

Class FTE implements Employee {
    List<Emp> reportees;
}

Class Trainee implements Employee {

}
```

Factory

- Creator Method <— static fun which returns obj
- GoF Factory Method <— polymorphic fun which returns obj
- Class Factory <— contains only creator/ factory methods
- GOF Abstract Factory <— a family of class factory

Collection

Stack

List

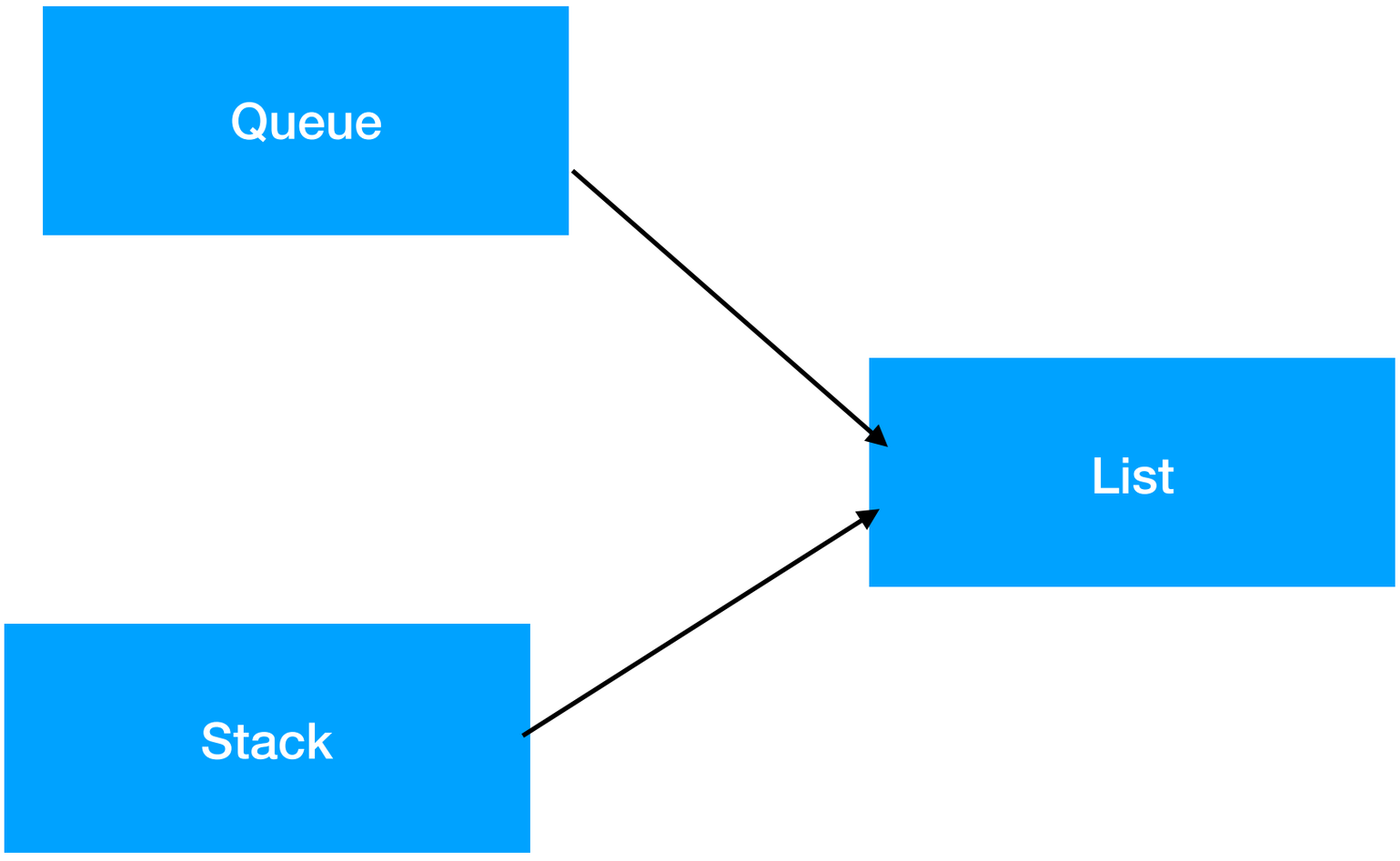
Queue

List

Stack

Queue

```
do(List list)
{
   list.remove(3);
}
```


Queue
List
Stack

<<Account>>

CA

SA

AccountService

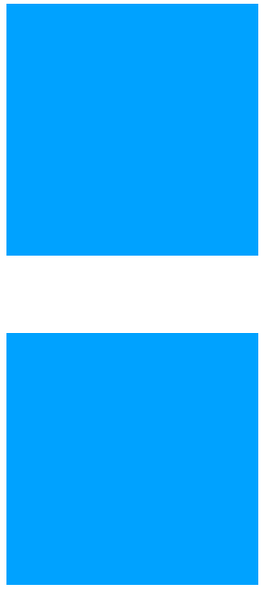
Account
Type

<<Type>>

Silver

Platinum

- XmlFormatter
- JsonFormatter
- TextEncoding
- BinaryEncoding



extends vs ref

	Extend (compile time)	Ref (runtime)

At runtime choose Parent	class CA extends ?	class CA{
		IX parent;
		}

Lazily Load Parent	Class CA extends CX{
	}
		class CA{
		IX parent;
		}

Multiple Inheritance	One	class CA{ IX parent; IY parent2;

Change Parent	class CA extends ?	class CA{
		IX parent;
		}

	Diamond problem	

```
class CA
{
   static CA ref;
   private CA() {}
   public static  CA get(){
      if(ref ==0)
         ref = new CA();

         return ref;
   }
}
```

Static Methods

Low coupling

```
class CY
{
   void do()
   {
      CX.fun();
   }
}
```

```
class Util
{
   static void fun()
   {
      ....
   }
}
```

```
class CA
{
  Int I;
  String s;
  void f()
}
```

```
Class Factory
{
  createCAWithX() { … }
  createCAForGuest(int x, int y) { … }
  createForBla(string s) { … }
  …
}
```

Lot of composition

```
class CA
{
    Connection c;
    Transaction t; ?
    Command cmd;?
    Reader reader; ?

}
```

```
class CABuilder
{
    AddConnec
   …

}
```