Mubarak

» Skan.ai - chief Architect

» Ai.robotics - chief Architect

» Genpact - solution Architect

» Welldoc - chief Architect

» Microsoft

» Mercedes

» Siemens

» Honeywell

Mubarak

Agenda

- Cyclomatic Complexity
- Cohesion
- Coupling
- Composition
- Expectations
- Years of Exp
- Technology stack
- Domain

	Cpu cycle
A +. B	5
add(10,20);	8
CreatThread	2,00,000
Destroy thread	1,00,000
Con.open(); stm.execute(“….”); con.close();	40,00,000

Architecture vs Design

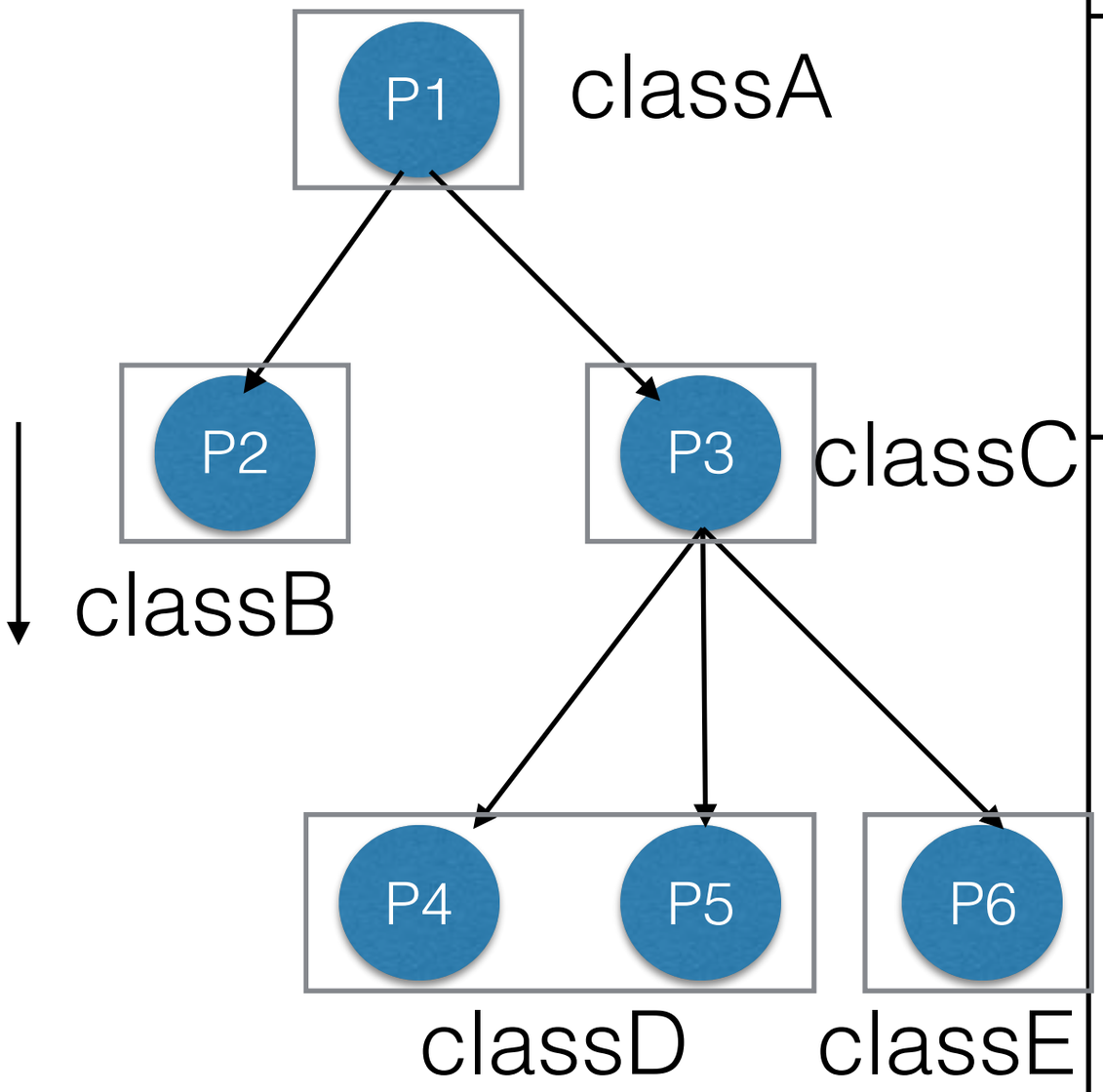
Quality

- Security (trust)
- Availability
- Scalability
- Performance
- Maintainability
- Reliability(trust)
- Robustness (rugud)
- Portability
- Usability
- Interoperability

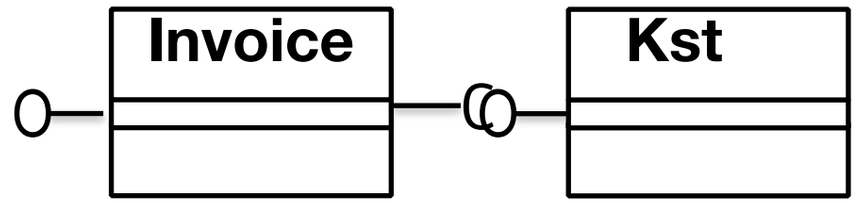
Approach

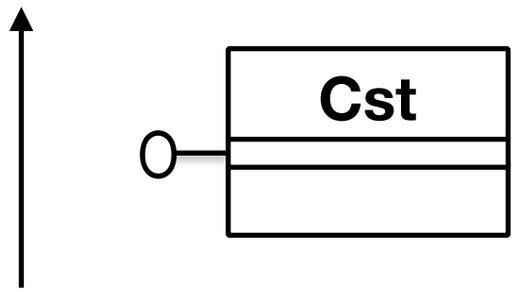
- Parallel
- Caching
- GC
- Lazy loading
- Load Balancing
- Unit Test
- Monitoring and Alerts
- Doc
- Transaction
-

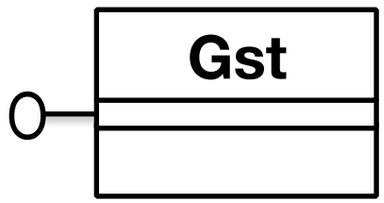
Procedural Prog (tree)



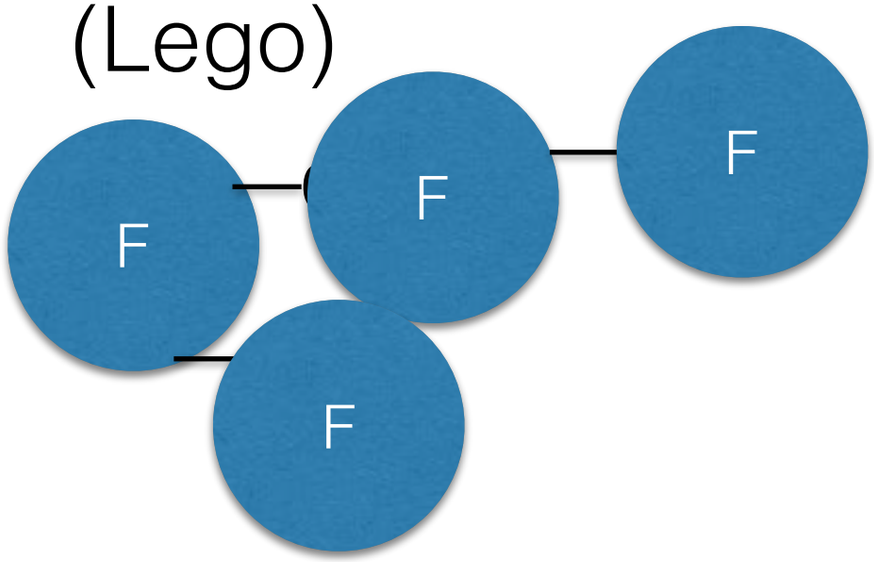
OO Prog (Lego)

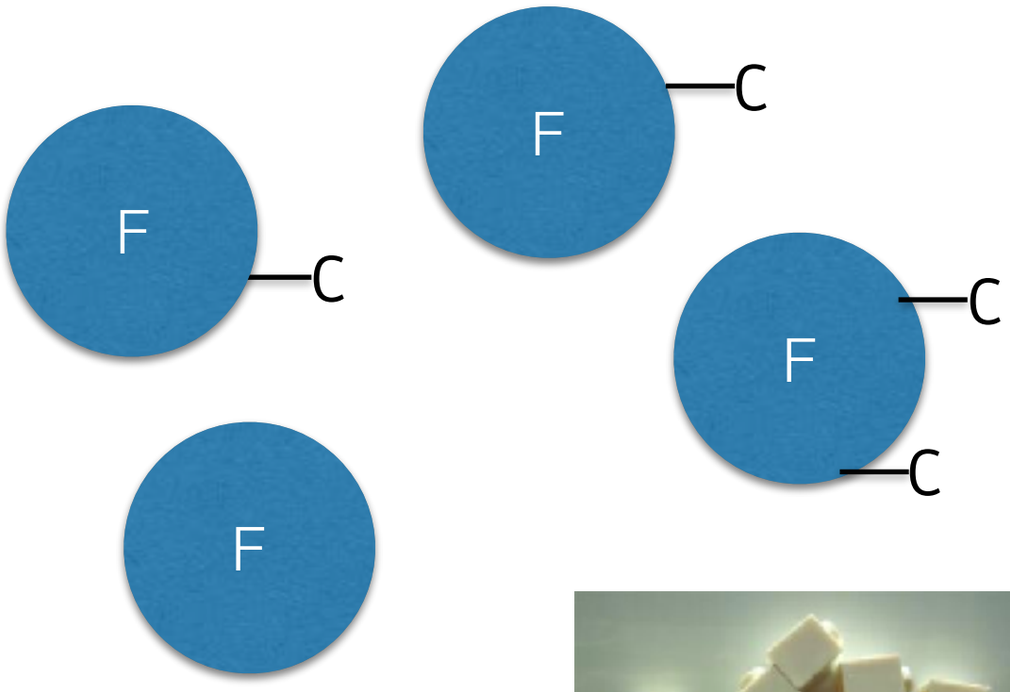




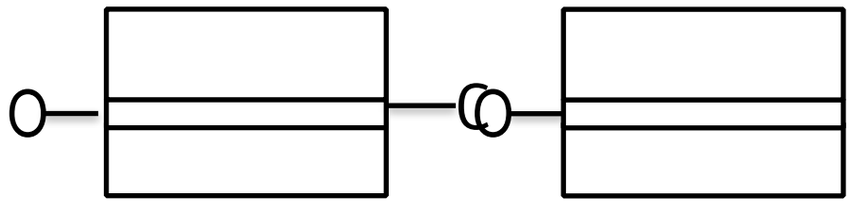


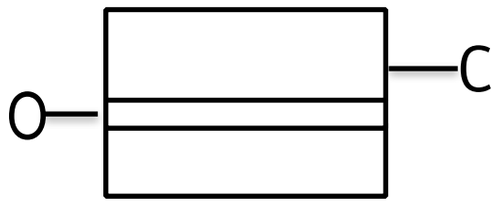
Functional Prog (Lego)

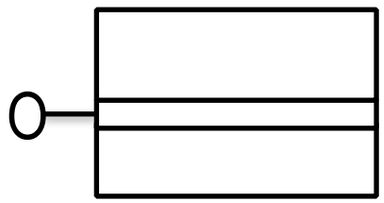


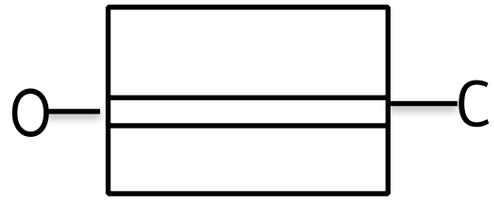


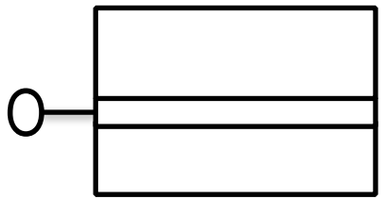
OO Prog (Lego)





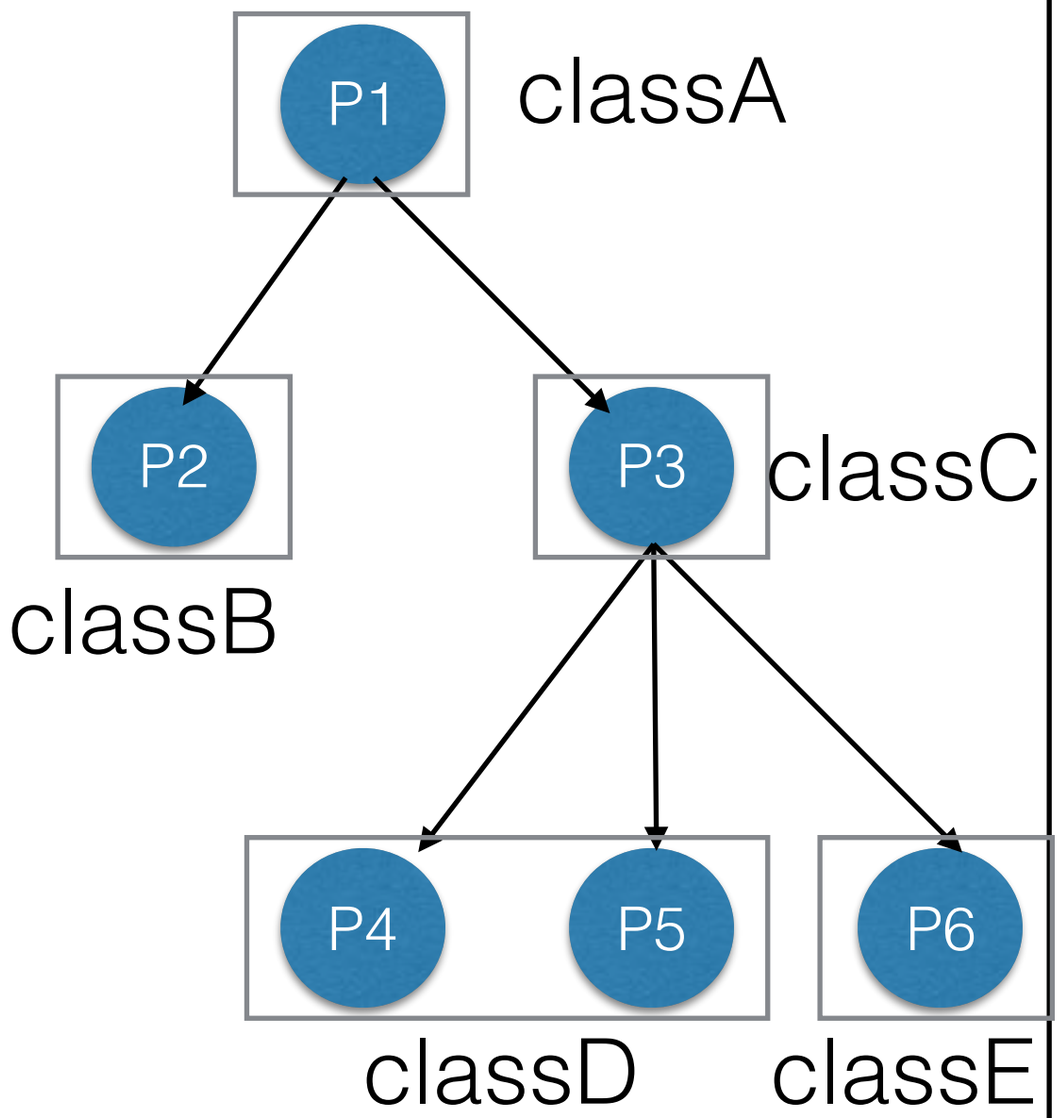






Procedural Prog

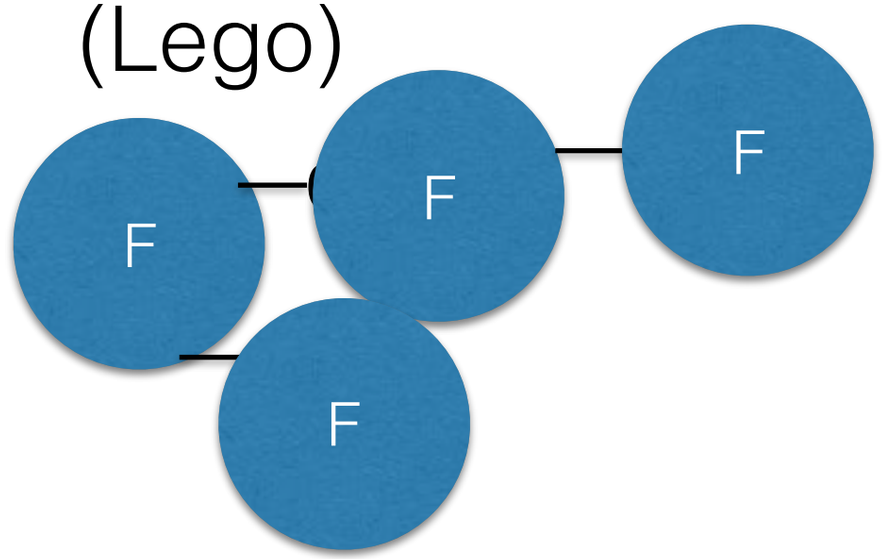
(tree)

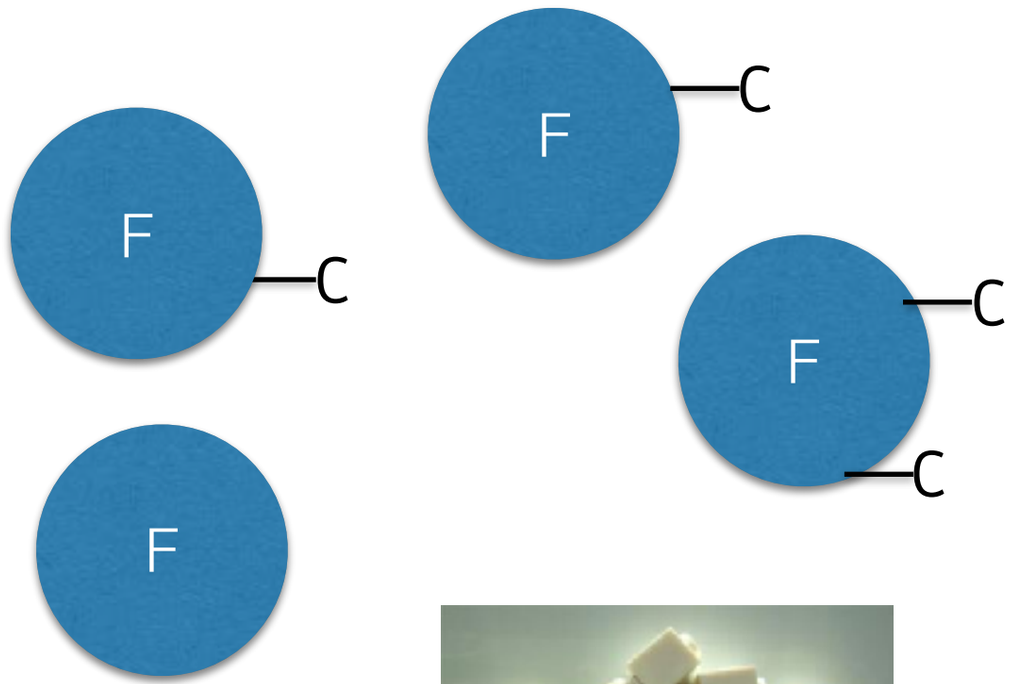


(top down)

Functional Prog

(Lego)





Polymorphism

?

logic1()
logic2()
logic3()

Single Dispatching
 # Interface, visitor

?

logic1()
logic2()
logic3()

Dual Dispatching
 # Same family : lookup
 # Different family : Visitor

?

logic1()
logic2()
logic3()

Multi Dispatching
 # Lookup

Single dispatch - virtual fun

```
class CA{
   void fun(){ //1
      ....
   }
}

class CB extends CA{
   void fun(){ //2
      ....
   }
}

class CC extends CB{
   void fun(){ //3
      ....
   }
}
```

```
void do(CA a)
{
   a.fun();  //1 | 2 | 3
}
```

Single dispatch - delegate

```
class Util {
  void fun(CA){ //1
    ....
  }
  void fun(CB){ //2
    ....
  }
  void fun(CC){ //3
    ....
  }
}
```

```
class CA{
  void fun(){ //1
    Util u = new Util();
    U.fun(this);
  }
}

class CB extends CA{
  void fun(){ //2
    Util u = new Util();
    U.fun(this);
  }
}

class CC extends CB{
  void fun(){ //3
    Util u = new Util();
    U.fun(this);
  }
}
```

When logic cannot be kept in the Family

```
void do(CA a)
{
  a.fun();  //1 | 2 | 3
}
```

Single dispatch - visitor

```
interface Visitor{
    void visit(CA);
    void visit(CB);
    void visit(CC);
}
class CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CB extends CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CC extends CB{
   void accept(Visitor v){
     v.visit(this);
   }
}
```

When logic cannot be kept in the Family and also cannot couple to the Delegated class

```
class LogicImp implements Visitor{
    void visit(CA) { } //1
    void visit(CB) { } //2
    void visit(CC) { } //3
  }
```

```
void do(CA a)
{
   LogicImp obj = new LogicImp();
   a.accept(obj);  //1 | 2 | 3
}
```

Dual dispatch - visitor

```
interface Visitor{
    void visit(CA);
    void visit(CB);
    void visit(CC);
}
class CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CB extends CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CC extends CB{
   void accept(Visitor v){
     v.visit(this);
   }
}
```

```
void do(CA a,CX x){
   a.accept(x);  //1 — 9
}
```

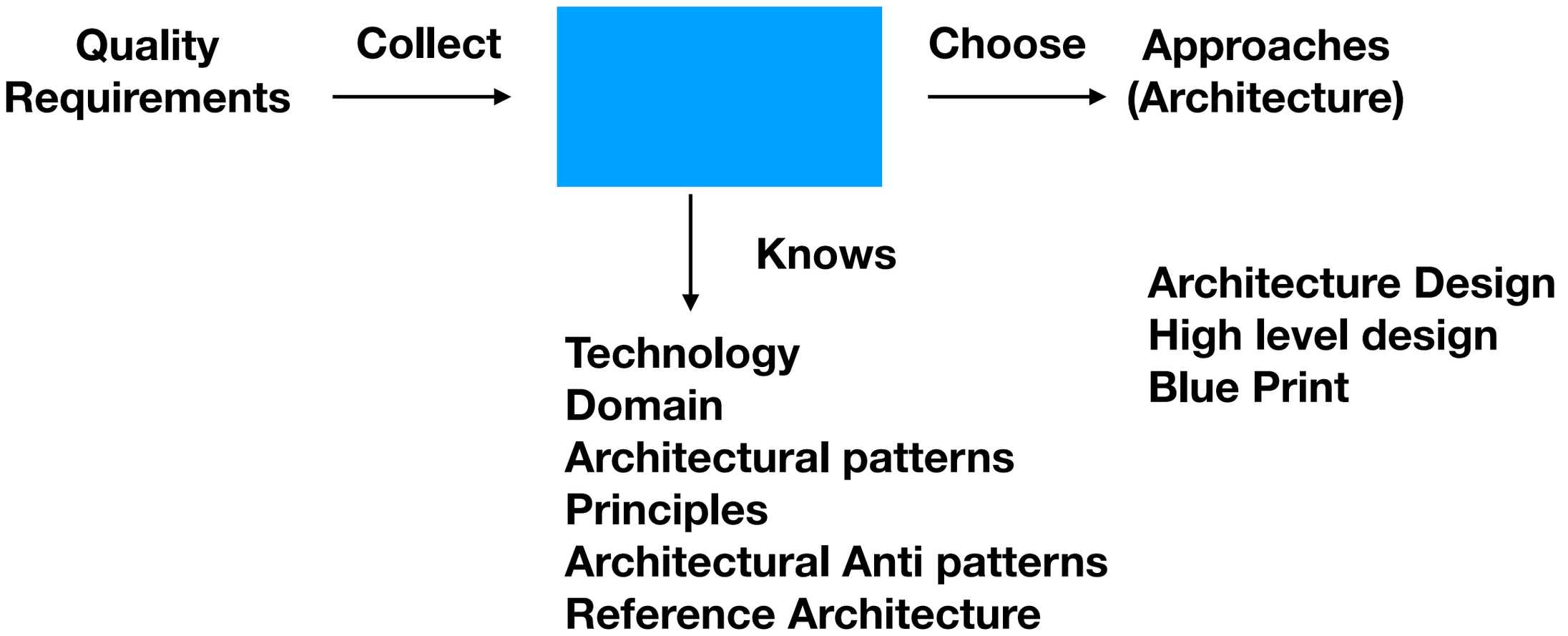
```
class CX implements Visitor{
    void visit(CA) { } //1
    void visit(CB) { } //2
    void visit(CC) { } //3
  }
}
class CY implements Visitor{
    void visit(CA) { } //4
    void visit(CB) { } //5
    void visit(CC) { } //6
  }
}
class CZ implements Visitor{
    void visit(CA) { } //7
    void visit(CB) { } //8
    void visit(CC) { } //9
  }
}
```

Dual dispatch

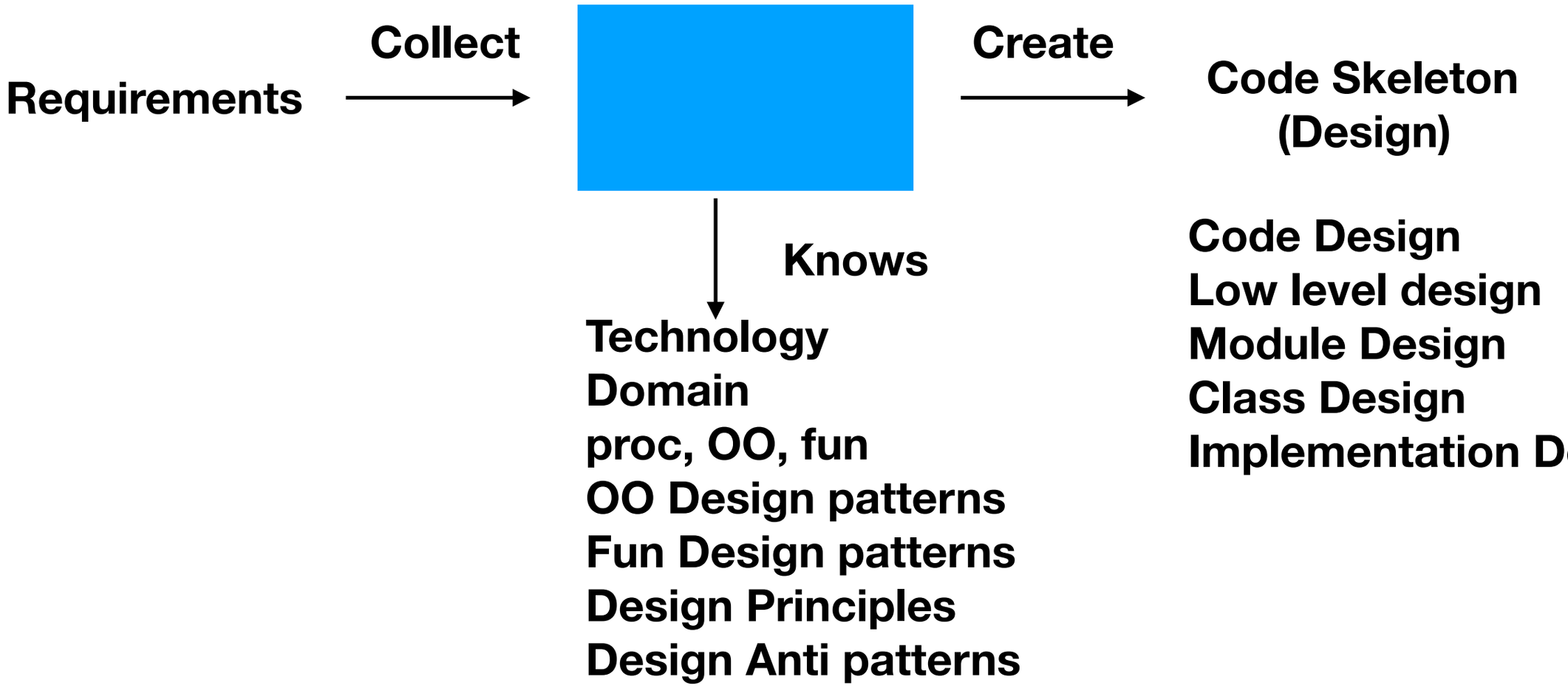
```
class CA{
  }

class CB extends CA{
 }

class CC extends CB{
  }
```

```
void do(CA a1,CA a2){
   Lookup
  }
```

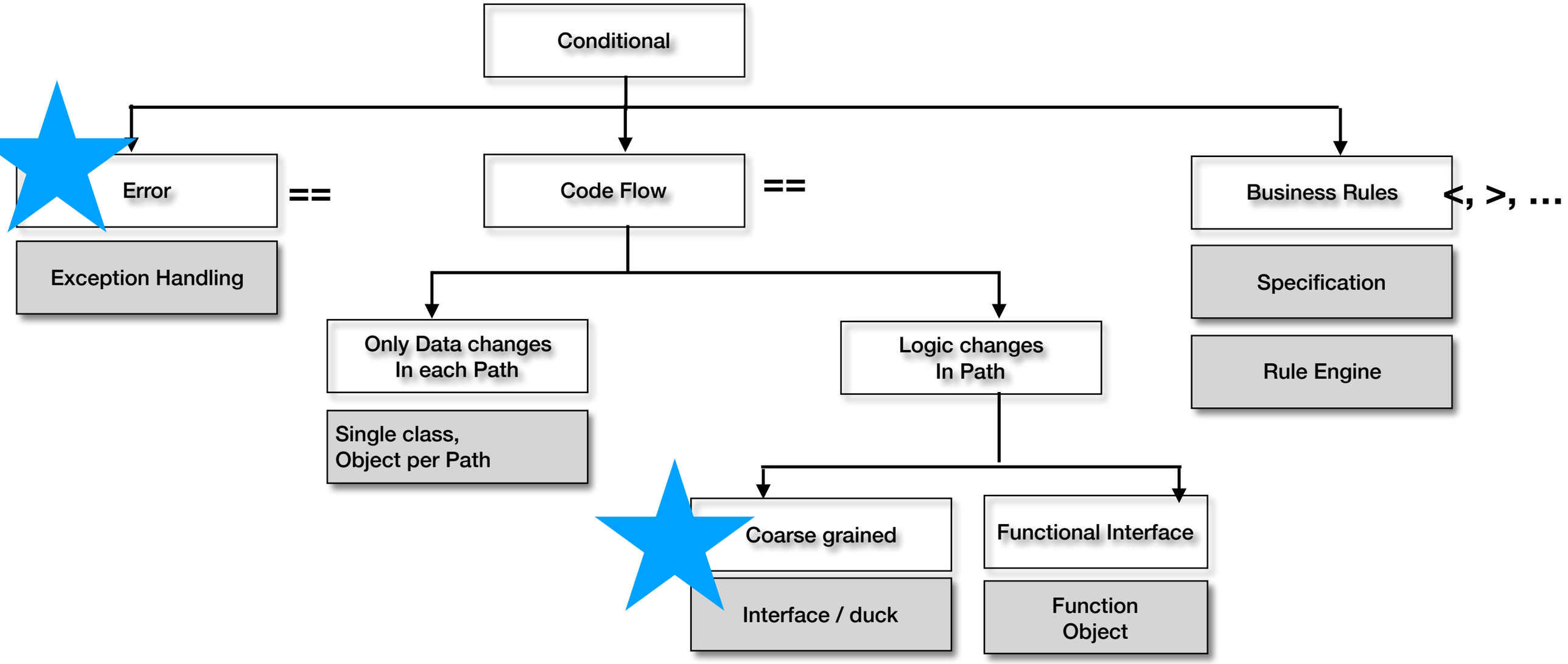
System Quality



Code Maintainability



Coding Style	Proc	OO	Fun
Performance	-	-	+ +
Security	-	-	-
Unit Testability	- -	+ +	+ + +
Time to develop/ cost	+ +	- -	+
Manage Large Code	- -	+ +	+
Learning Curve	+ +	- -	-
Language	C, java, js, py	Java, js, py	Java, js, py, scala, kotlin, Haskel


Conditional
Error
==
Exception Handling
Code Flow
==
Only Data changes
In each Path
Single class,
Object per Path
Logic changes
In Path
Coarse grained
Interface / duck
Functional Interface
Function
Object
Business Rules
<, >, ...
Specification
Rule Engine

Manage
Cyclomatic complexity

Error Flag

Exception Handling

Flow Flag

==

Business Rules
Flow

Specification

Rule Engine

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 SRP &
coupling

Visitor

dual dispatch

2 objects belonging to
different families

Visitor

2 objects of
same family

Use Lookup

multi dispatch

Decision Table

Coupling

	Method Call	Class Instantiation	Object Deallocation
Example	obj.fun();	new CA();	delete obj;
Approach1	Interface	DI	Virtual destructor
Approach 2	Lamda	Factory	
Approach 3			

Principles

- SRP (***)
  - Library max : 30 Avg : 15 classes
  - Class Max: 12 Avg : 4 public methods
  - Method Max: Fit Screen Avg : 5 lines
- LSP
- Low Cyclomatic Complexity (< 10)
- Program to an interface (upcasting)
- Prefer Composition over Inheritance
- OCP
- SOC
- DRY (*)
- KISS
- YAGNI
- DIP
- Low Coupling (**)
- Uni directional Coupling
- Boundary Control Entity
- Agregate Root

Anti patterns

- Flag —> interface or duck typing, exceptions. Lamda
- Type Check
- Down casting
- Arrow code
- Bool, Null, int for error Handling
- Swiss knife/ God Class
- Functional Interface (single method interface)<— lilliputs
- Bi directional coupling
- * to * coupling

- Over loading family of classes
- Static methods
- Inheritance (extends)

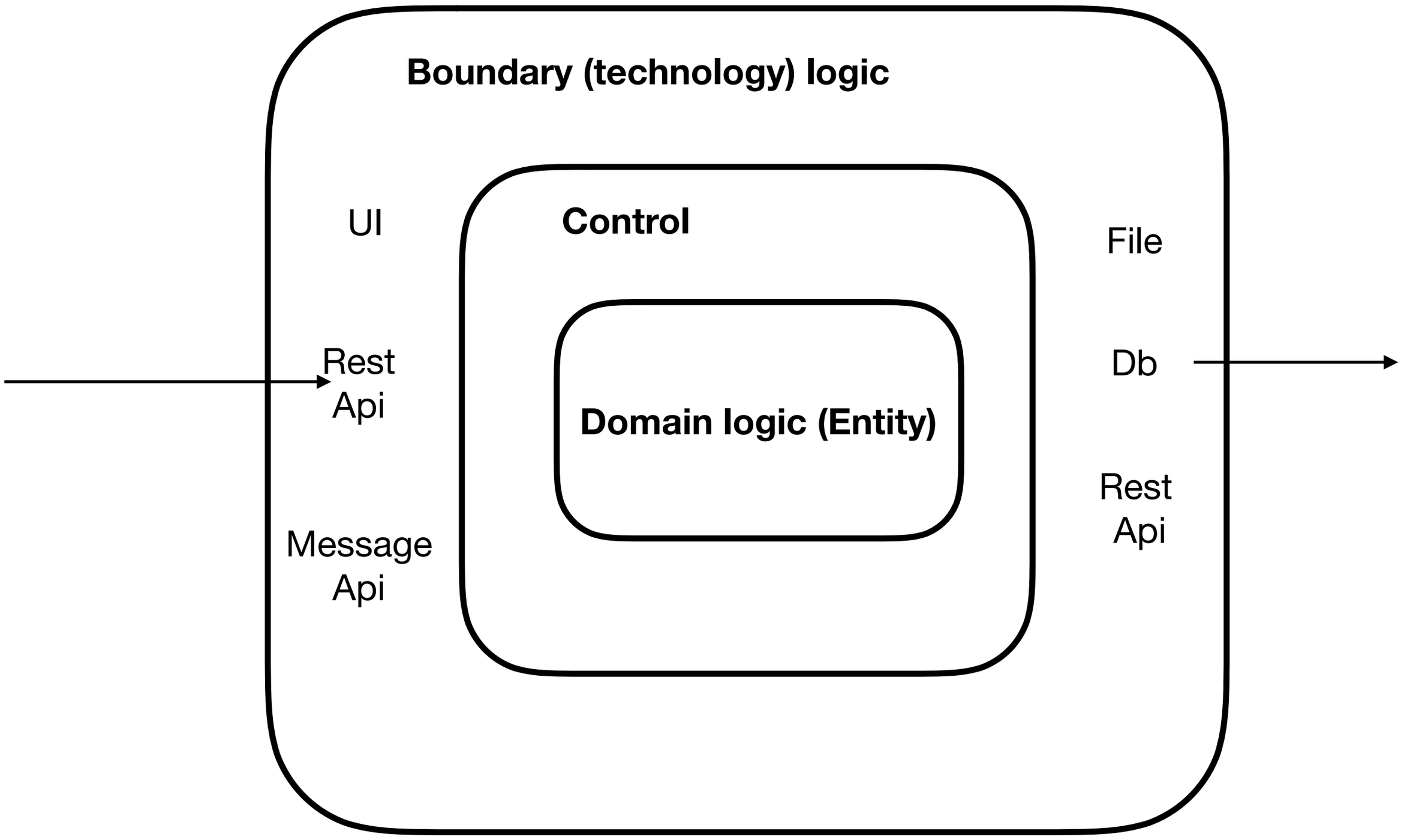
	Inheritance	Ref
Change the parent At runtime	No	Yes
Lazy Load Parent	No	Yes
Multiple Parent	No	Yes

Seperation of Concerns

- Error handling logic should not be mixed with domain logic
- Security
- Transaction
- Persistence

SOC

- Boundary (technology) and Domain
- Error logic and Domain
- Separate rules from Logic
-


Boundary (technology) logic
UI
Control
File
Rest
Api
Domain logic (Entity)
Db
Rest
Api
Message
Api

Boundary (technology logic)

Control (flow logic)

Entities (Domain logic)

Aggregates

Root, Entities, VO

Aggregates

Root, Entities, VO

Aggregates

Root, Entities, VO

API Boundary

Control

Domain

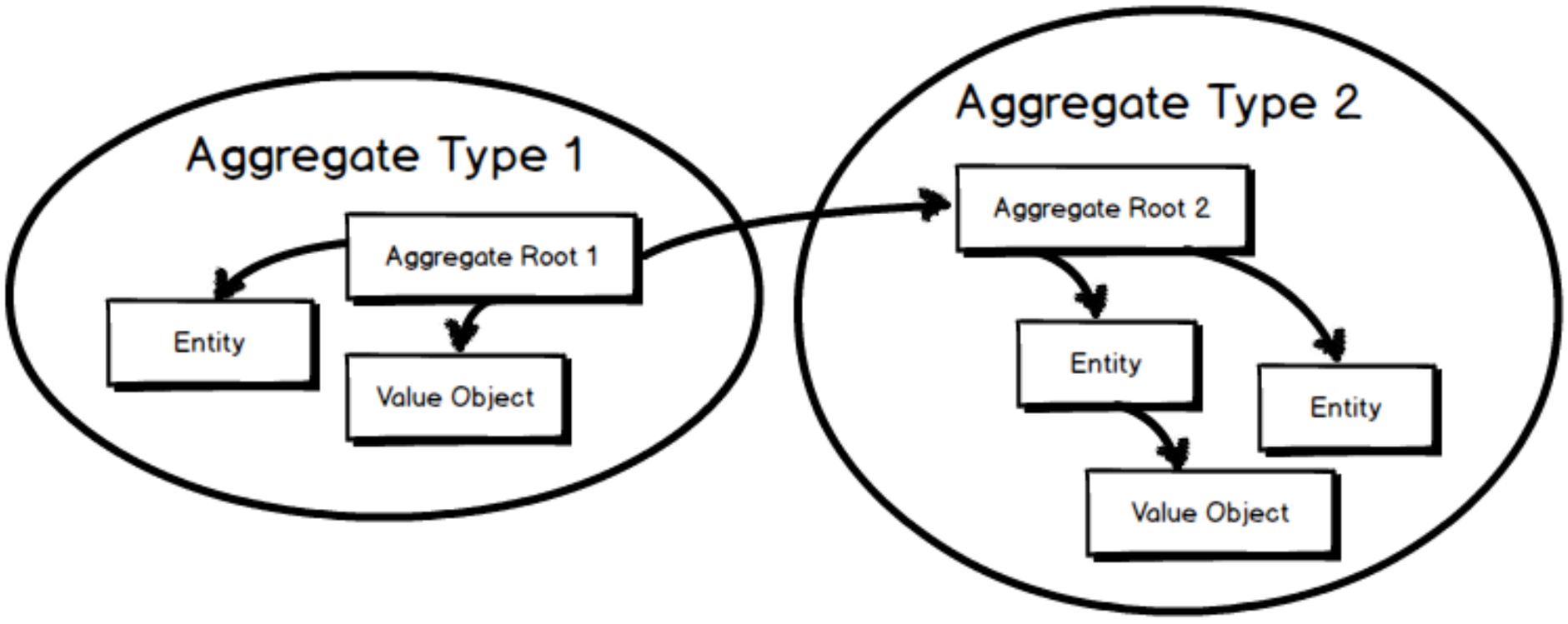
Dao Boundary

API Boundary

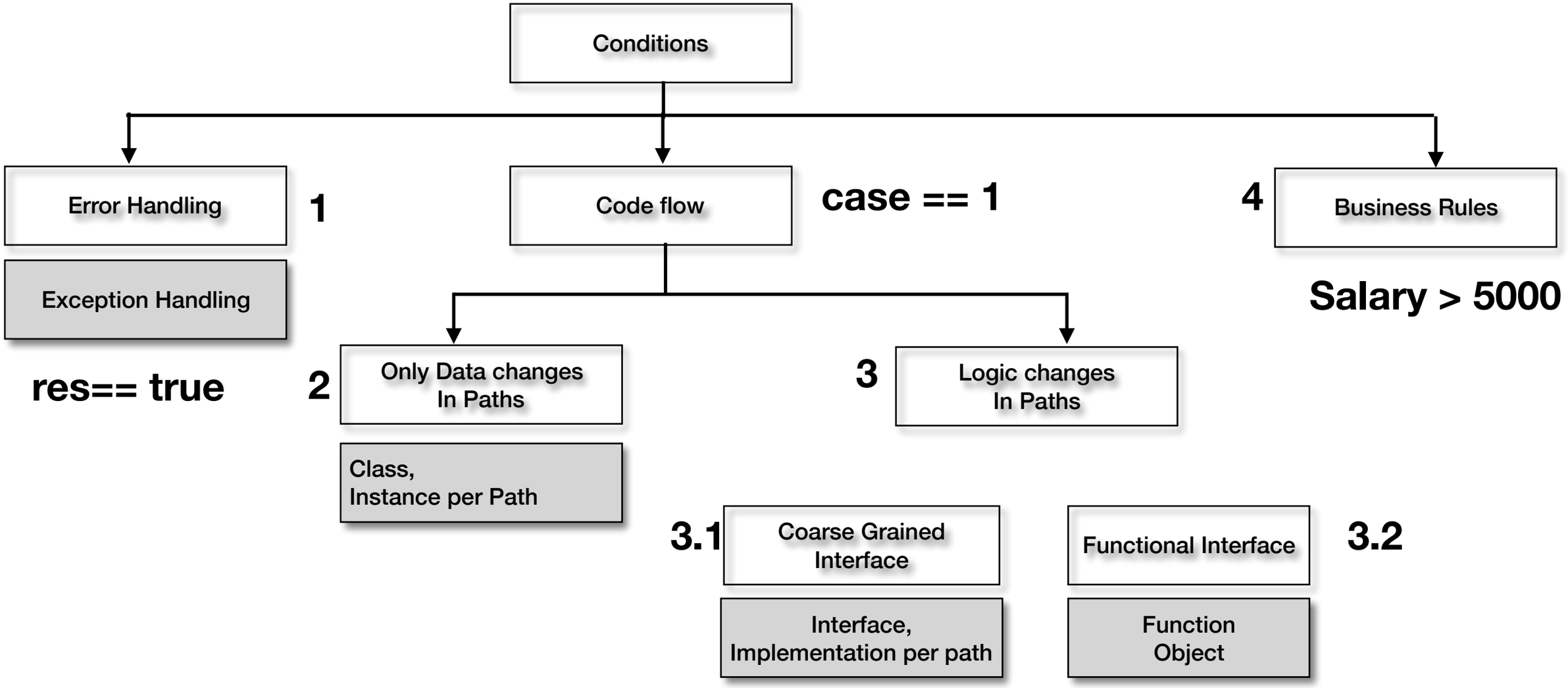
Domain

Dao Boundary

Aggregate Root (DDD)



Invoice
-> LineItem
-> Address


Conditions
Error Handling
1
Code flow
case == 1
4
Business Rules
Exception Handling
Salary > 5000
res== true
2
Only Data changes
In Paths
3
Logic changes
In Paths
Class,
Instance per Path
3.1
Coarse Grained
Interface
Functional Interface
3.2
Interface,
Implementation per path
Function
Object

Manage
Cyclomatic complexity

Error Flag

Exception Handling

Flow Flag

==

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

dual dispatch

2 objects belonging to
different families

Visitor

2 objects of
same family

Use Lookup

Business Rules
Flow

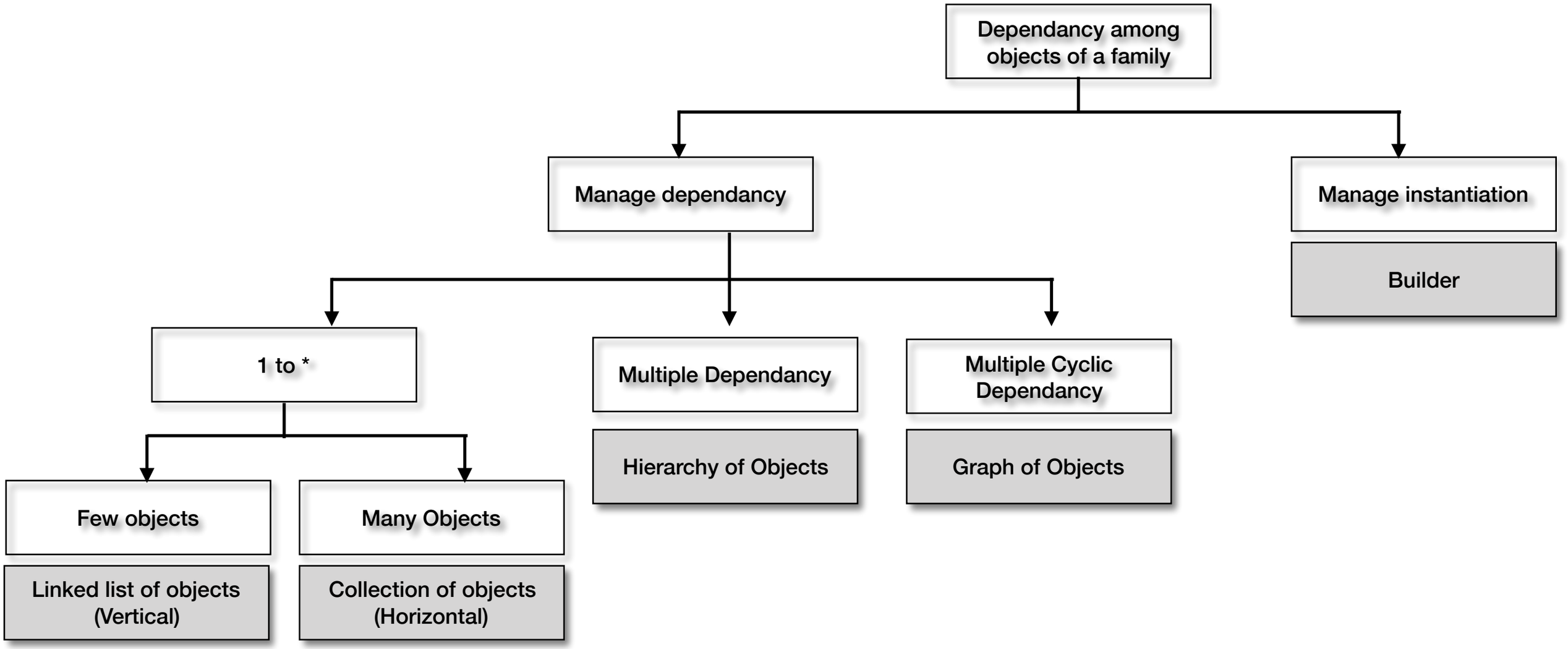
Specification

Rule Engine

multi dispatch

Decision Table

- 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)

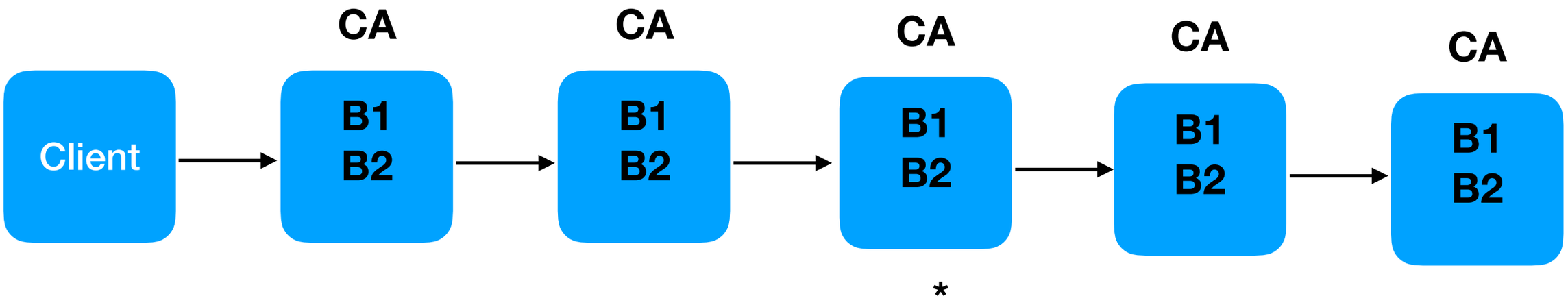
Manage Cohesion

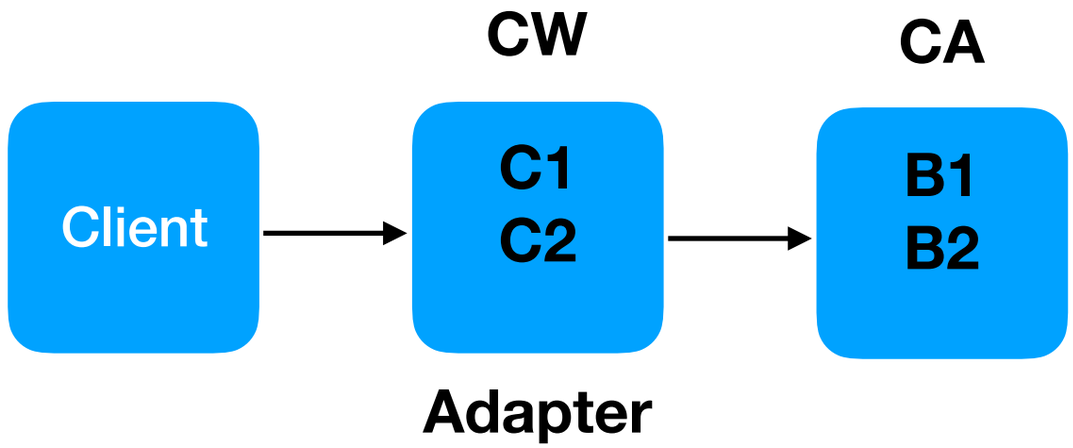
Separate Technology Code	Separate Cross Cutting Logic	Separate Business Rules	Separate Error Handling	Separate Read/Write Logic	Separate Flow and Steps	Unrelated Logic
UI Database File API EMail Messaging	Exception handling Caching Log Transaction Authorization	If sal > 5000	If res == false			Things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

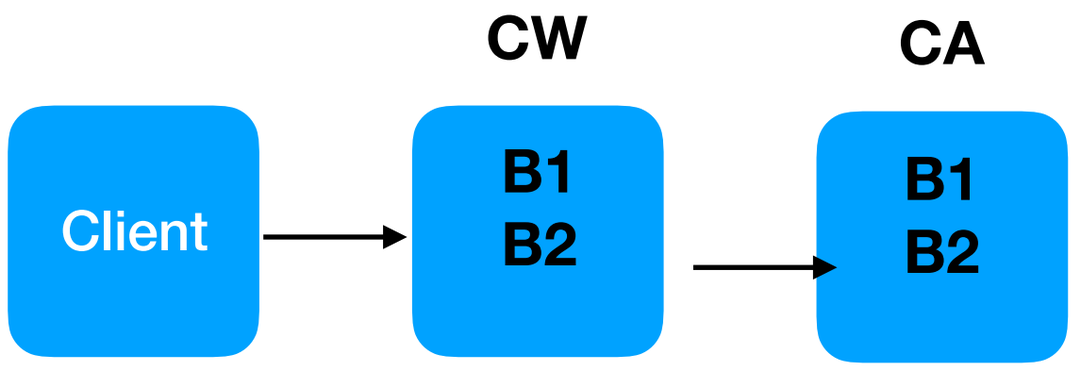
Layered Design	Facade	Specification	Exception Handling	CQS	Facade
Boundary Control Entity	Decorator	Rule Engine			
Hexagonal Arch	AOP				
Pipes Filter					

	Proc (tree)	OO (lego)	Fun
Lang	C, py, java, C#, JS, c++	Java, C#, C++, py, js	py,js, J8,c#
Constructs	if/switch/goto/ Static methods	Polymorphism/ Exceptions	High order fun/ recursion/ closure
Performance	-	-	+ +
Security	-	-	-
Learning Curve	+ +	- -	-
Development Time	+ +	- -	+
Unit Test	- -	+	+ +
Code Maintainability/ Support Time	- -	+ +	+

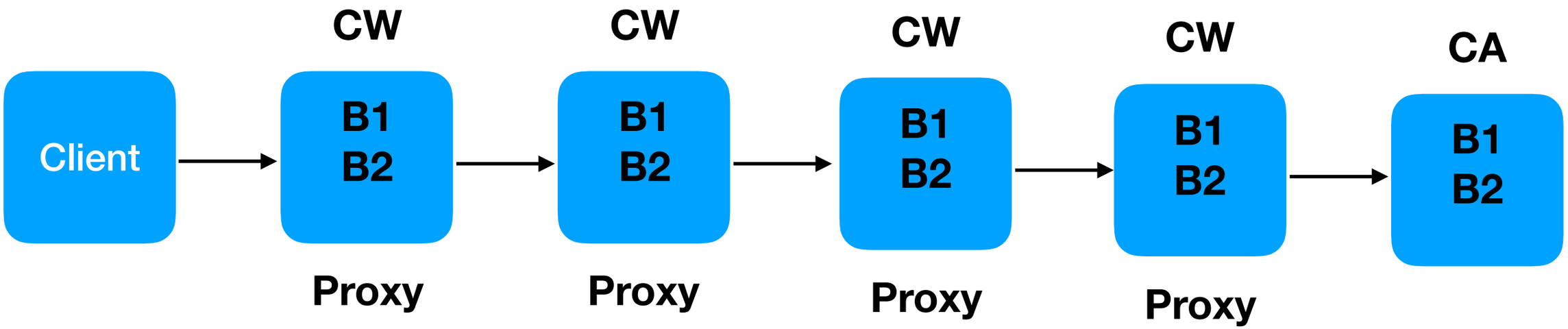
Change behaviour
Change Part of behaviour
Rename behaviour
Enrich behaviour
Add behaviour





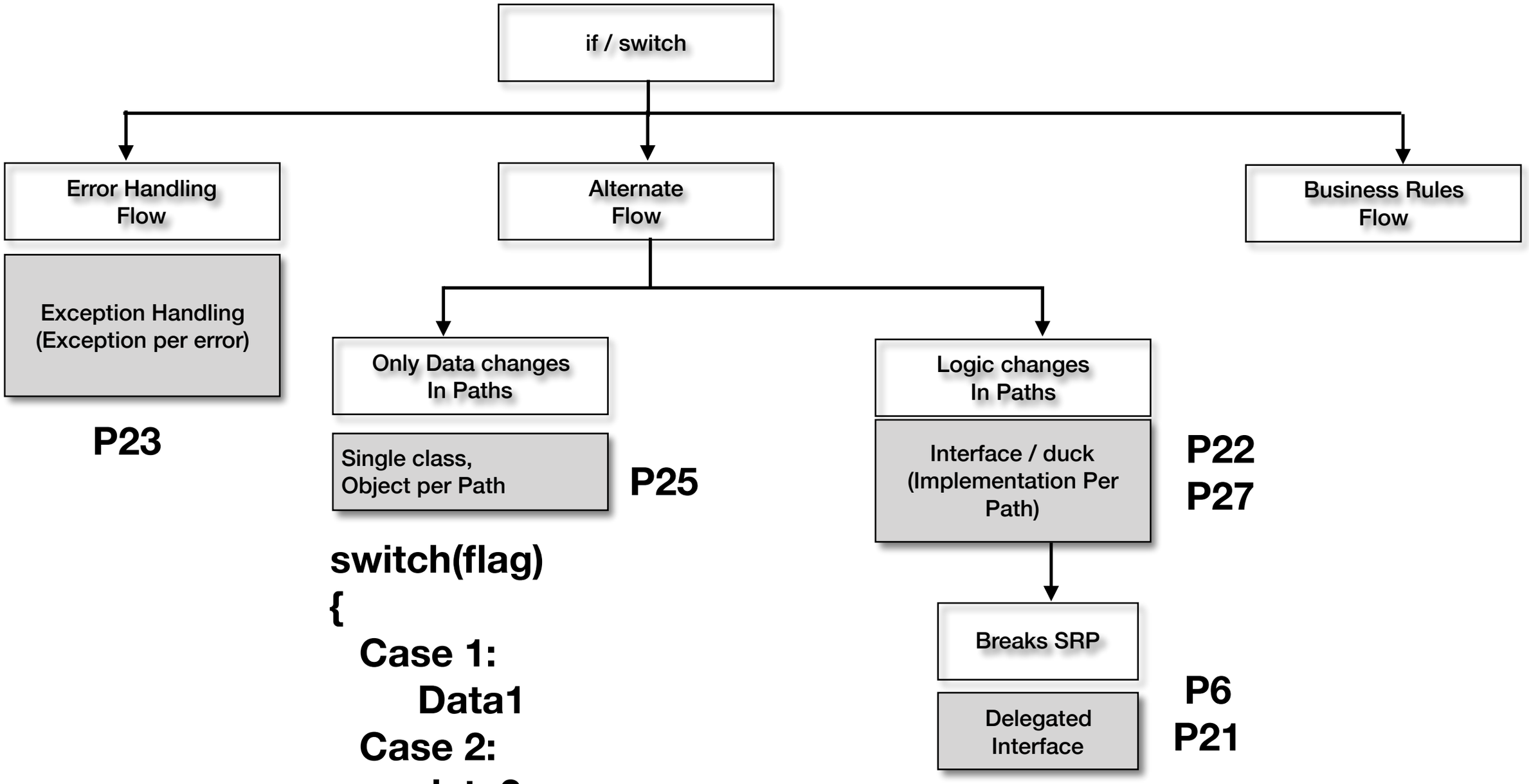


Authentication
Autorization
Audit
Cache
Exception Handling
Lazy loading
Simplify network call



Separation of Concern

- Domain logic and error handling logic
- Domain logic and technology logic




```
switch(flag)
{
  Case 1:
      Data1
  Case 2:
      data2
  Case 3:
      Data3
}
```

```
switch(flag)
{
  Case 1:
      Logic1 amount * 0.5;
  Case 2:
      Logic2 (amount - 5000) * 0.08;
  Case 3:
      Logic3
}
```

Method Call	Instantiation	Deallocation
ca.f1();	new CA()	delete ca;
Interface	Di # constructor #setter	GC
Lamda	Factory # Class Factory # Abstract Factory # factory Method # creator Method	Virtual destructor
Duck typing		
Adapter		

Tight coupling	Interface typing (java)	Duck typing (py, js)	Lamda (py,js, java)

class CA
{
  void f1(){
  …
  }
}
	interface IA{
	  void f1();
	}
	
	class CA implements IA
	{
	  void f1(){
	  …
	  }
	}
		class CA{
		  void f1(){
		  …
		  }
		}
			class CA{
			  void f1(){
			  …
			  }
			}

do(CA obj)
{
  obj.f1();
}
	do(IA obj)
	{
	  obj.f1();
	}
		do(obj)
		{
		  obj.f1();
		}
			do(Lamda fun)
			{
			  fun();
			}

do(new CA( ))	do(new CA( ))	do(new CA( ))	CA obj = new CA( ) do(()-> obj.f1())

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

vptr

?

Runtime

Stack

a

x

x.f(a)
x.vptr.vtbl[0](a)

Compile time

- 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
-

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());

“Change logic
In a Class”

```
class CA
{
  void fun(){
      … logic 1
      … logic 2 *
      … logic 3
  }
}
```

Composition
<<stratergy>>

```
interface IX{
  void logic2();
}

class CA
{
  IX ref;

  void fun(){
      … logic 1
      ref.logic2();
      … logic 3
  }
}

class CB implements  IX{
  void logic2(){ …. }
}

CA o = new CA (new CB() );
o.fun();
```

Lamda

```
class CA
{
  Lamda fun;

  void fun(){
      … logic 1
      fun();
      … logic 3
  }
}
```

Inheritance
<<template method>>

```
class CA
{
  IX ref;

  void logic2() { … }

  void fun(){
      … logic 1
      logic2();
      … logic 3
  }
}

class CB extends CA{
  void logic2(){ …. }
}

CA o = new CB();
o.fun();
```

“enrich logic
In a Class”

```
class CA
{
  void fun(){
       … logic
  }
}
```

Composition
<<proxy>>

```
class wrapper
{
  CA ref;

  void fun(){
       … enrich
       ref.fun();
       … enrich
  }
}

  w->CA
```

Composition
<<Decorator>>

```
Interface IX{
  void fun();
}
class CA implements IX
{
  void fun(){
       … logic
  }
}

class wrapper implements IX
{
  IX ref;

  void fun(){
       … enrich
       ref.fun();
       … enrich
  }
}
```

w->w->w->w->w->CA

Lamda
<<Currying>>

Composition
<<Decorator>>

```
Interface IX{
  void fun();
}
class CA implements IX
{
   void fun(){
        … logic
   }
  }

class wrapper implements IX
{
   IX ref;

   void fun(){
        … enrich
        ref.fun();
        … enrich
   }
}
```

w->w->w->w->w->CA

Composition
<<COR>>

```
Interface IX{
  void fun();
}

class CA implements IX
{
   IX ref;

   void fun(){
       if(cond)
        … logic //handle
       else
        ref.fun(); //delegate
       }
}
```

CA->CA->CA->CA->CA

“change interface Of Class”

```
Interface IX{
  void fun();
}
class CA implements IX
{
  void fun(){
      … logic
  }
}
```

<<Composition>>
Adapter

```
Interface IY{
  void fun2();
}
class Wrapper implements IY
{
   CA ref;

   void fun2(){
       ref.fun();
   }
}
```

```
IY y = new Wrapper(new CA);
Y.fun2();
```

Single dispatch - visitor

```
interface Visitor{
    void visit(CA);
    void visit(CB);
    void visit(CC);
}
class CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CB extends CA{
   void accept(Visitor v){
     v.visit(this);
   }
}

class CC extends CB{
   void accept(Visitor v){
     v.visit(this);
   }
}
```

When logic cannot be kept in the Family and also cannot couple to the Delegated class

```
class LogicImp implements Visitor{
    void visit(CA) { } //1
    void visit(CB) { } //2
    void visit(CC) { } //3
  }
```

```
void do(CA a)
{
   LogicImp obj = new LogicImp();
   a.accept(obj);  //1 | 2 | 3
}
```

“change logic
Based on state”

```
class CA
{
   data; <—state
   void fun(){
       if(data == x)
           … logic 1
       if(data == y)
           … logic 2
       if(data == z)
           … logic 3
   }
}
```


<<Composition>>
State

```
Interface State{
  void fun();
}
Class State1 implements State{
   void fun(){ ….. }
}
Class State2 implements State{
   void fun(){ ….. }
}
Class State3 implements State{
   void fun(){ ….. }
}
class CA {
   State ref = new State1();

   void fun(){
       ref.fun();
       changeState();
   }
}
```

<<Factory Method>>

```
class CA
{
   void fun(){
        … logic
   }
   CB createCB(){
       ….
    }

}
```

<<Creator Method>>

```
class CA
{
   void fun(){
        … logic
   }

   Static  CB createCB(){
       ….
    }
}
```

<<Class Factory>>

```
class Factory
{
    CA createCA(){
        ….
    }
   CB createCB(){
        ….
    }

}
```

<<builder>>

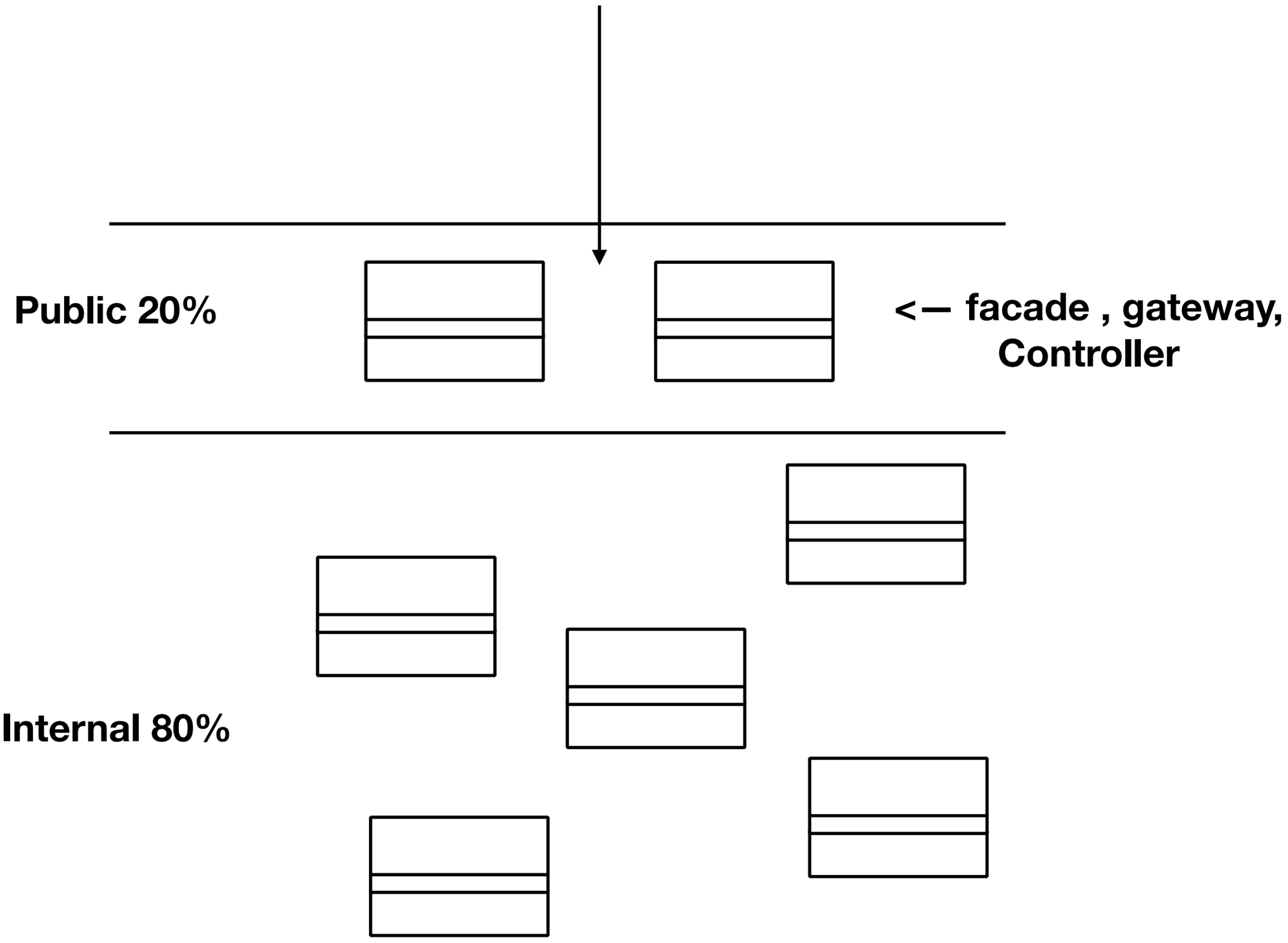
```
class Builder
{
    void addCA(){
        ….
    }
   void addCB(){
        ….
    }
    CX getCX(){
        ….
    }
}
```

<<Abstract Factory>>

```
Interface Factory{
    CA createCA();
    CB createCB();
}
class FactoryX
        implements Factory
{
    CA createCA(){
        ….
    }
    CB createCB(){
        ….
    }

}
```

<<Prototype>>

```
class CA
{
        CA clone(){
            ….
        }
}
```


Public 20%
<— facade , gateway, Controller
Internal 80%