Mubarak

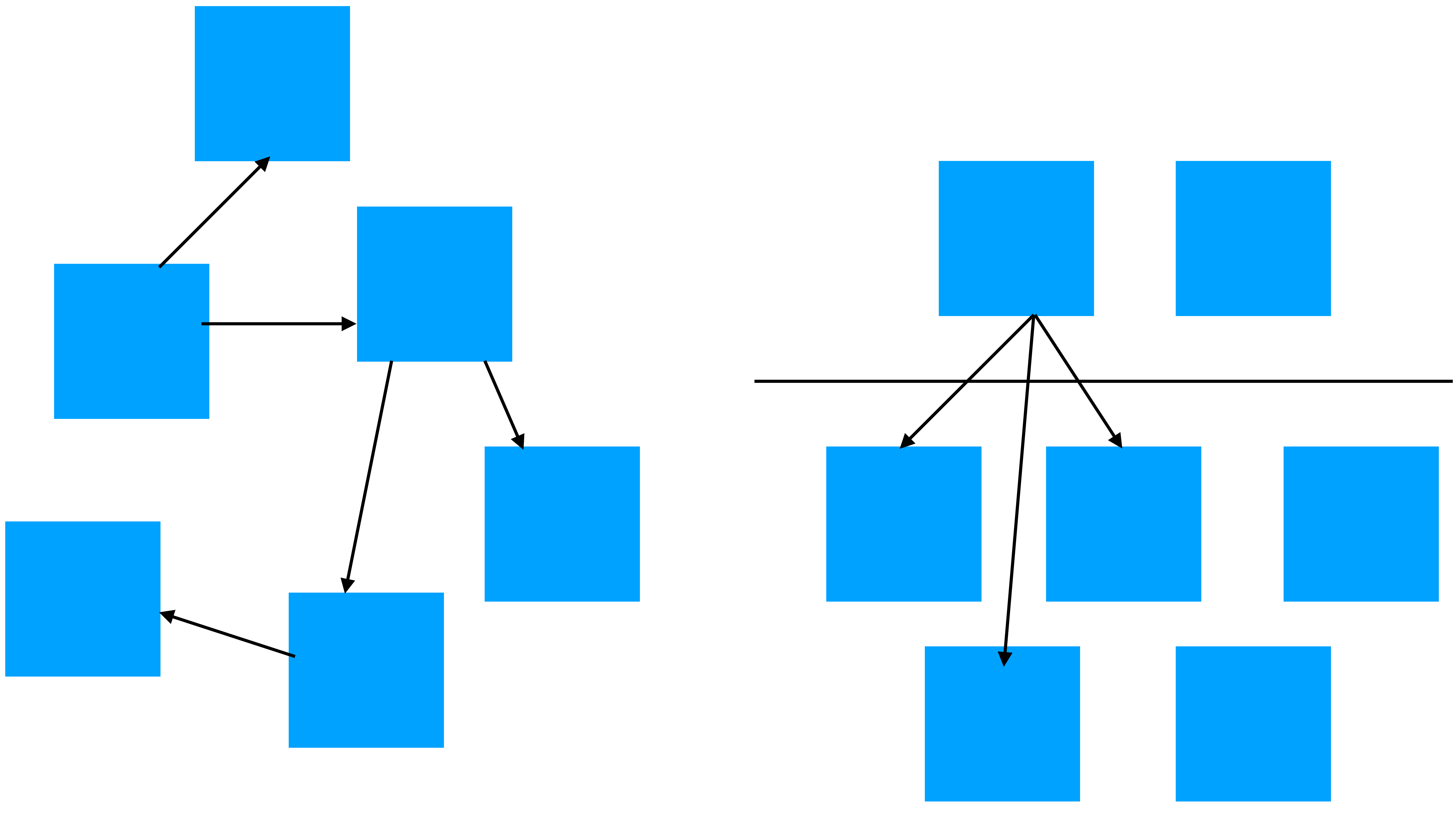
A LITTLE BIT ABOUT me...

» Skan.ai - chief Architect
» Ai.robotics - chief Architect
» Genpact - solution Architect
» Welldoc - chief Architect
» Microsoft
» Mercedes
» Siemens
» Honeywell

Mubarak

Agenda

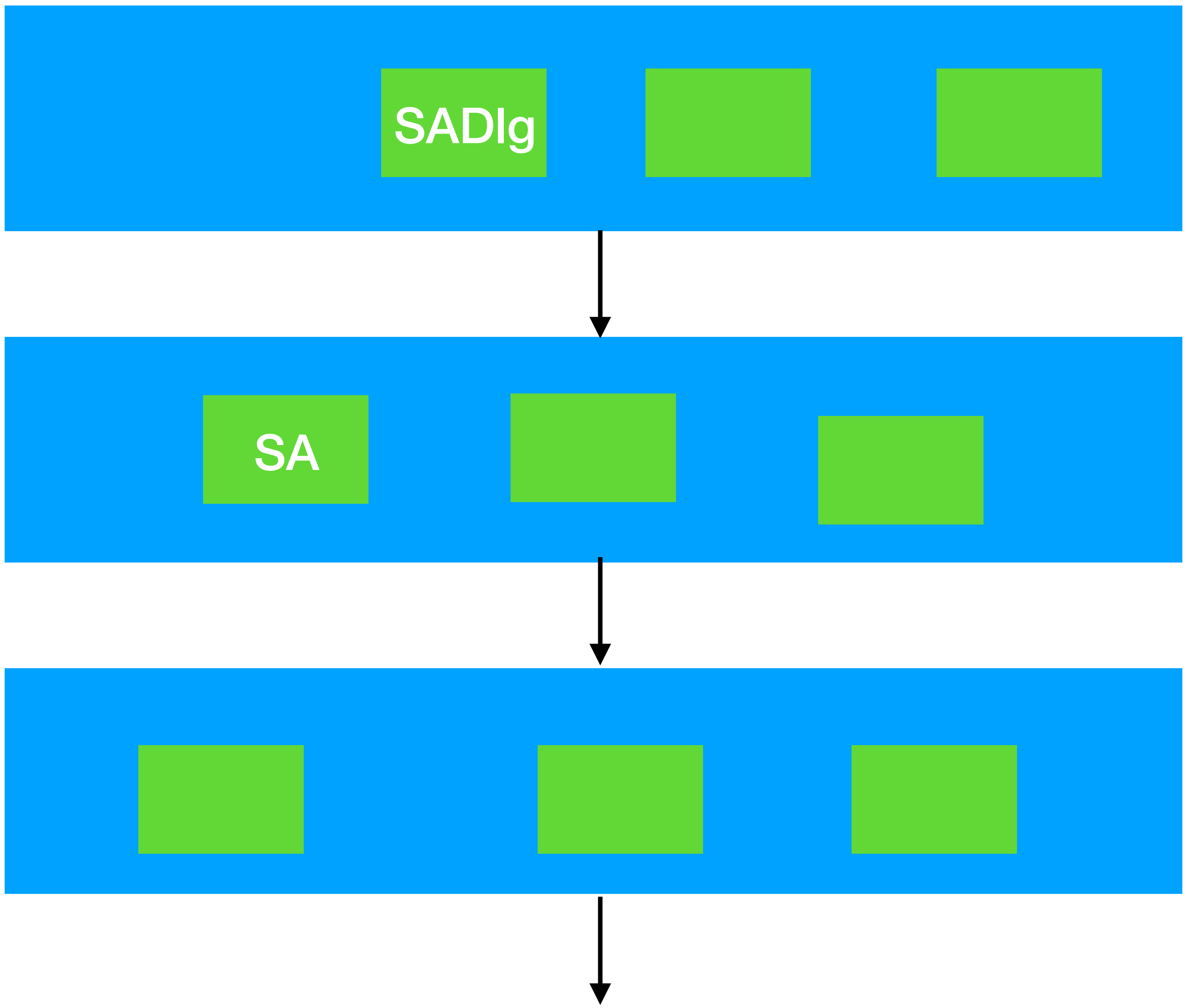
- **Cyclomatic Complexity**
- **Cohesion**
- **Coupling**
- Composition
- Expectations
- Years of Exp
- Technology stack

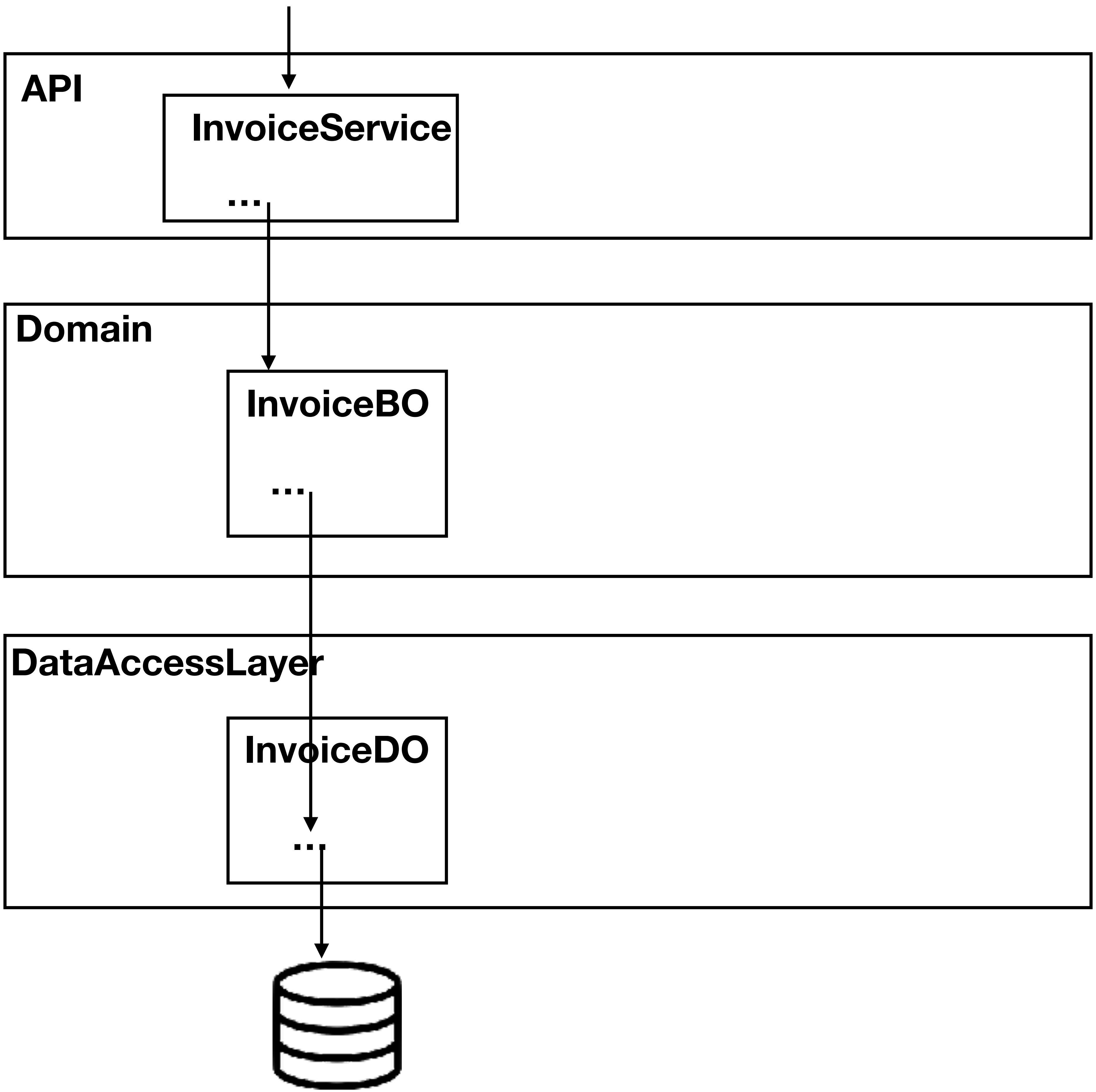


Good

- Program to an Interface
- Unit testable (**) <—
- Upcast
- Interface
- Exception Handling
- Low Coupling (**) <—
- KISS
- Prefer composition over inheritance

- LSP
- DRY (*)
- OCP <—
- POLA
- SRP (***) <—
- YAGNI
- DIP (*)


SADlg
SA


API
InvoiceService
...
Domain
InvoiceBO
...
DataAccessLayer
InvoiceDO
...

Bad

- Type Check -> Anti Abstraction
- Flag
- Cyclomatic Complexity
- Downcast
- Overloading family of types
- Magic strings /numbers
- Cyclic Coupling
  - Bi directional
  - * to *

- bool/ int/ null for error
- Arrow Code
- Swiss knife
- Static methods
- Functional Interface
- God Class

Coupling

- Uni directional (A->B) ~ok
- Low Coupling . Good
- Cyclic Coupling (A->B->C->A) ~Bad
    - Bi directional (A<->B)
    - * to *

- Util
- Handler
- Controller
- Manager
- Helper
- Service
- Executor
- Mediator
-

SRP

- Library
- Class
  - Good : 5 pub methods
  - Max : 12 pub methods
- Fun
  - Good : 6 lines
  - Max : Fit Screen

- Flag ==> Interface, EH, Lambda
- Coupling ==> Interface, Lambda
-

Review

- For vs foreach
- 2 + 3 : 4 cpu cycle
- Fun call : 10 cpu cycles
- Create thread : 200,000 cpu cycles <— kernel resource
- File write : 10,00, 000 cpu cycles <— disk i/o
- Db write : 40, 00, 000 cpu cycles <— network i/o

- Tiger tiger = new Tiger();
- Animal animal = tiger; <—up (abstraction)
- Tiger2 tiger2 = (Tiger) animal; <—down (anti abstraction)

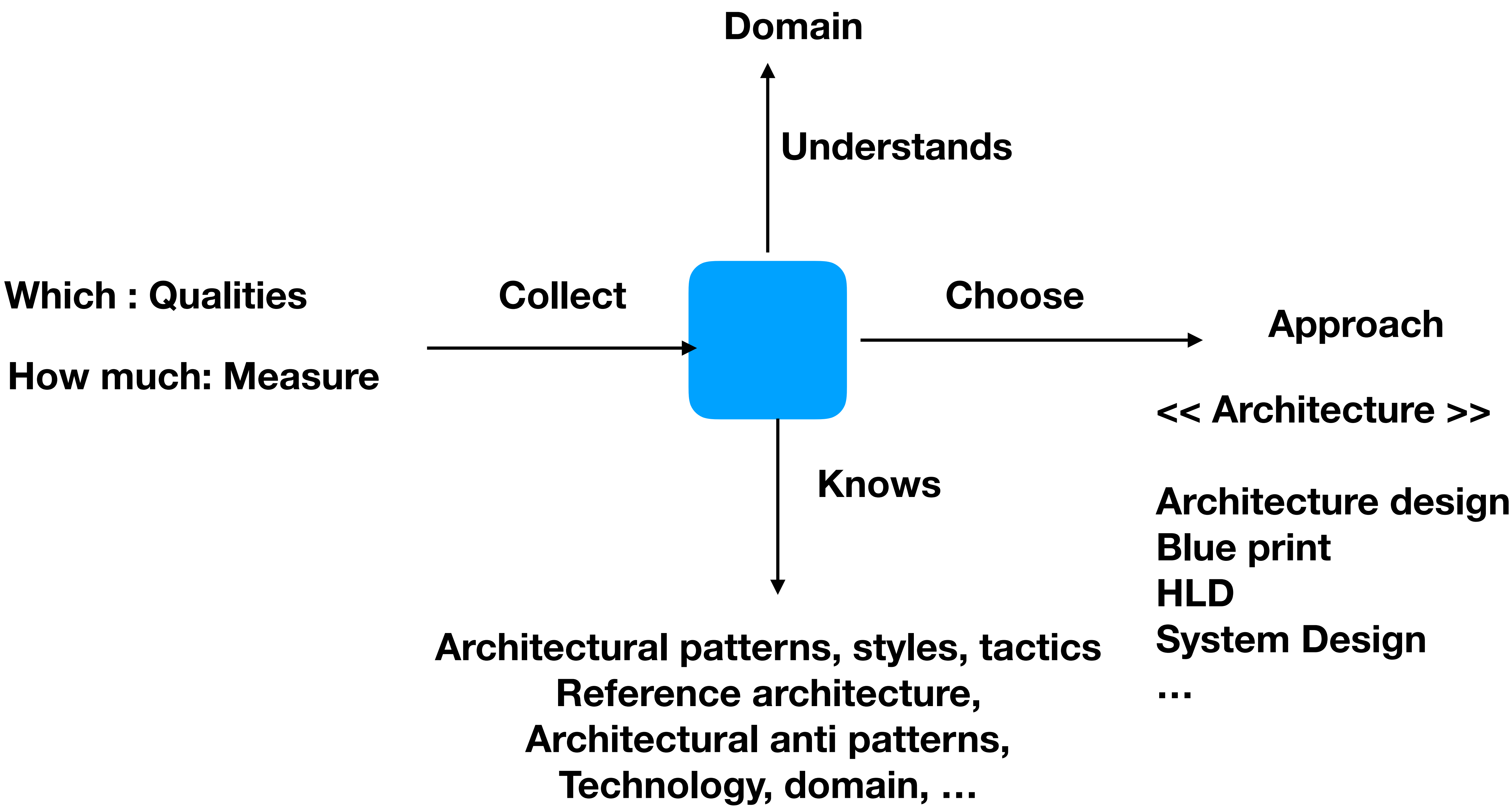
Architecture Design

vs

Code Design ★

Quality	Approach
Performance	Caching
Security	Indexing
Maintainability	Concurrency
Reliability	Pooling
Availability	Data Virtualization
Robustness	Lazy Loading
…	Reusability
	Extensible

“system quality”



Code Maintainability

Requirements → Understand → Create → Code Skeleton (Design)

Knows

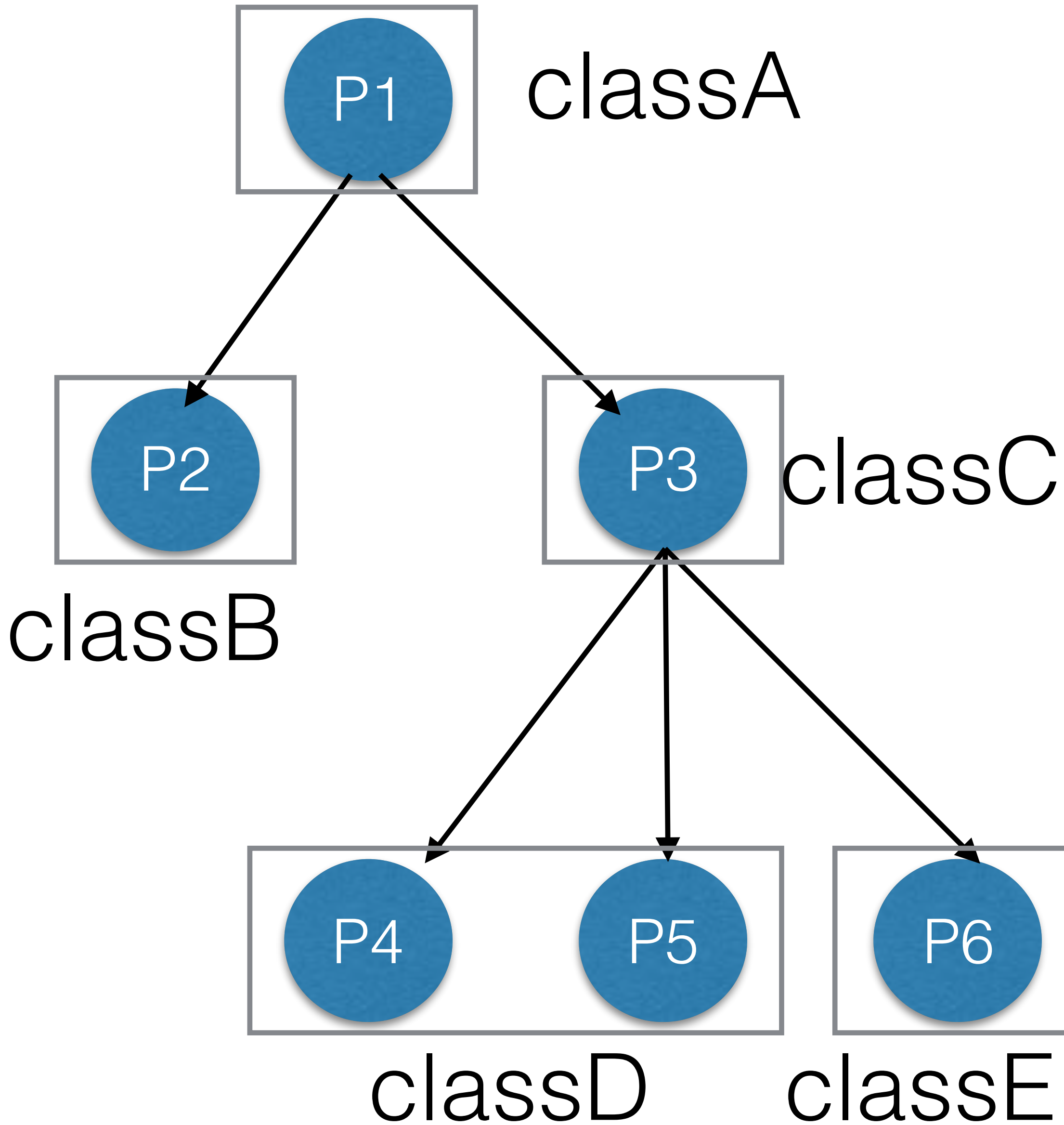
Technology
Domain
proc, OO, fun
OO Design patterns
Fun Design patterns
Design Principles
Design Anti patterns

Code Design
Low level design
Module Design
Class Design
Implementation Design

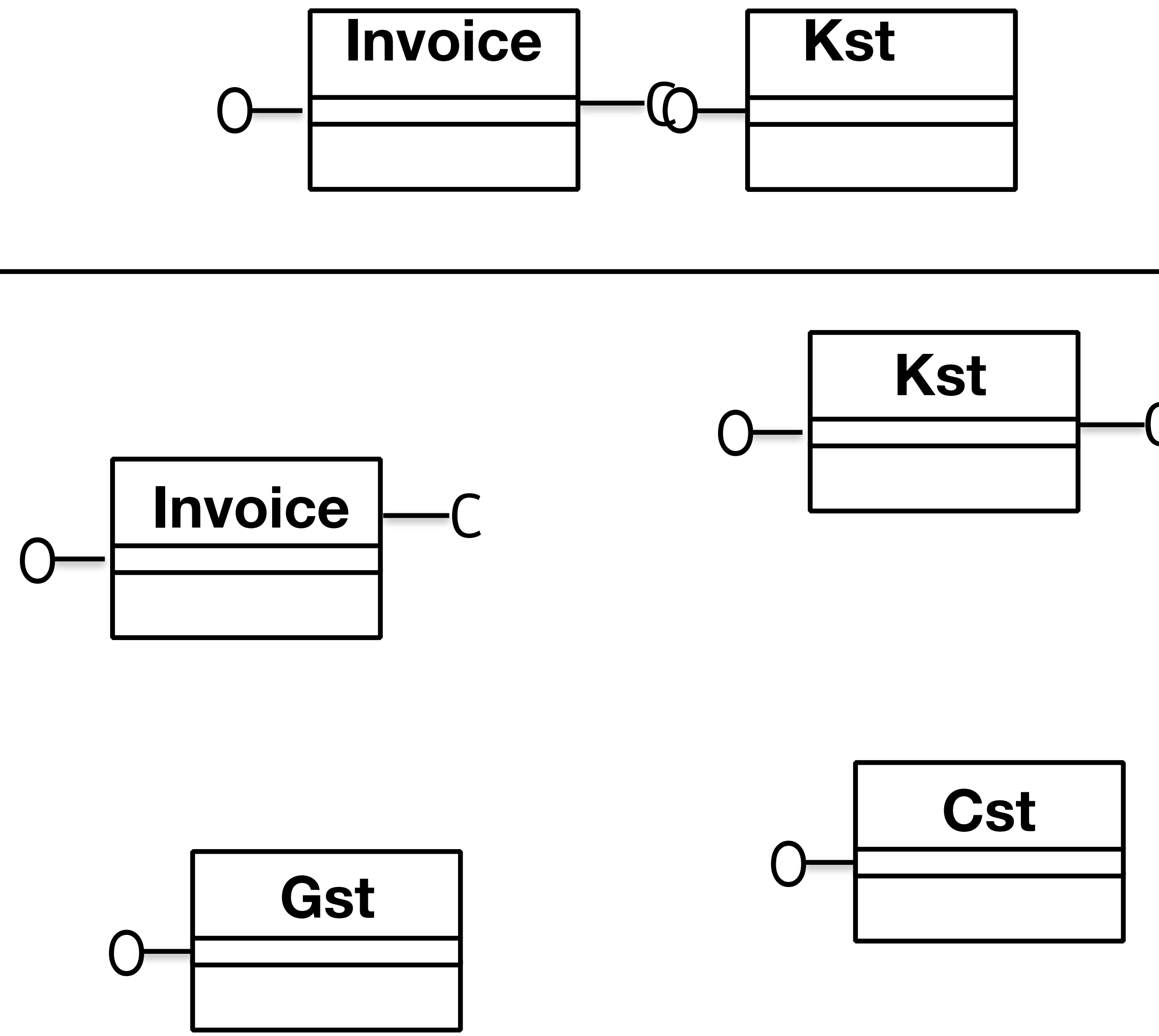
SOC

- Error handling and Domain logic
- Domain logic and Domain rules <—-
- Domain flow and Domain actions (steps)
- Technology logic and domain logic
- …

Procedural Prog
(tree)

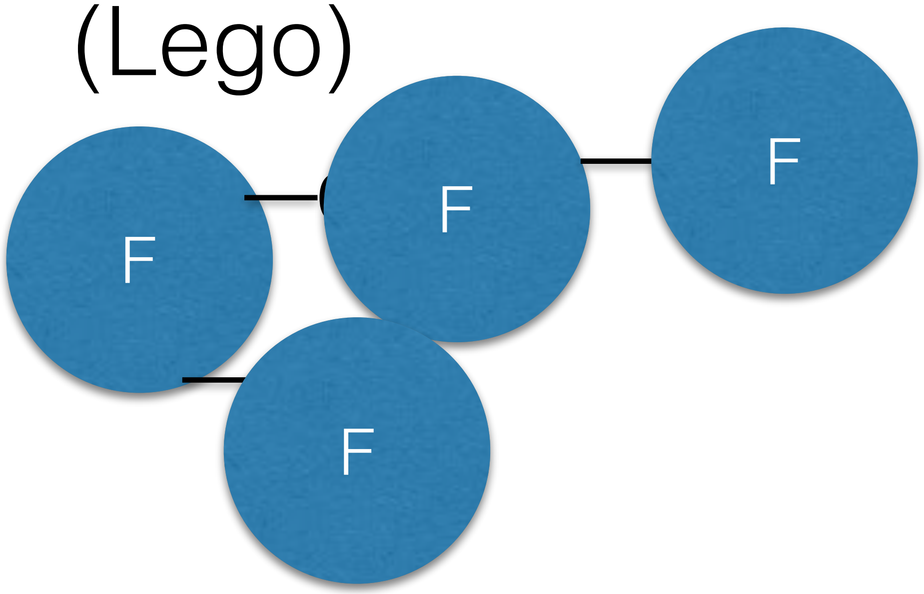


OO Prog
(Lego)



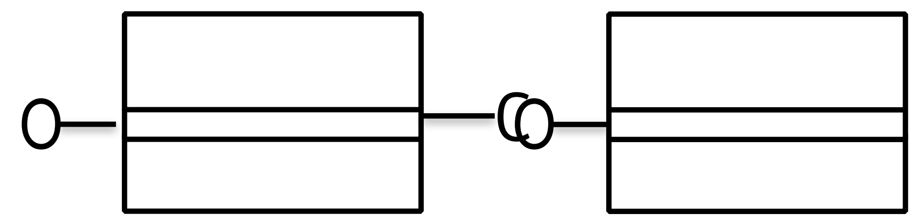
Functional Prog

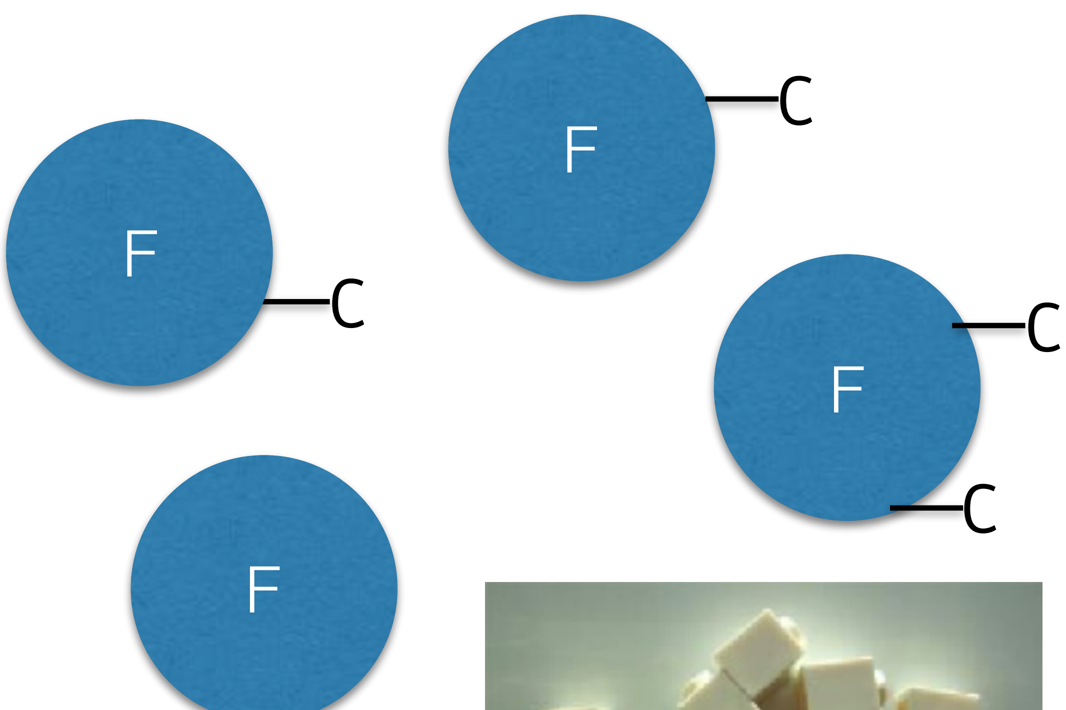
(Lego)

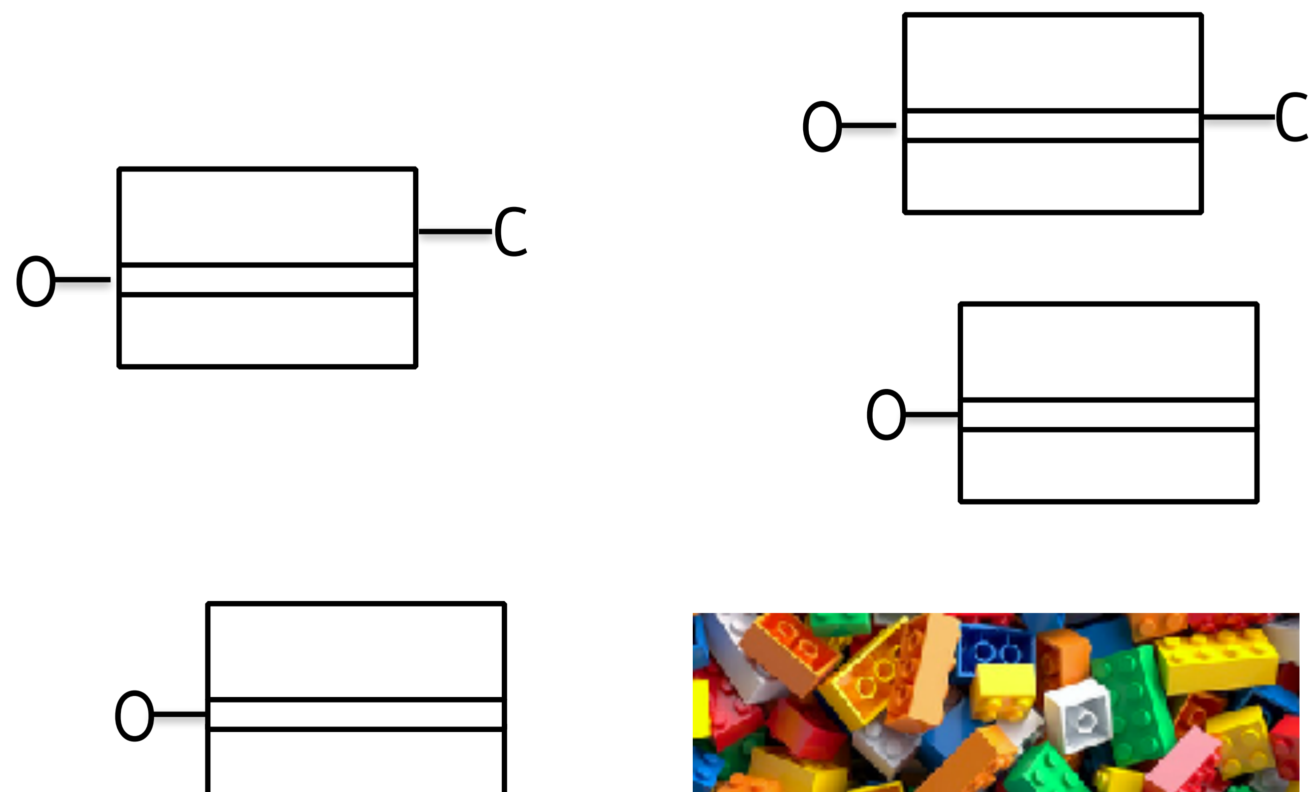


OO Prog

(Lego)







Tight coupling	Interface typing (java, c++, C#) Compiled Languages	Duck typing (py, js) Dynamic Languages

class Parrot
{
    void fly(){
        …
    }
}
	interface Bird{
	    void fly();
	}
	
	class Parrot implements Bird
	{
	    void fly(){
	        …
	    }
	}
		class Parrot{
		    void fly(){
		        …
		    }
		}

do(Parrot obj)
{
    obj.fly();
}
	do(Bird obj)
	{
	    obj.fly();
	}
		do(obj)
		{
		    obj.fly();
		}

do(new Parrot( ))	do(new Parrot( ))	do(new Parrot( ))

Interface typing (java, c++)	Duck typing (py, js)	Lamda (py,js, java)

interface Bird{
    void f1();
}
 	
class Parrot implements Bird
{
    void f1(){
        …
    }
}
	class Parrot{
	    void f1(){
	        …
	    }
	}
		class Parrot{
		    void fly(){
		        …
		    }
		}

do(Bird obj)
{
    obj.f1();
}
	do(obj)
	{
	    obj.f1();
	}
		do(Lamda fun)
		{
		    fun();
		}

do(new Parrot( ))	do(new Parrot( ))	Parrot parrot = new Parrot( ) do(()-> parrot.fly())

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

class Parrot
{
void fly(){
…
}
}
	interface Bird{
	void f1();
	}
	
	class Parrot implements Bird
	{
	void f1(){
	…
	}
	}
		class Parrot{
		void f1(){
		…
		}
		}
			class Parrot{
			void fly(){
			…
			}
			}
				class CA{
				void f1(){
				…
				}
				}

do(Parrot obj)
{
obj.fly();
}
	do(Bird obj)
	{
	obj.f1();
	}
		do(obj)
		{
		obj.f1();
		}
			do(Lamda fun)
			{
			fun();
			}
				do(string cn,string fn){
				Class c = class.forName(cn);
				m = c.getMethod(fn);
				…
				m.invoke(obj,[]);
				}

do(new Parrot( ))	do(new Parrot( ))	do(new Parrot( ))	CA obj = new CA( ) do(()-> obj.fly())	do("Parrot","fly")

	Proc	OO	Functional
Performance	n/a	n/a	
Security	n/a	n/a	
Learning Curve	++	- -	
Development Effort	++	- -	
Unit test	- -	+ +	
Manage large code	- -	++	

Liskov Substitution

```
interface Bird{
  Fly
  Chirp
  Eat
}
```

```
interface Bird{

}
```

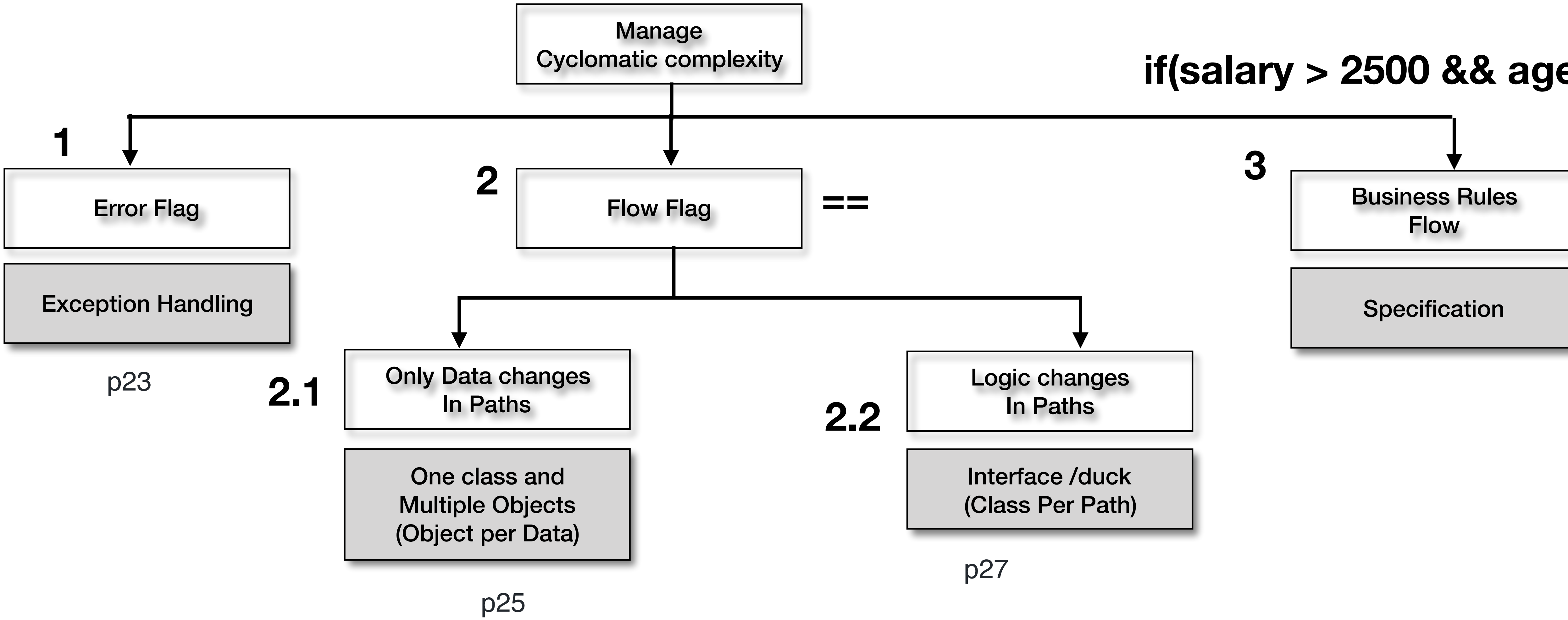
```
interface FlyingBird{
  Fly
  Chirp
  Eat
}
```

```
fun(Flying bird){
  …
  bird.fly();
  …
}
```

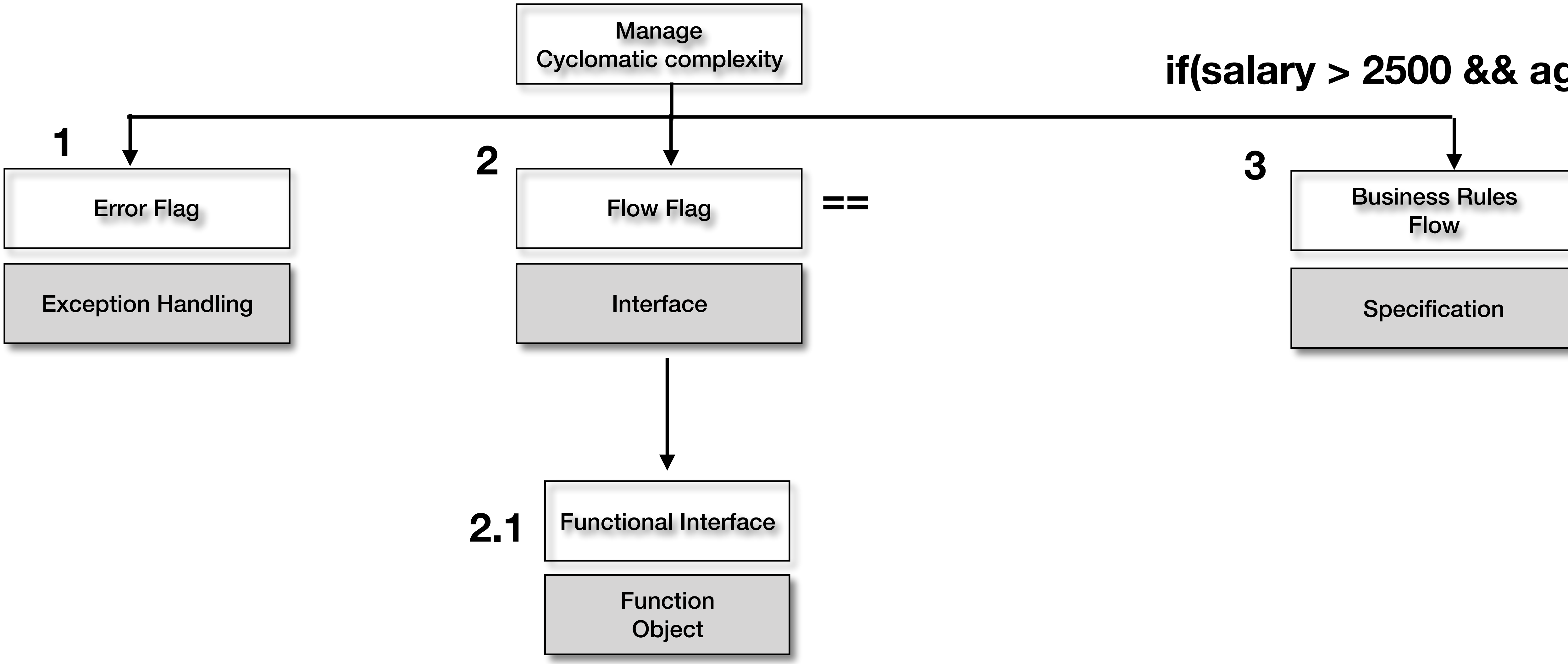
Depends on	Proc	OO	Fun
Performance	n/a	n/a	+ +
Security	n/a	n/a	n/a
Development time	+ +	- -	-
Learning curve	+ +	- -	-
Abstraction	- -	+ +	+
Low coupling	- -	+ +	++
Manage large Code (Maintainability)	- -	+ +	+
Unit test	- -	+	+ +
Debug	- -	+	+ +
Lang	C, C++, java, C#, py, js	C++, java, C#, py, js	C++, java, C#, py, js

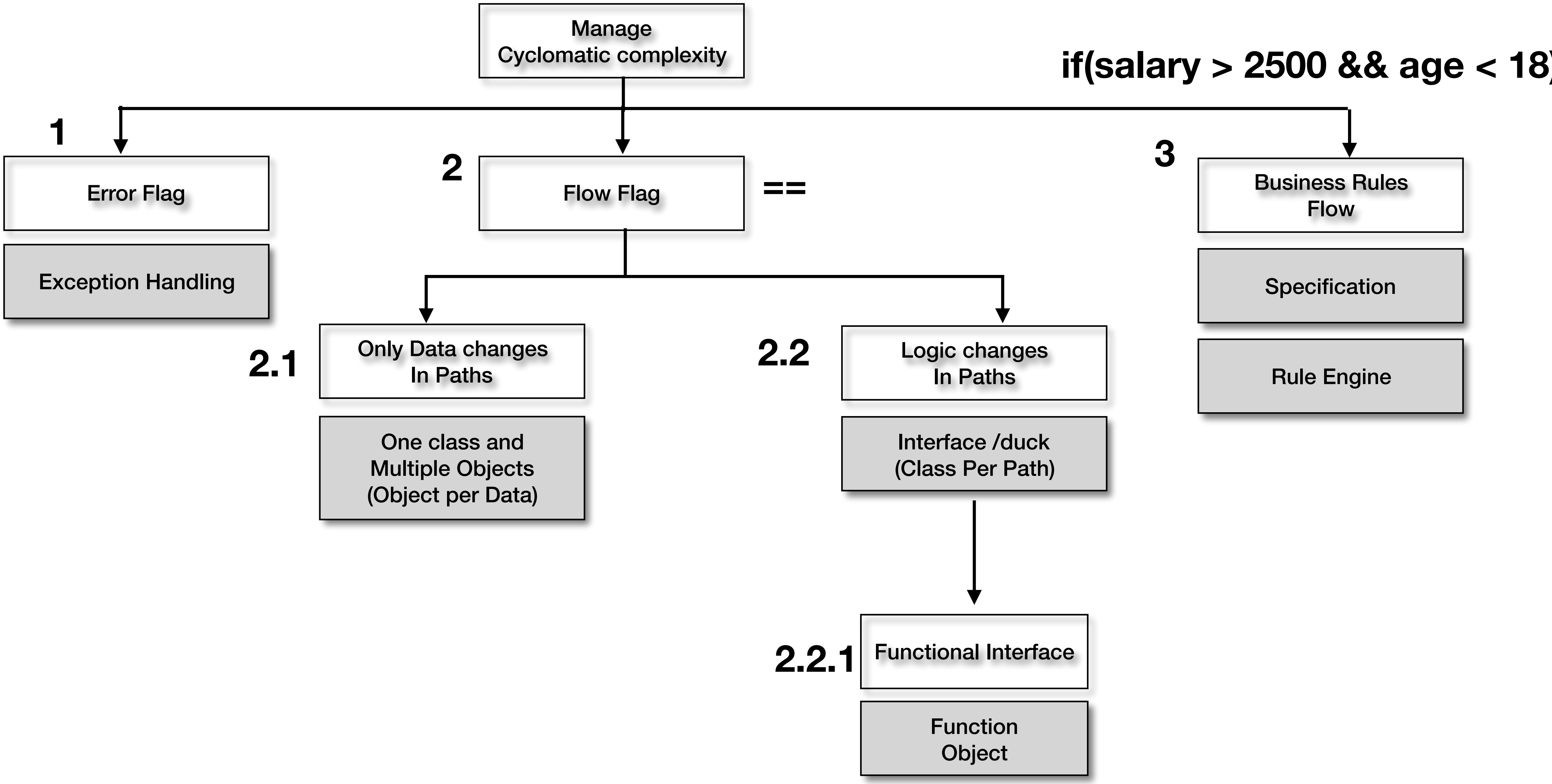
	Method call	Instantiation	Deallocation
Examples of Tight coupling	obj.fun()	new Emp()	delete emp;
Approach for Low coupling	# Interface typing * # Duck typing # Lamda	# DI * **11** # factory **24,3**	# Garbage collector # smart pointers # virtual destructor
	# reflection # wrapper **18**		

if(salary > 2500 && age < 18)

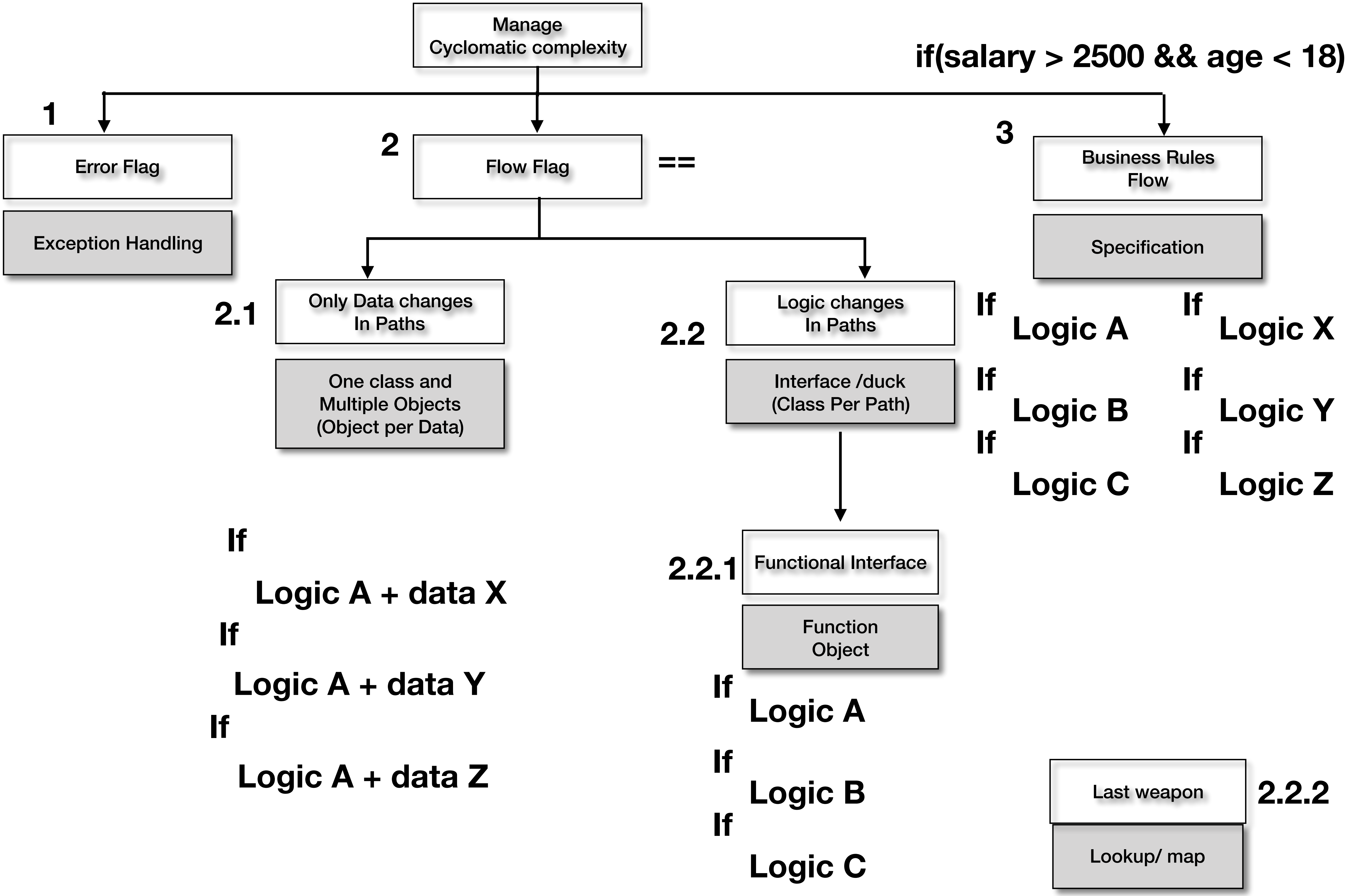


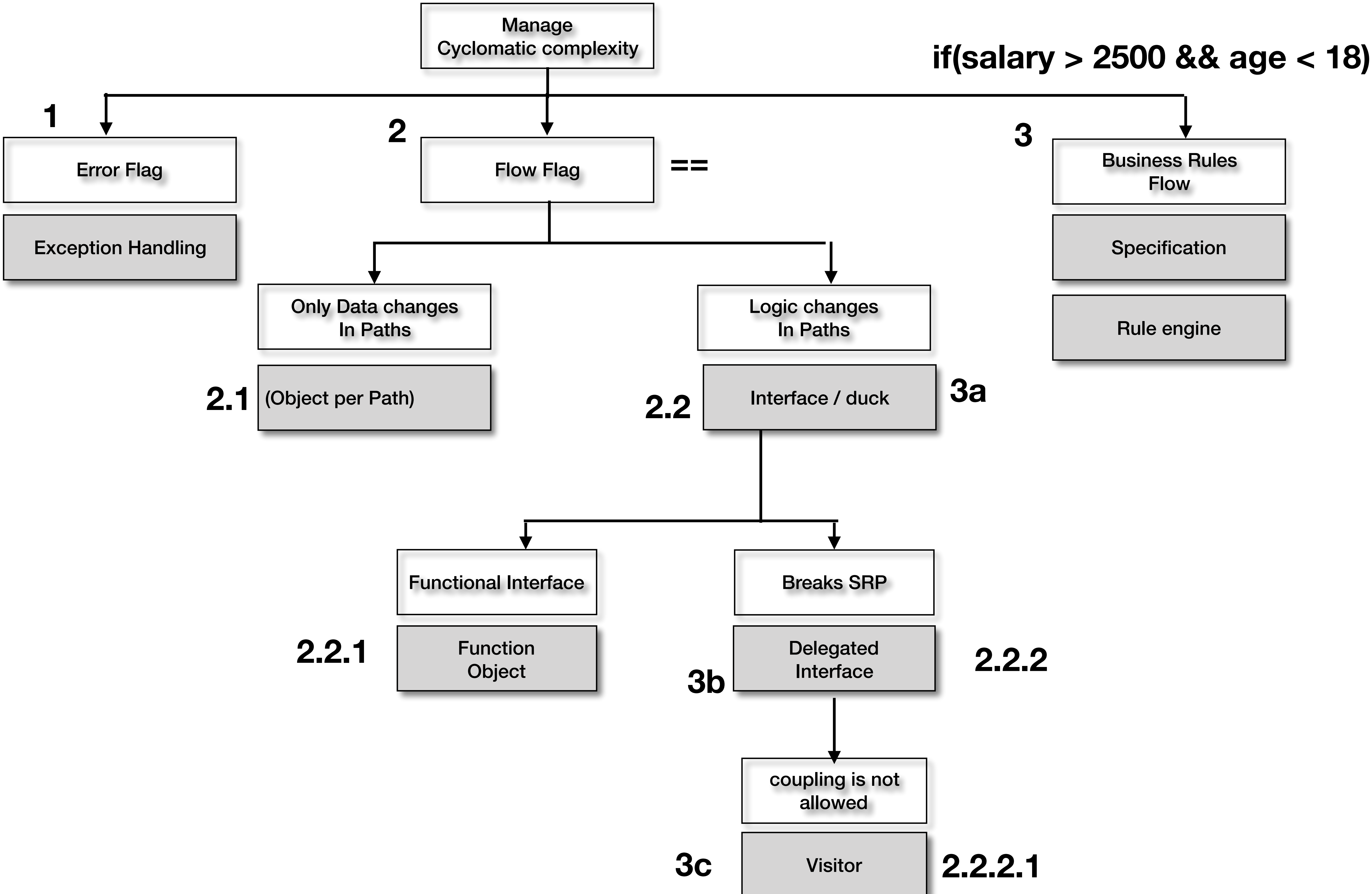
if(salary > 2500 && age < 18)

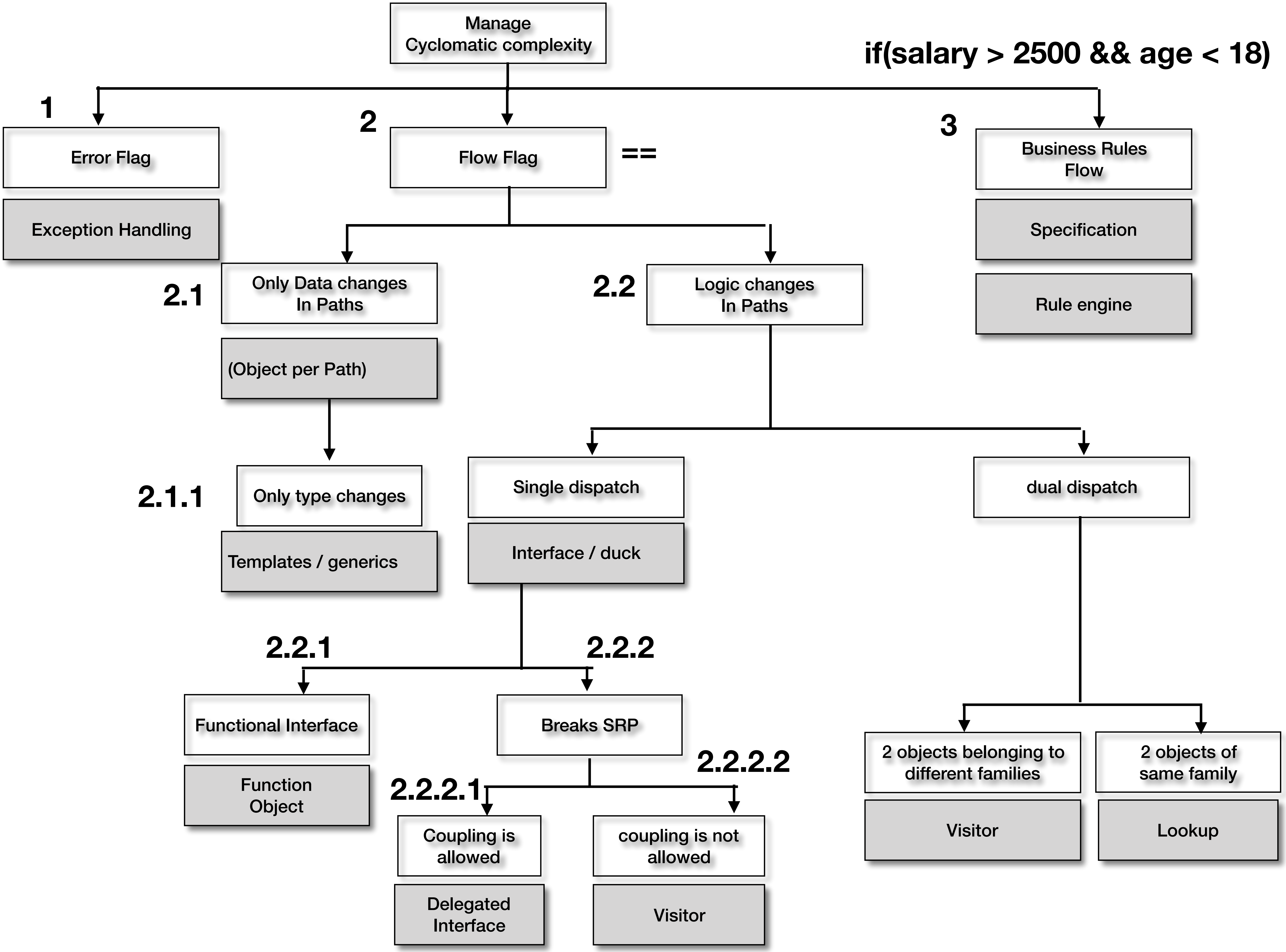


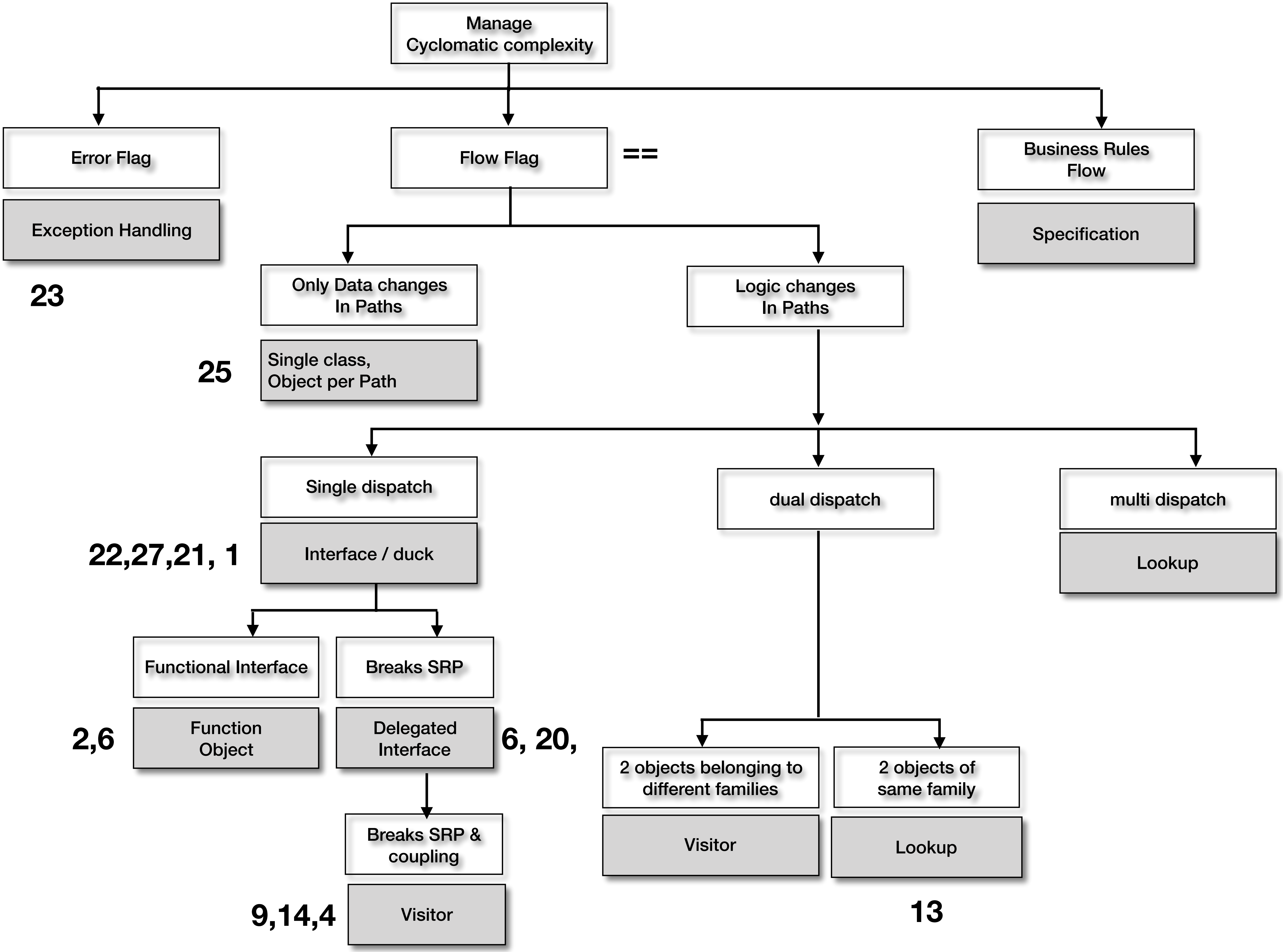


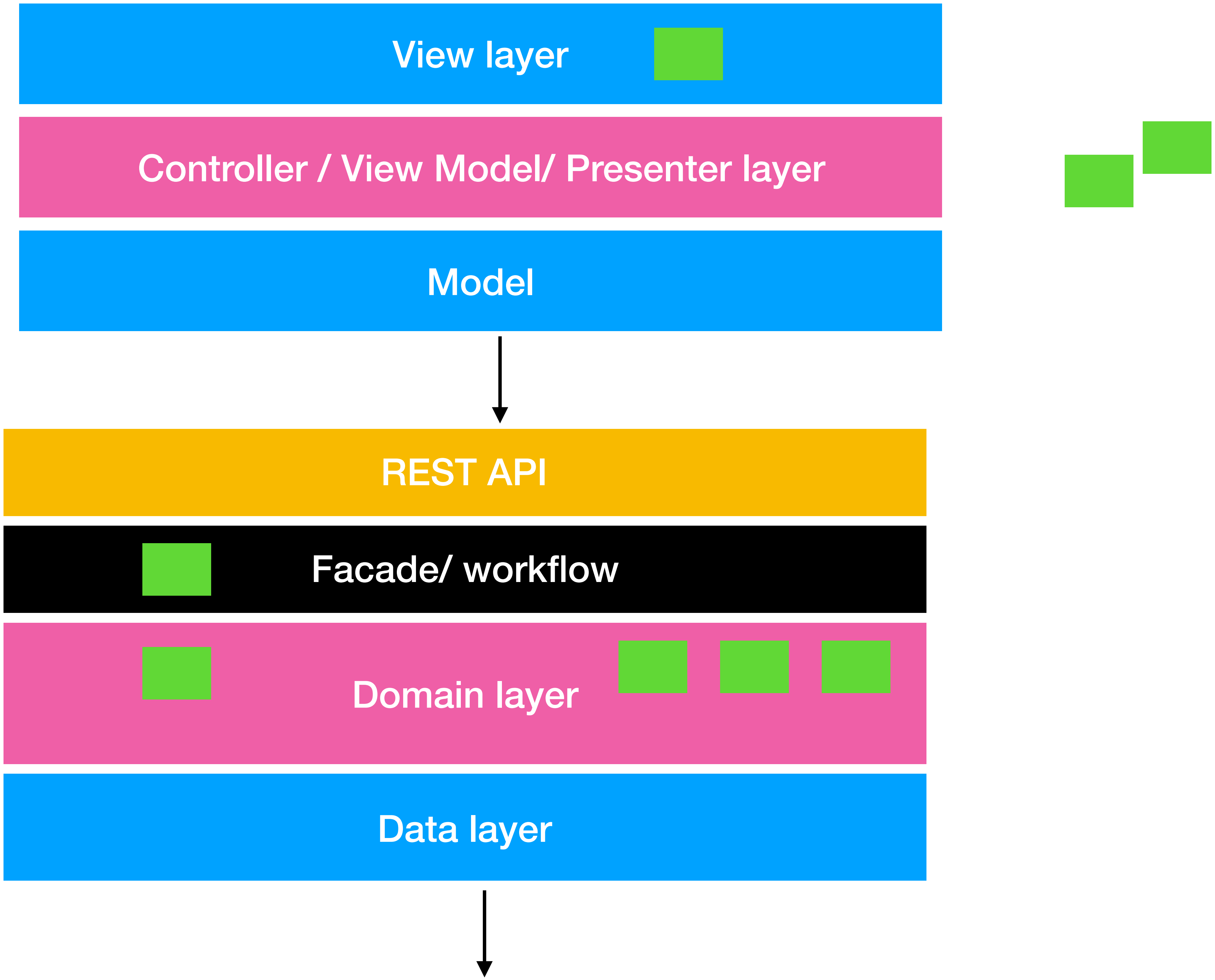
Manage
Cyclomatic complexity
if(salary > 2500 && age < 18)
1
Error Flag
Exception Handling
2
Flow Flag
==
2.1
Only Data changes
In Paths
One class and
Multiple Objects
(Object per Data)
2.2
Logic changes
In Paths
Interface /duck
(Class Per Path)
2.2.1
Functional Interface
Function
Object
3
Business Rules
Flow
Specification
Rule Engine

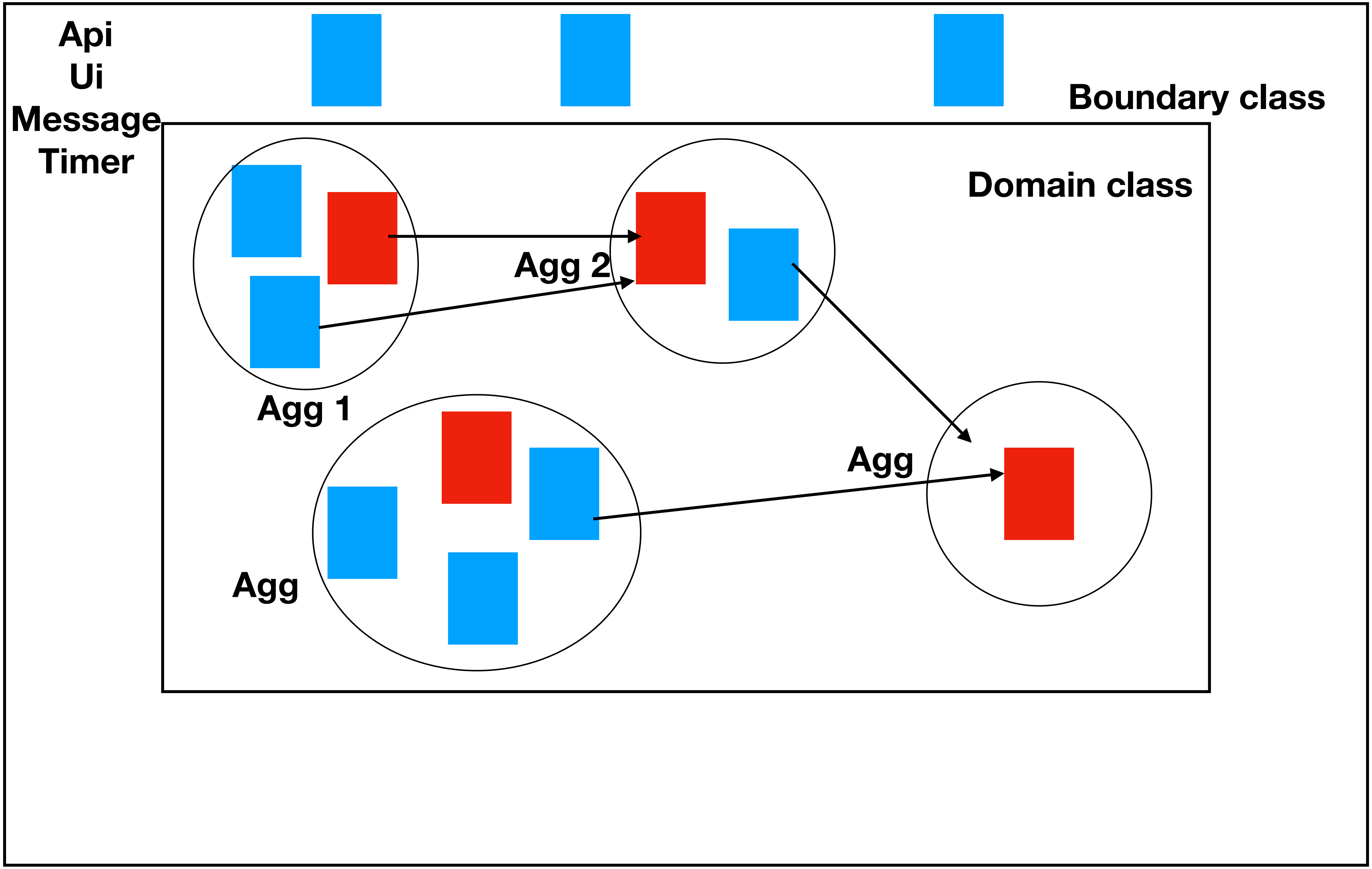

Manage Cyclomatic complexity
if(salary > 2500 && age < 18)
1
Error Flag
Exception Handling
2
Flow Flag
==
3
Business Rules Flow
Specification
2.1
Only Data changes In Paths
One class and Multiple Objects (Object per Data)
2.2
Logic changes In Paths
Interface /duck (Class Per Path)
If
Logic A
If
Logic B
If
Logic C
If
Logic X
If
Logic Y
If
Logic Z
If
Logic A + data X
If
Logic A + data Y
If
Logic A + data Z
2.2.1
Functional Interface
Function Object
If
Logic A
If
Logic B
If
Logic C
Last weapon
2.2.2
Lookup/ map

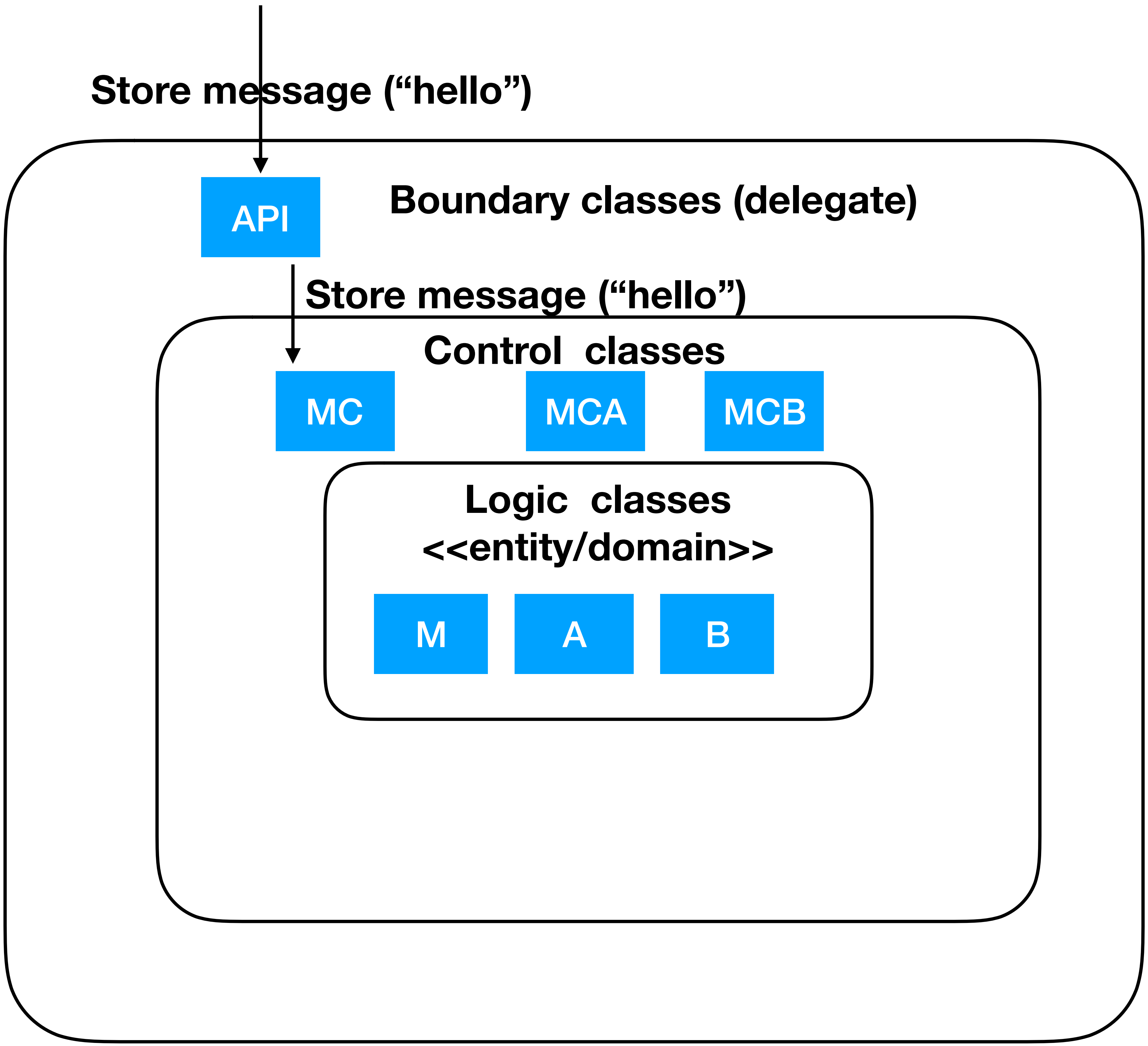

Manage
Cyclomatic complexity
if(salary > 2500 && age < 18)
1
Error Flag
Exception Handling
2
Flow Flag
==
3
Business Rules
Flow
Specification
Rule engine
Only Data changes
In Paths
2.1
(Object per Path)
Logic changes
In Paths
2.2
Interface / duck
3a
Functional Interface
2.2.1
Function
Object
Breaks SRP
Delegated
Interface
3b
2.2.2
coupling is not
allowed
3c
Visitor
2.2.2.1

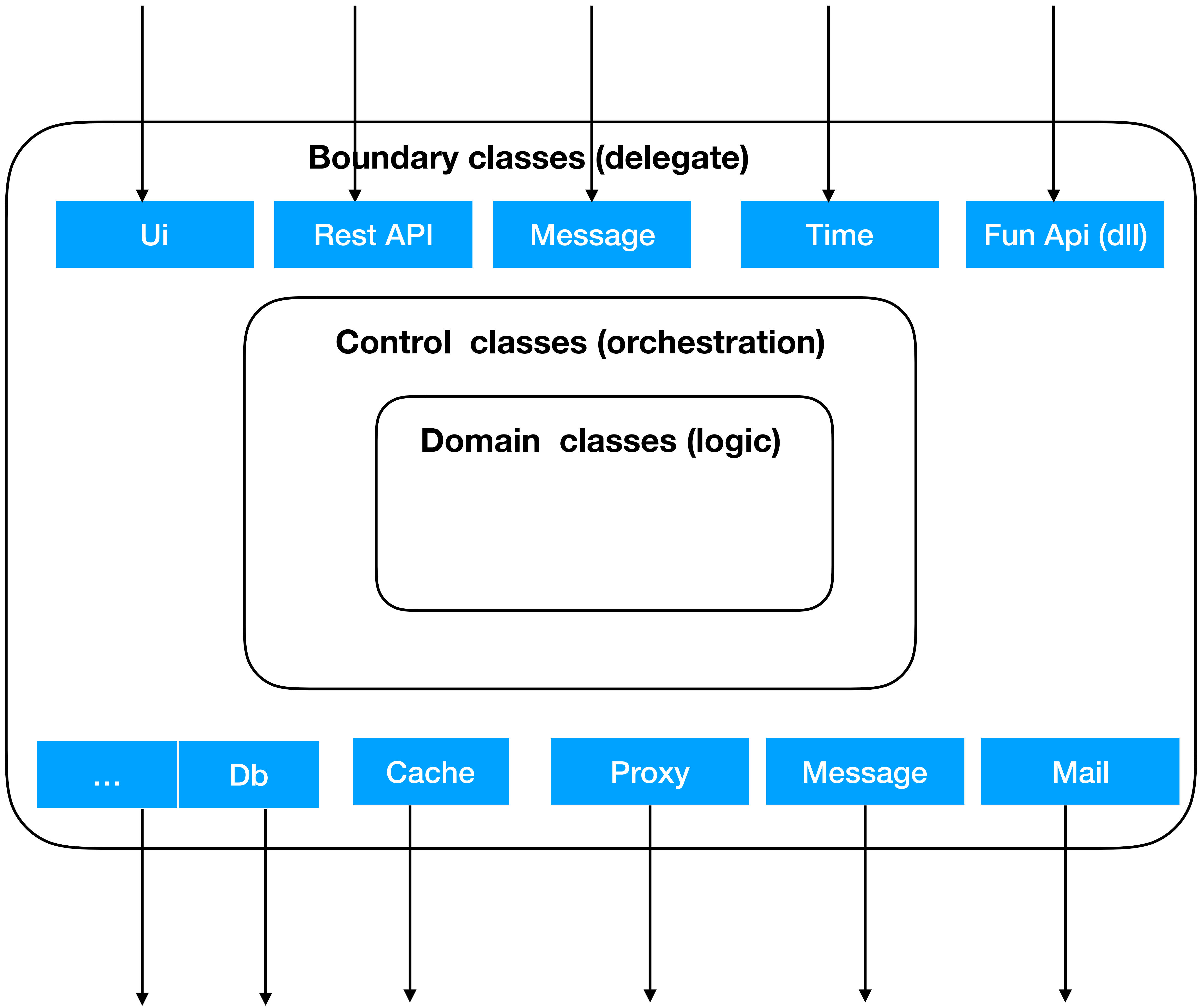

Manage Cyclomatic complexity
if(salary > 2500 && age < 18)
1
Error Flag
Exception Handling
2
Flow Flag
==
3
Business Rules Flow
Specification
Rule engine
2.1
Only Data changes In Paths
(Object per Path)
2.2
Logic changes In Paths
2.1.1
Only type changes
Templates / generics
Single dispatch
Interface / duck
dual dispatch
2.2.1
2.2.2
Functional Interface
Function Object
Breaks SRP
2.2.2.1
2.2.2.2
Coupling is allowed
Delegated Interface
coupling is not allowed
Visitor
2 objects belonging to different families
Visitor
2 objects of same family
Lookup

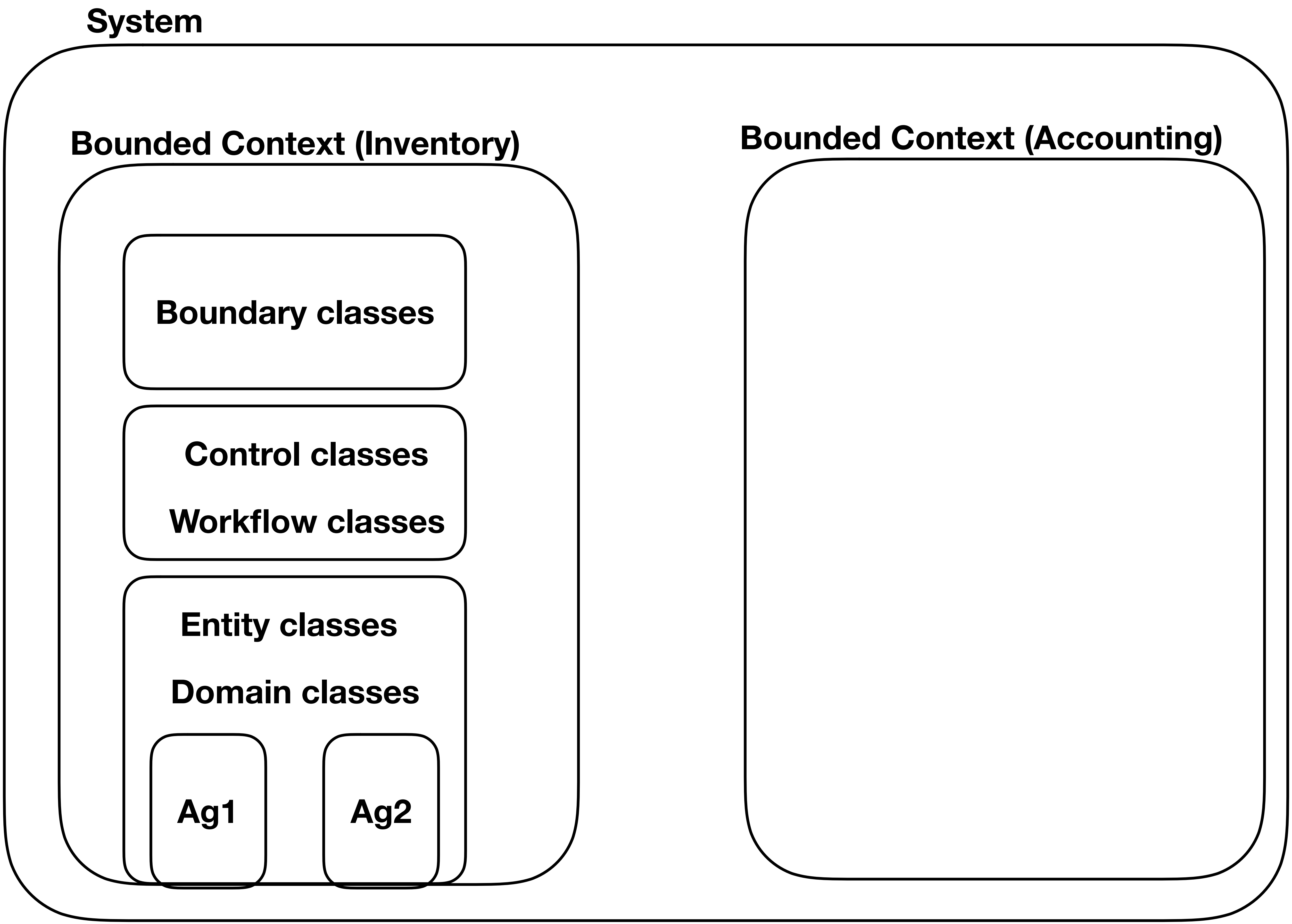

Manage Cyclomatic complexity
Error Flag
Exception Handling
23
Flow Flag
==
Business Rules Flow
Specification
Only Data changes In Paths
Single class, Object per Path
25
Logic changes In Paths
Single dispatch
Interface / duck
22,27,21, 1
dual dispatch
multi dispatch
Lookup
Functional Interface
Function Object
2,6
Breaks SRP
Delegated Interface
6, 20,
Breaks SRP & coupling
Visitor
9,14,4
2 objects belonging to different families
Visitor
2 objects of same family
Lookup
13


View layer
Controller / View Model/ Presenter layer
Model
REST API
Facade/ workflow
Domain layer
Data layer


Api
Ui
Message
Timer
Boundary class
Domain class
Agg 2
Agg 1
Agg
Agg


Store message (“hello”)
API
Boundary classes (delegate)
Store message (“hello”)
Control classes
MC
MCA
MCB
Logic classes
<<entity/domain>>
M
A
B


Boundary classes (delegate)
Ui
Rest API
Message
Time
Fun Api (dll)
Control classes (orchestration)
Domain classes (logic)
...
Db
Cache
Proxy
Message
Mail


System
Bounded Context (Inventory)
Bounded Context (Accounting)
Boundary classes
Control classes
Workflow classes
Entity classes
Domain classes
Ag1
Ag2

Boundary classes

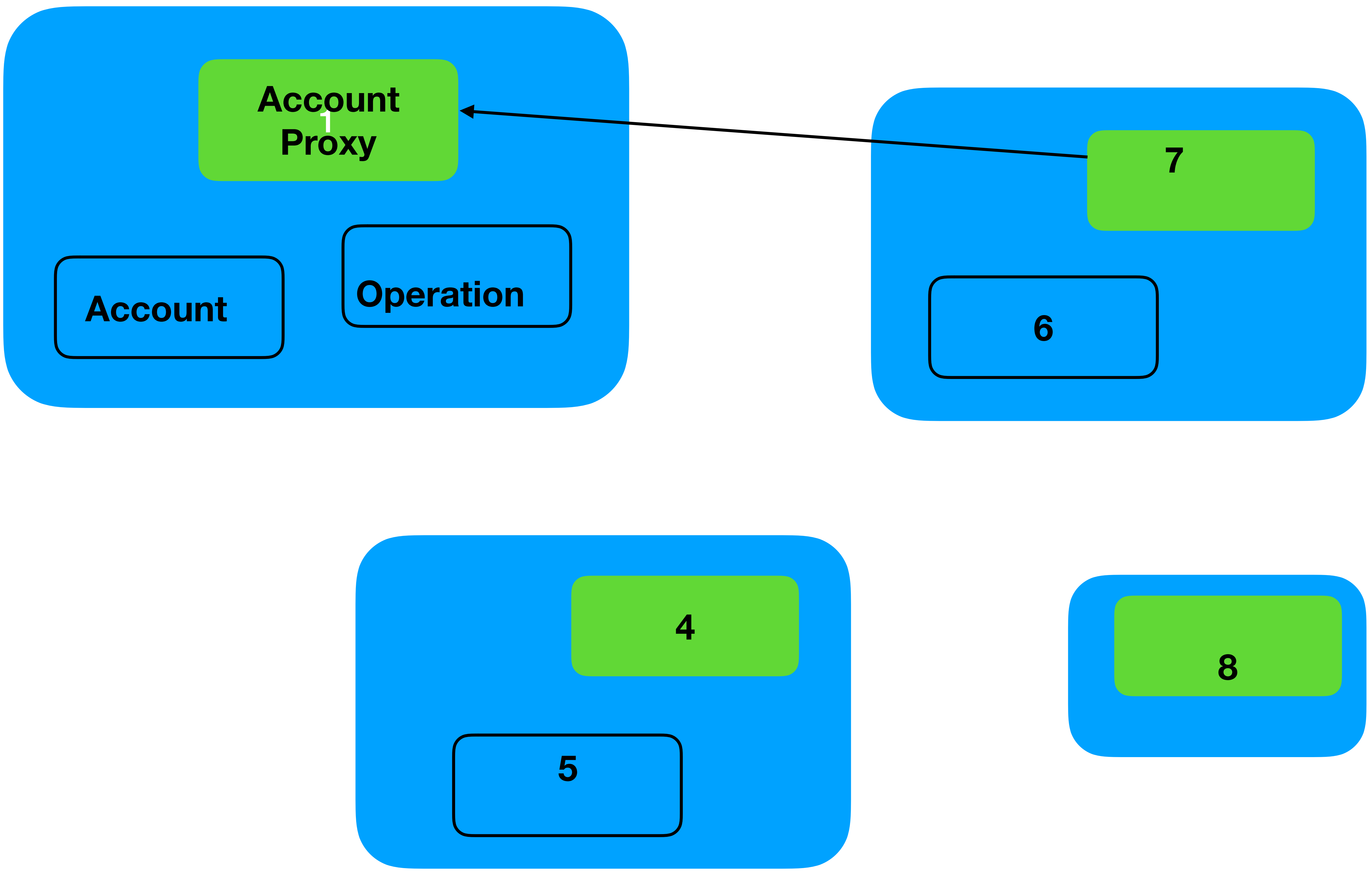
Control classes

Workflow classes

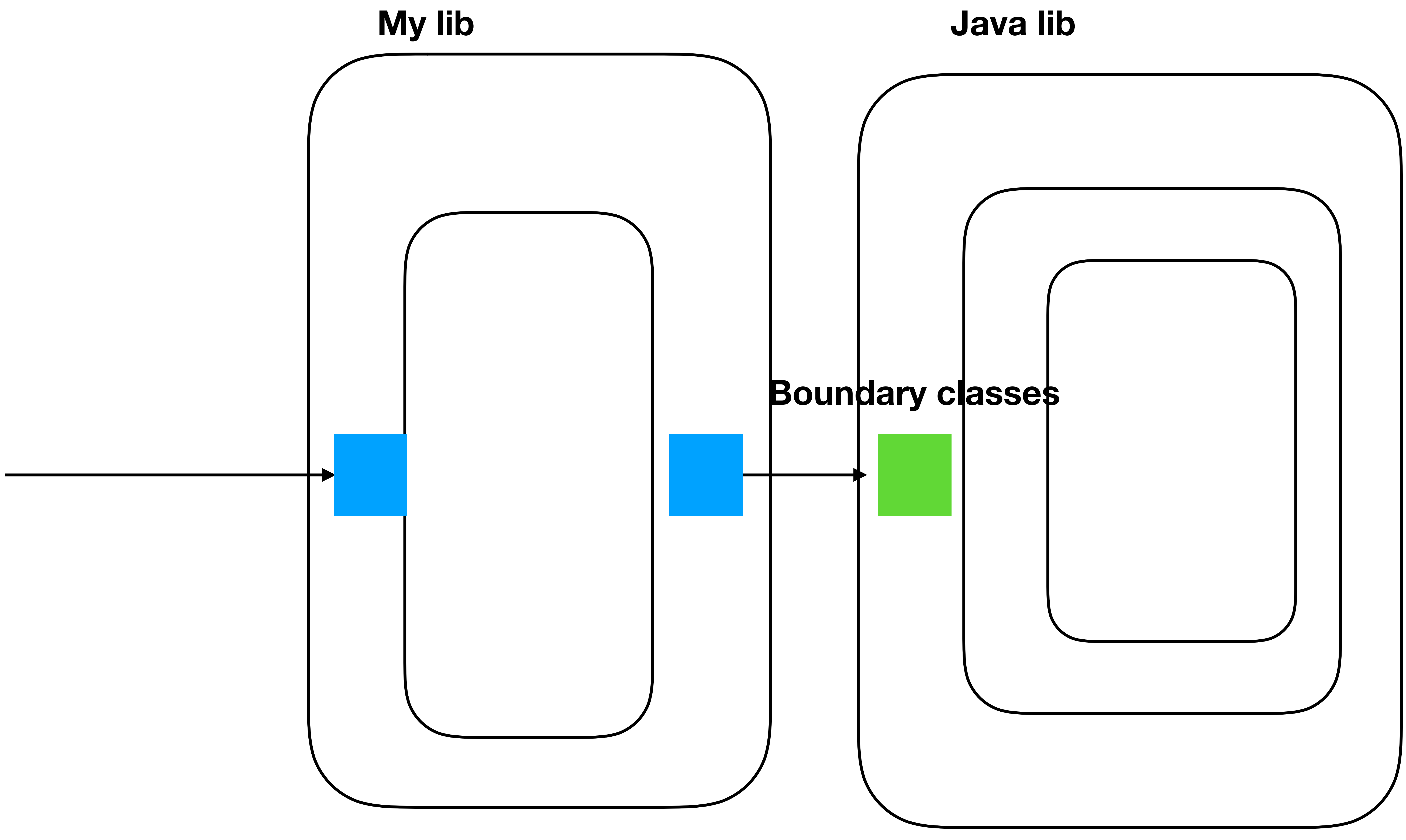
Entity classes

Domain classes

String



Account
1
Proxy
Account
Operation
7
6
4
5
8


My lib
Java lib
Boundary classes

Good

- LSP
- ISP
- Unit testable
- OCP
- Exception handling
- Separate error handling from domain logic (SOC)
- Boundary Control Entity (SOC)
- Separate flow from action (SOC)
- Separate Rules from action (SOC)
- Lookup (LW) map, vector, ..

- YAGNI
- KISS
- DIP
- Program to an interface
- Low coupling (**)
- SRP (***)
- DRY (**)
- Prefer composition over inheritance

Bad

- Flag
- If/switch
- Cyclomatic Complexity
- Type check
- Downcasting
- Fun overloading on Family of class
- Arrow code
- Bool, null, int for error

- Magic numbers/string
- Inheritance
- Swiss knife
- God Class
- Static Methods
- Functional interface
- Cyclic coupling/bi directional
- * to * coupling

SRP

- Bank
- Util
- Handler
- Controller
- Manager
- Helper
- Service
- Executor
- Mapper
-

- Class
  - Max behaviour : 12
  - Avg behaviour : 5
- Fun
  - Max lines : fit screen
  - Avg : < 10

Types of Factory

```
class Rectangle{ <— domain class
  void draw(){
    … domain logic
  }
  Square CreateSquare(){ <— factory method
    return new Square();
  }
}
```

```
class Rectangle{ <— domain class
  void draw(){
    … domain logic
  }
  static Square CreateSquare(){ <— creator method
    return new Square();
  }
}
```

```
interface ShapeFactory{ <—  abstract factory
   Square CreateSquare();
   Rect CreateRect();
}
class 2DShapeFactory implements ShapeFactory{}
class 3DShapeFactory implements ShapeFactory{}
```

```
class ShapeFactory{ <—  class factory
  Square CreateSquare(){ <— factory/creator methods
    return new Square();
  }
  Rect CreateRect(){  <— factory/creator methods
    return new Rect();
  }
}
```

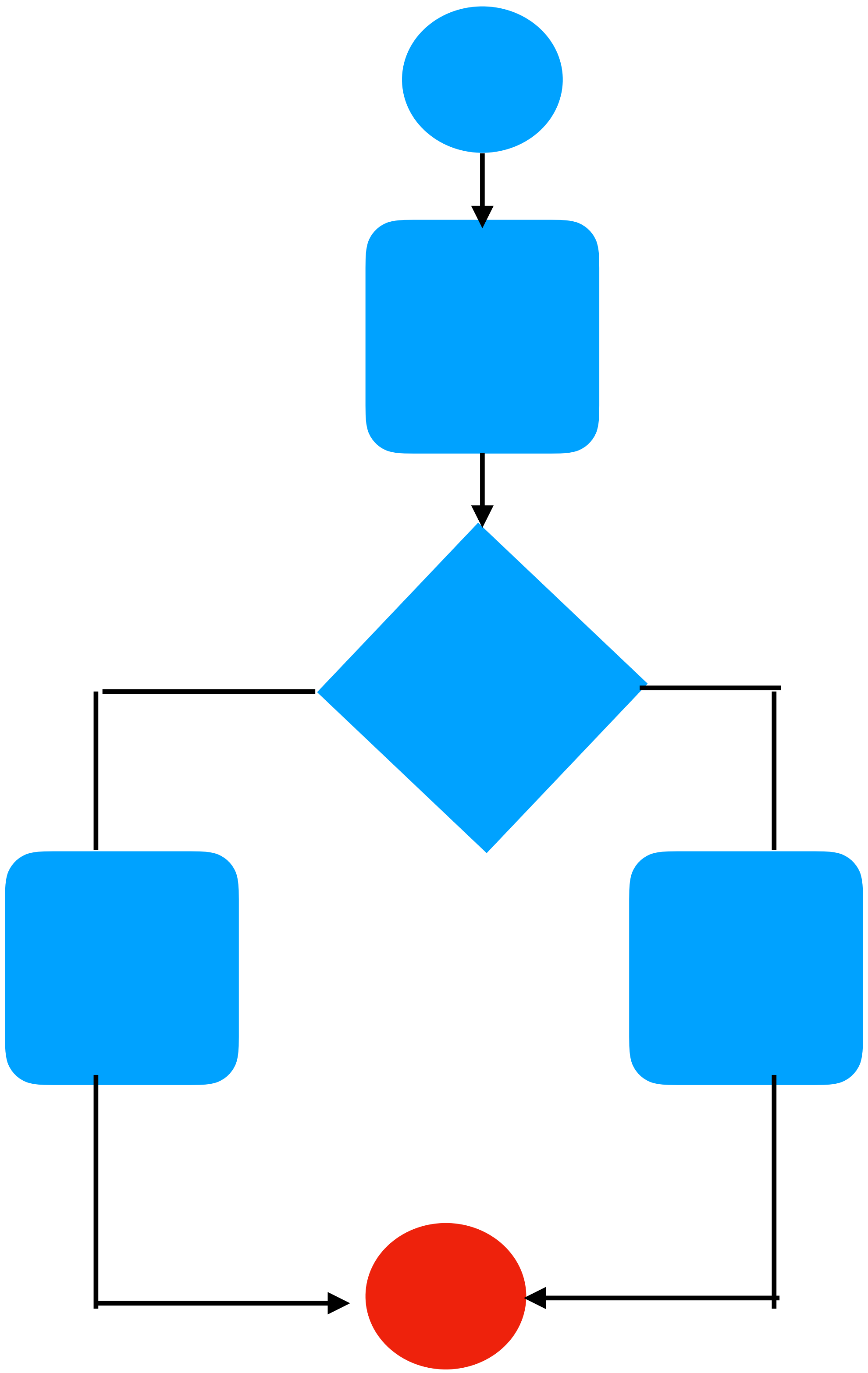
Good

- OCP
- Low cyclomatic complexity (<10 for a method)
- Program to an Interface
- Aggregate Root (*)
- Low coupling (**)
- KISS
- YAGNI
- DRY (*)
- LSP
- ISP
- SOC
    - Domain logic , error handling logic
    - Domain logic, Domain Rules
    - Domain logic, technology logic
- **SRP (***)**
    - Library:
        - Max : 30 class
        - Avg : 15 class
    - Class/ Interface
        - Max: 12 public methods
        - Avg: 5 public methods
    - Method:
        - Max: fit screen
        - Avg : 6 lines
- Boundary Control Entity (*)
- Hexagonal Arch
- Prefer Aggregation over Inheritance

Bad

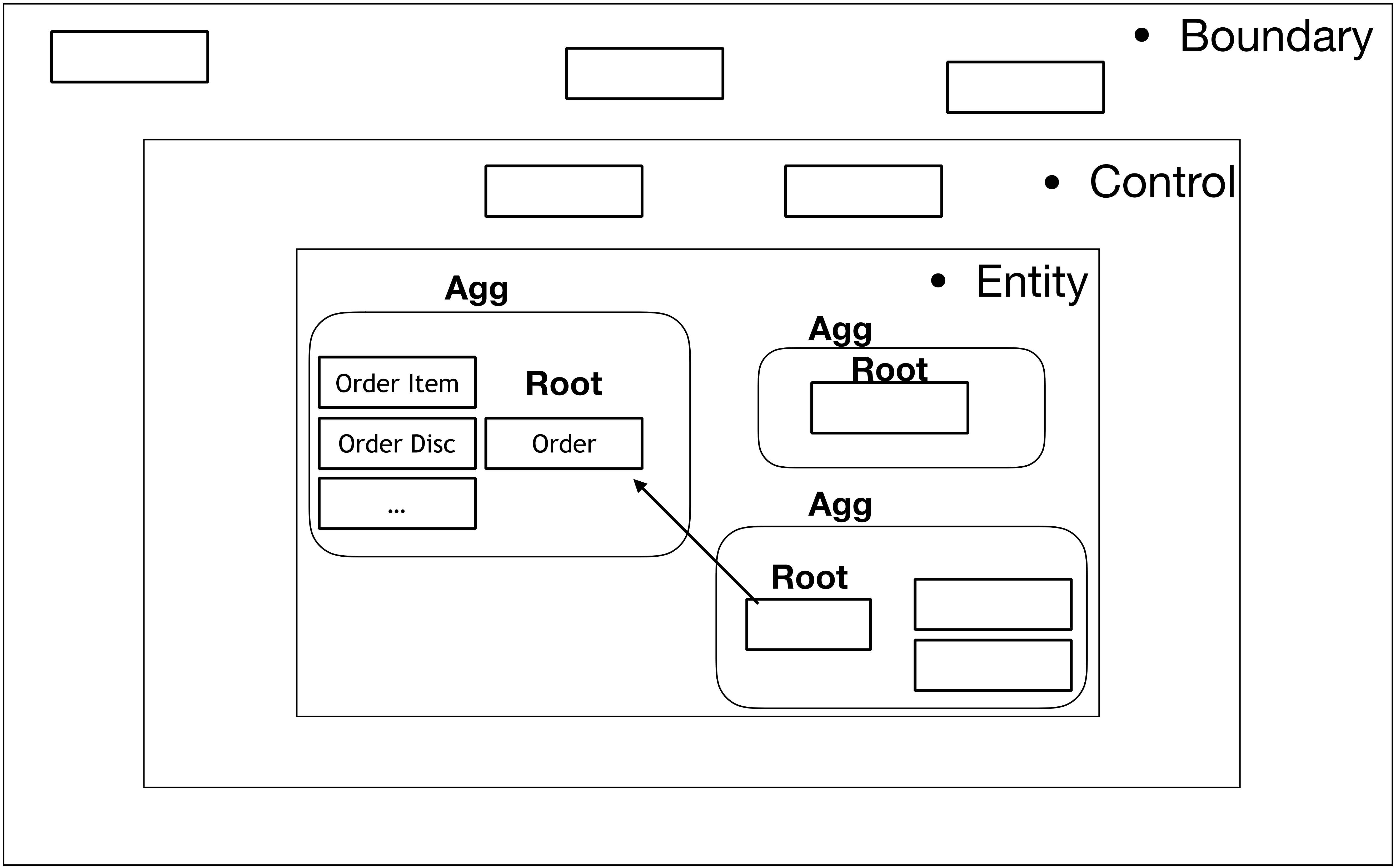
- Flag
- Inheritance (extends)
- High Cyclomatic Complexity
- Type check
- Down cast
- Overloading
- Static Methods

- Cyclic coupling
- Functional Interface
- God Class
- Swiss knife
- Arrow code
- Bool, null, int, optional for error handling
- Bool, nullable , optional parameter

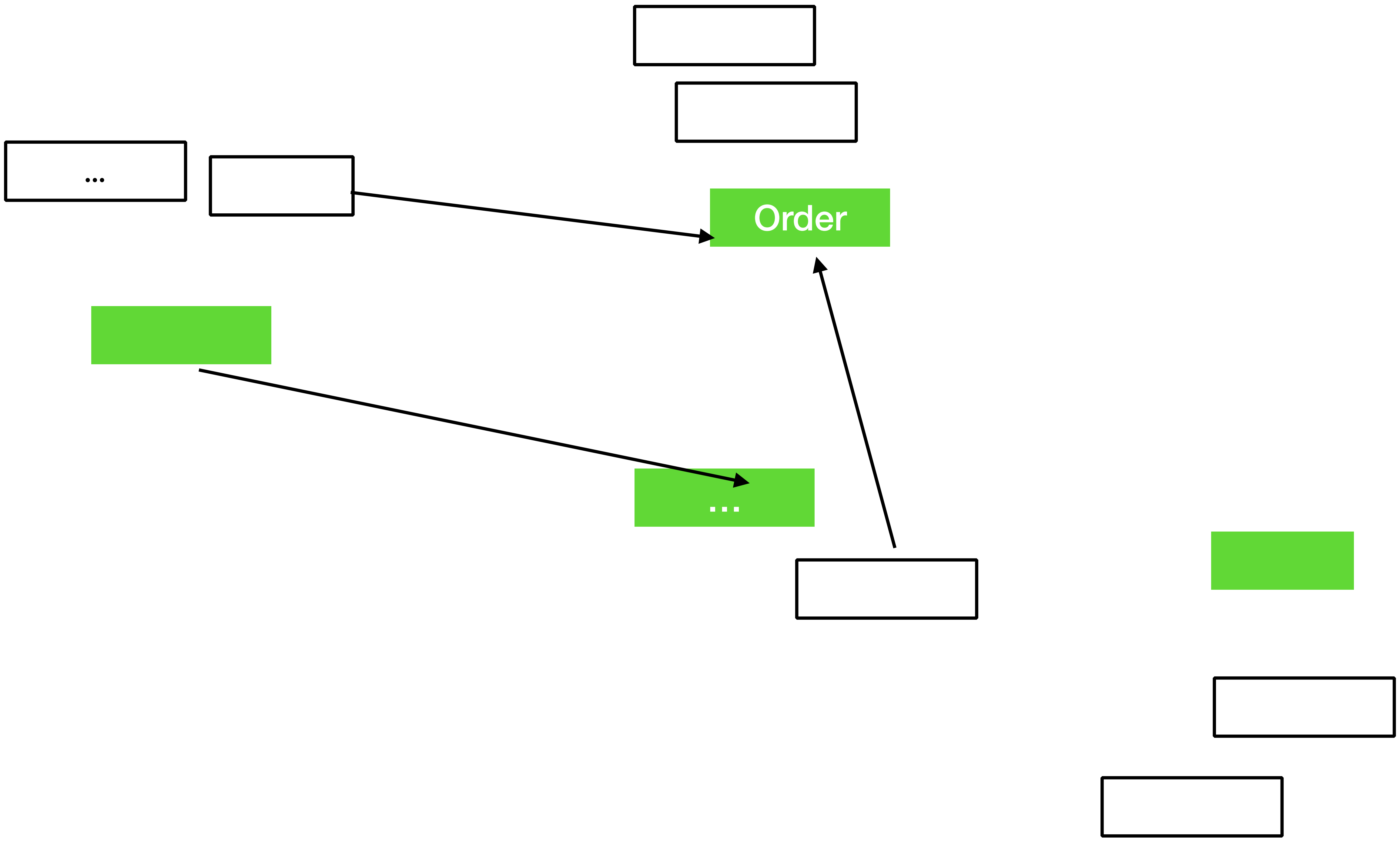


- Boundary Control Entity (*)

- Aggregate



DDD



Complexity

Error Handling

<< exceptions>>

1

Control flow

Only data changes For every path

<<Object per path>>

2

Logic changes For every path

Functional Interface

<<interface>>

3

Non Functional Interface

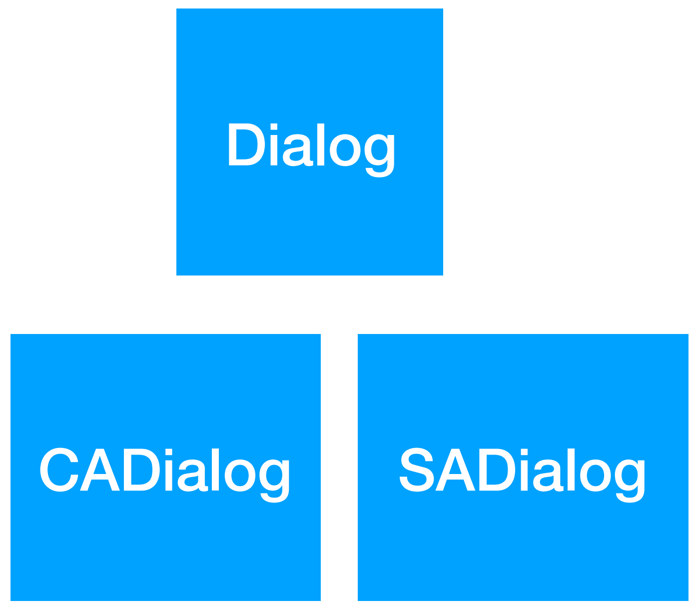
<<Lamda>>

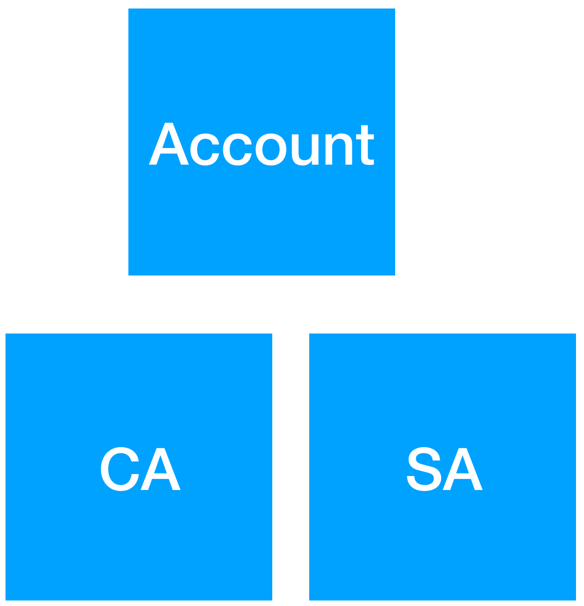
4

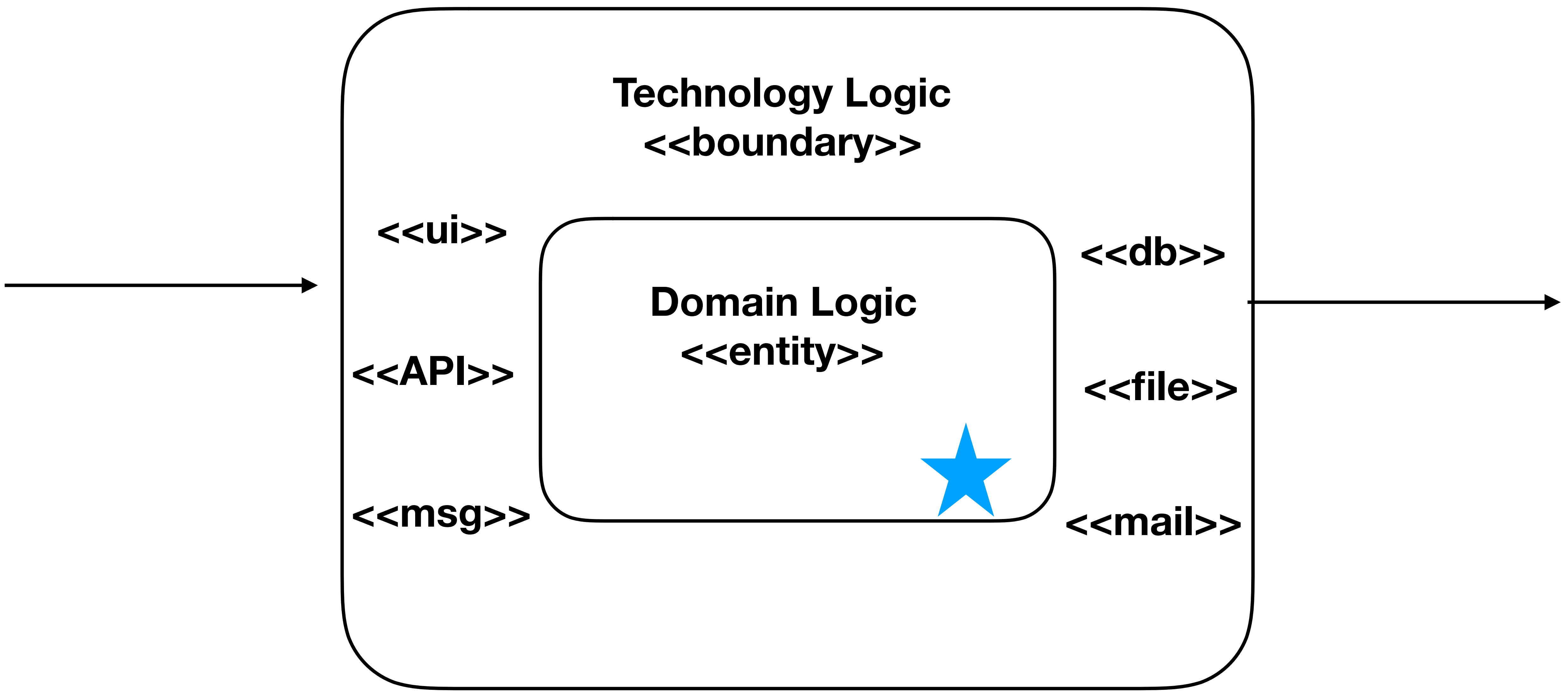
Business rules

<<Specification Pattern>>

5


Dialog
CADialog
SADialog


Account
CA
SA



path = SurveyData.processed.getPath();

Flag => interface
No. Of methods in the interface depends on no. of places the flag is used
No. of implementation depends on possible values for the flag

```
if(IsRule1(data))
{
     ….. logic
}
```

Specification Pattern

```
bool IsRule1(data){
  if((Salary > 6000 && Salary < 9000) && age < 25)
  {
    return true;
  }
  return false;
}
```

Flag == Polymorphism (interface)

Open for adding new code
and **closed** for changing existing code

Flag => formater.format(document)

Flag == Interface
Flag == Exceptions

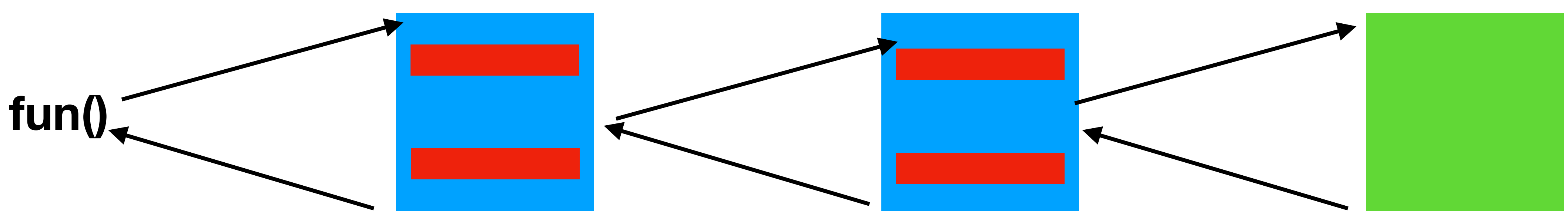
```
Repostory repo = new Repository();
res = repo.connect();
if(res == true){
        res = repo.authenticate();
        if(res == true){
                Emp emp = rep.get(empCode);
                if(emp != null){
                        return emp.getSalary();
                }else{
                        return 0;
                }
        }else{
                return 0;
        }
}else{
        return 0;
}
```

```
Interface Bird{
  ....
}
Interface FlyingBird extends Bird{
   fly()
   ...
}
```

```
Class Parrot implements Bird{
 ....
}

doJob(new Parrot());
```

```
doJob(FlyingBird bird){

   brid.fly();

}
```

Liskov substitution principle


fun()

LSP

```
class Stack {}
class Queue {}
class List {}
```

<<reference/ aggregation>>

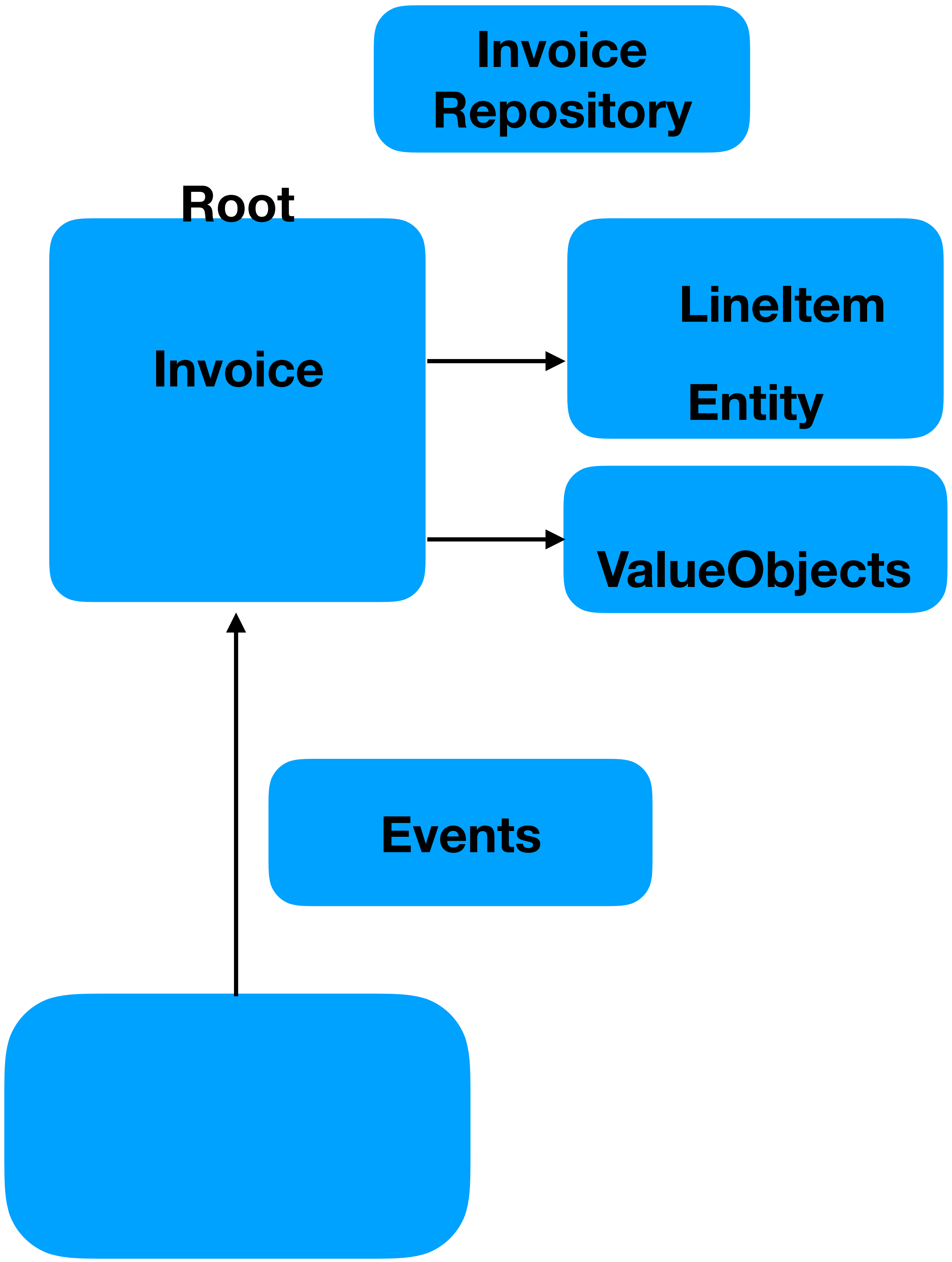
```
class Collection {
    add(index, item) { … }
}
class Stack{
  Collection ref;
}
class Queue {
  Collection ref;
}
class List {
  Collection ref;
}
```

<<Inheritance/ extends>>

```
class Collection {
    add(index, item) { … }
}
class Stack extends Collection {}
class Queue extends Collection {}
class List extends Collection {}
```

```
fun(Collection c){
    c.add(3,"hello");
}
```

Aggregate


Invoice
Repository
Root
Invoice
LineItem
Entity
ValueObjects
Events

Polymorphism

? logic1()
logic2()
logic3()

Single Dispatching
 # Interface, visitor

Account Dialog ? 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
}
```

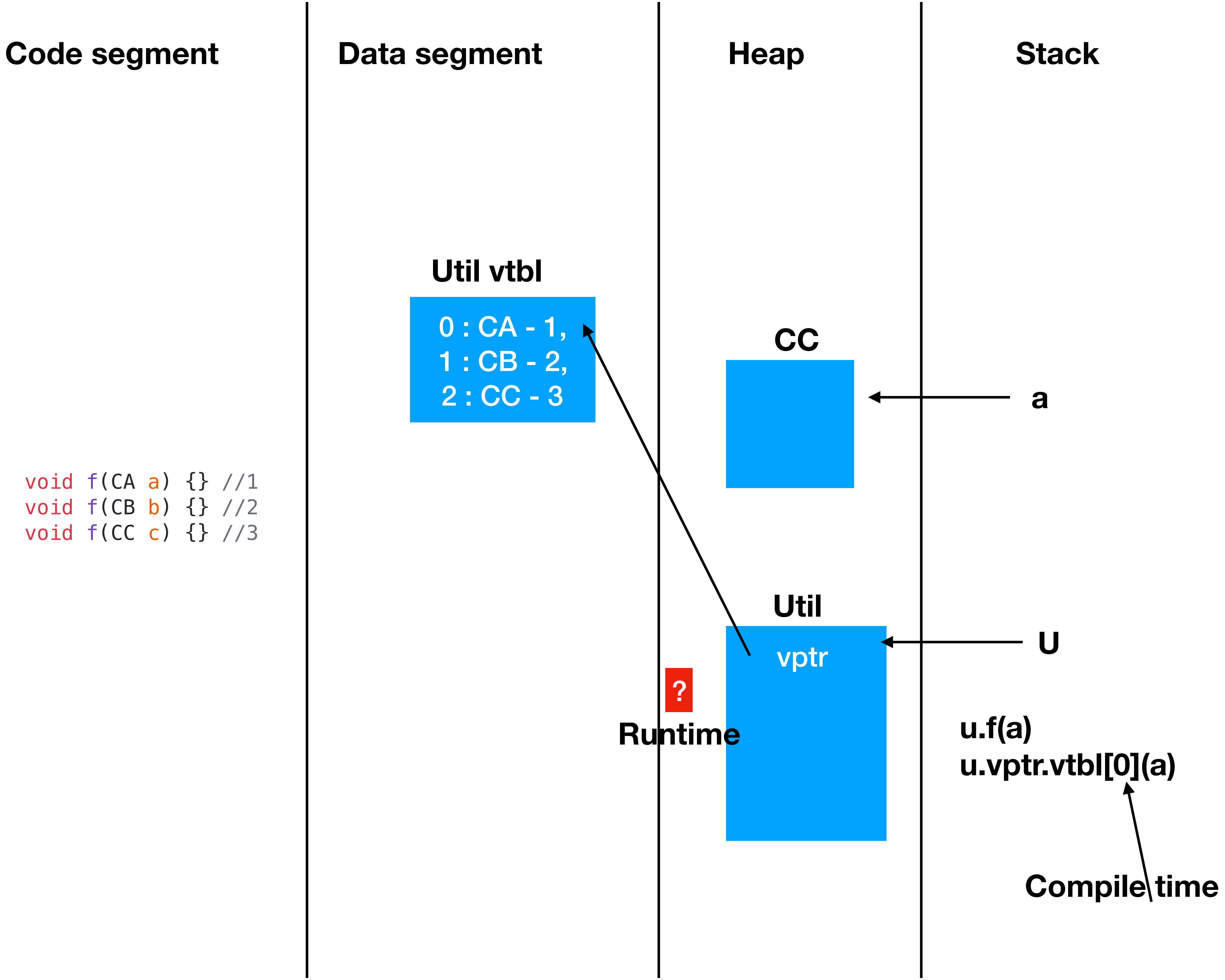
Single dispatch - delegate

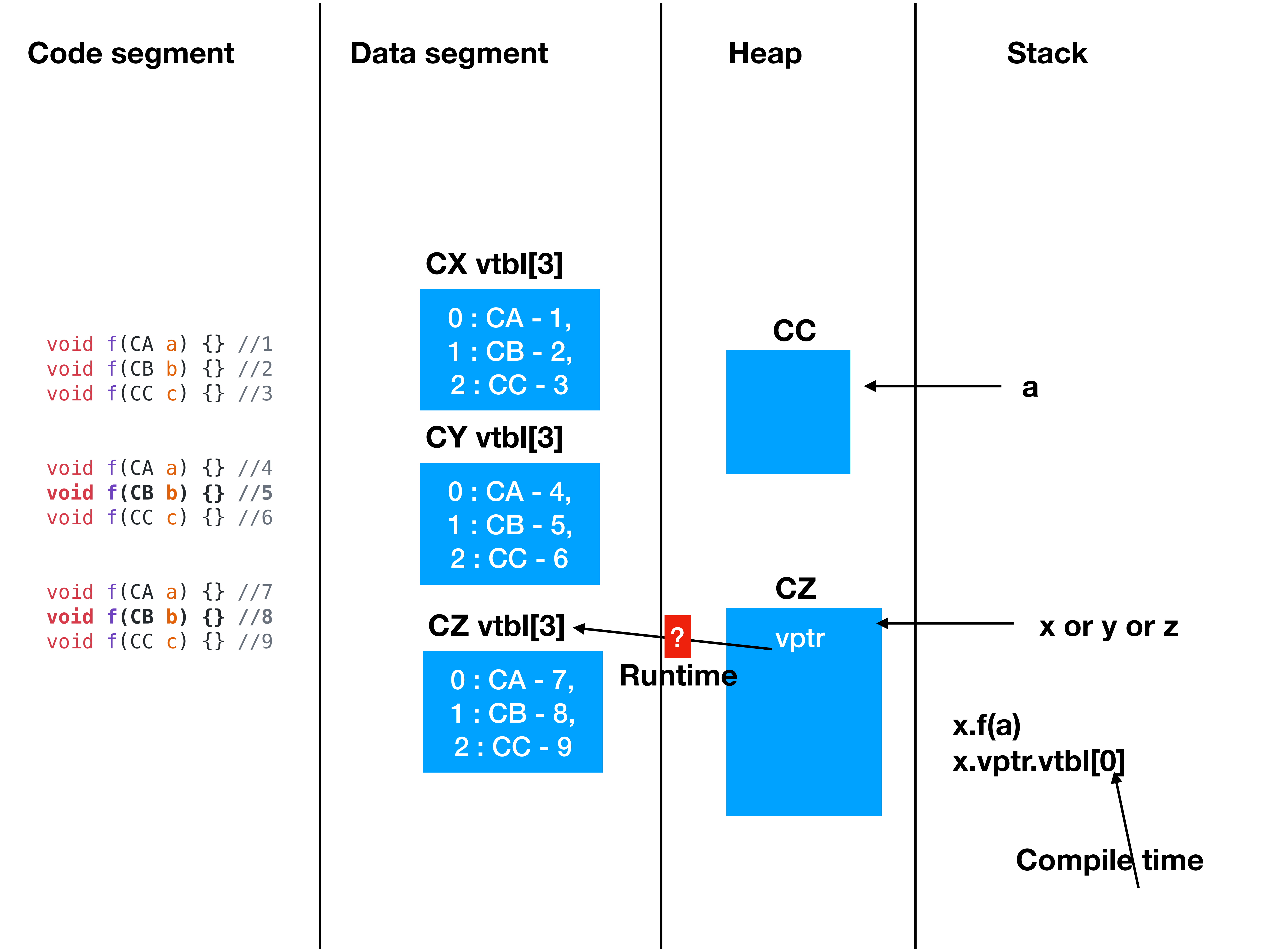
When logic cannot be kept in the Family

```
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);
  }
}
```

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


Code segment
Data segment
Heap
Stack
Util vtbl
0 : CA - 1,
1 : CB - 2,
2 : CC - 3
CC
a
void f(CA a) {} //1
void f(CB b) {} //2
void f(CC c) {} //3
Util
vptr
U
?
Runtime
u.f(a)
u.vptr.vtbl[0](a)
Compile time


Code segment
Data segment
Heap
Stack
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
CX vtbl[3]
0 : CA - 1,
1 : CB - 2,
2 : CC - 3
CY vtbl[3]
0 : CA - 4,
1 : CB - 5,
2 : CC - 6
CZ vtbl[3]
0 : CA - 7,
1 : CB - 8,
2 : CC - 9
CC
a
CZ
vptr
x or y or z
?
Runtime
x.f(a)
x.vptr.vtbl[0]
Compile time

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());

<<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>>

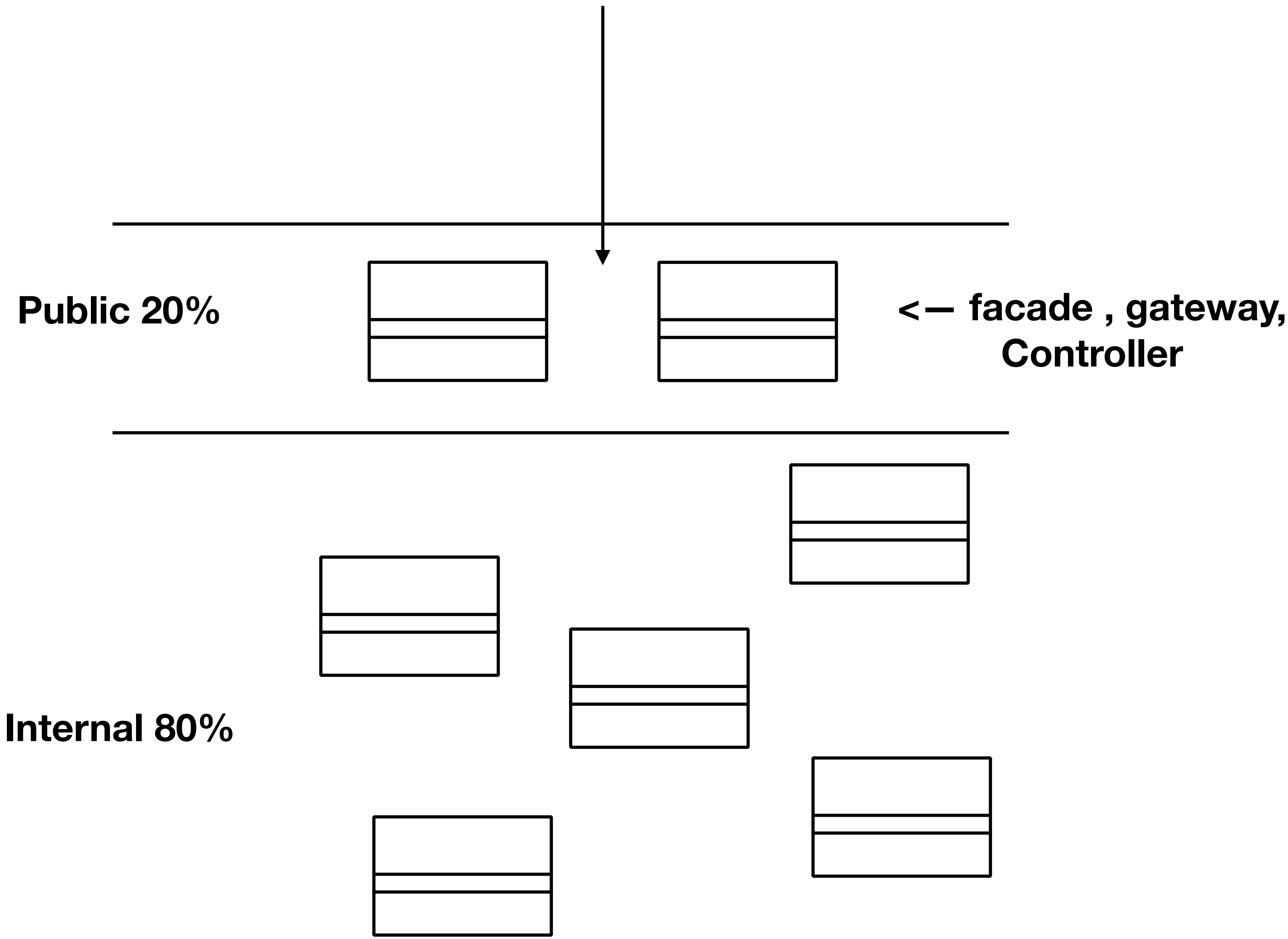
```
class Builder
{
  void addCA(){
     ....
  }
  void addCB(){
     ....
  }
  CX getCX(){
     ....
  }
}
```

<<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%