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

- **Complexity (high -> low)**
- **Coupling (high -> low)**
- **Cohesion (Low -> High)**
- Composition

- Expectations
- Years of Exp
- Technology stack

https://forms.gle/nKn4mG1bVVX1U16x8

Good

- Polymorphism/ Abstraction/ Interface/ upcast
- Exception Handling
- SRP (***)
    - Size (not too many methods
    - Cohesion
- Low Coupling (**)
- LSP
- ISP
- DRY
- Favour composition over inheritance

- KISS
- YAGNI
- OCP
- DDD Aggregates
- CC < 10
- Efferent coupling < 6
- Boundary Control Entity (*)
- DBC

Bad

- If/switch
- Flag
- Overloading family of types
- Checking a type
- Downcast
- Magic numbers/strings
- Functional interface (tiny class)
- Arrow code
- Avoid Inheritance (extends)

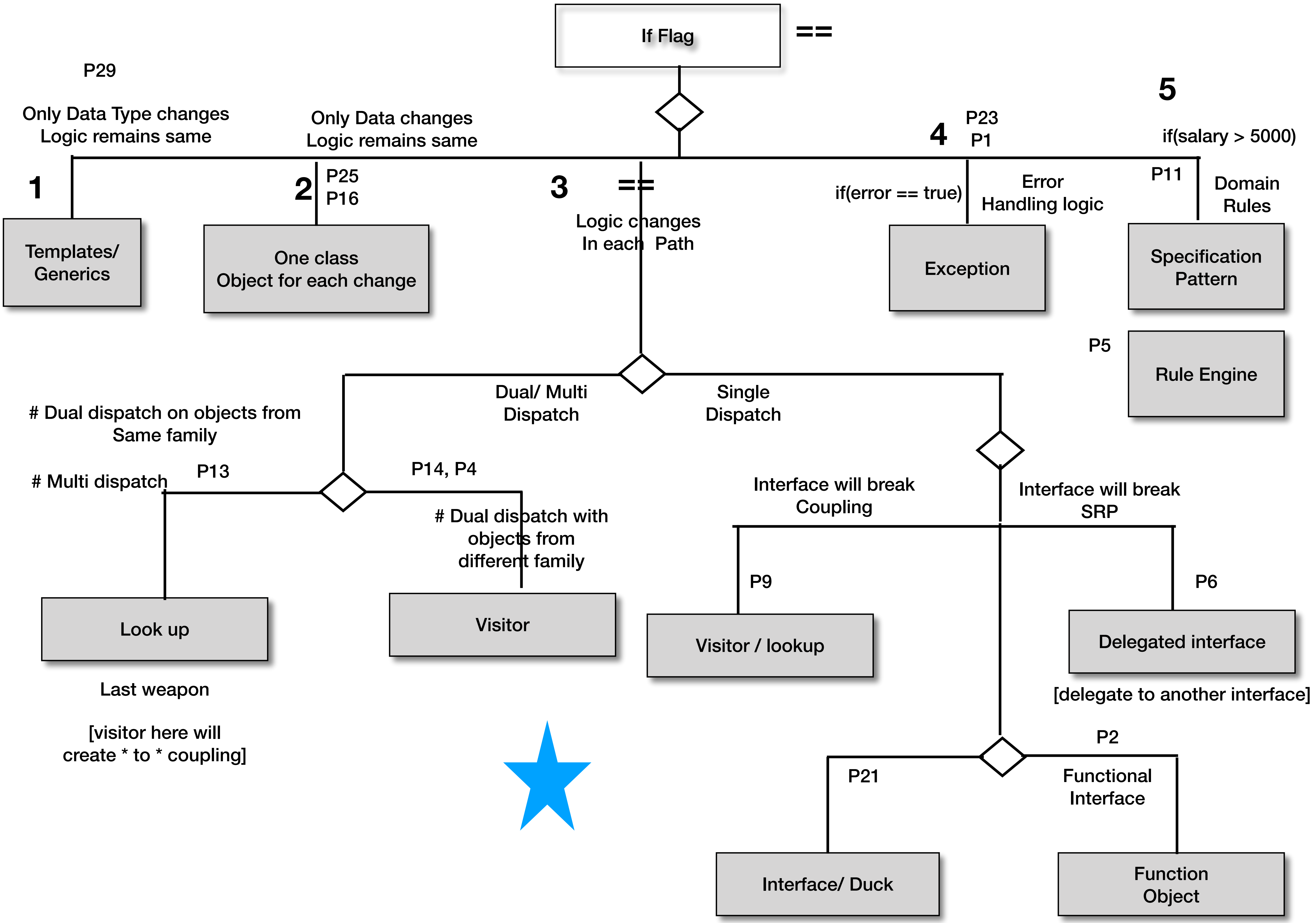
- Error handling
    - Bool, int, null,
- Duplicate Code
- Commented Code
- Dead Code
- Static methods
- God Class
- Coupling
    - Tight coupling between units
    - * to *
    - Bi directional coupling

Size

- Module
  - Max class : 25
- Class / Interface
  - Max public methods : ~12
  - Avg : ~4
- Fun
  - Max : fit screen
  - Avg : 10 lines

SOC

- Things which do not change together should not be kept together
- Domain logic and error handling logic
- Domain logic and boundary logic
- Domain Rule and Domain Logic
- Domain flow and Domain step



If Flag

==

P29

Only Data Type changes
Logic remains same

1

Templates/
Generics

Only Data changes
Logic remains same

2

P25
P16

One class
Object for each change

3

==

Logic changes
In each Path

4

P23
P1

if(error == true)

Error
Handling logic

Exception

5

if(salary > 5000)

P11

Domain
Rules

Specification
Pattern

P5

Rule Engine

Dual/ Multi
Dispatch

# Dual dispatch on objects from
Same family

# Multi dispatch

P13

Look up

Last weapon

[visitor here will
create * to * coupling]

P14, P4

# Dual dispatch with
objects from
different family

Visitor

Single
Dispatch

Interface will break
Coupling

P9

Visitor / lookup

Interface will break
SRP

P6

Delegated interface

[delegate to another interface]

P21

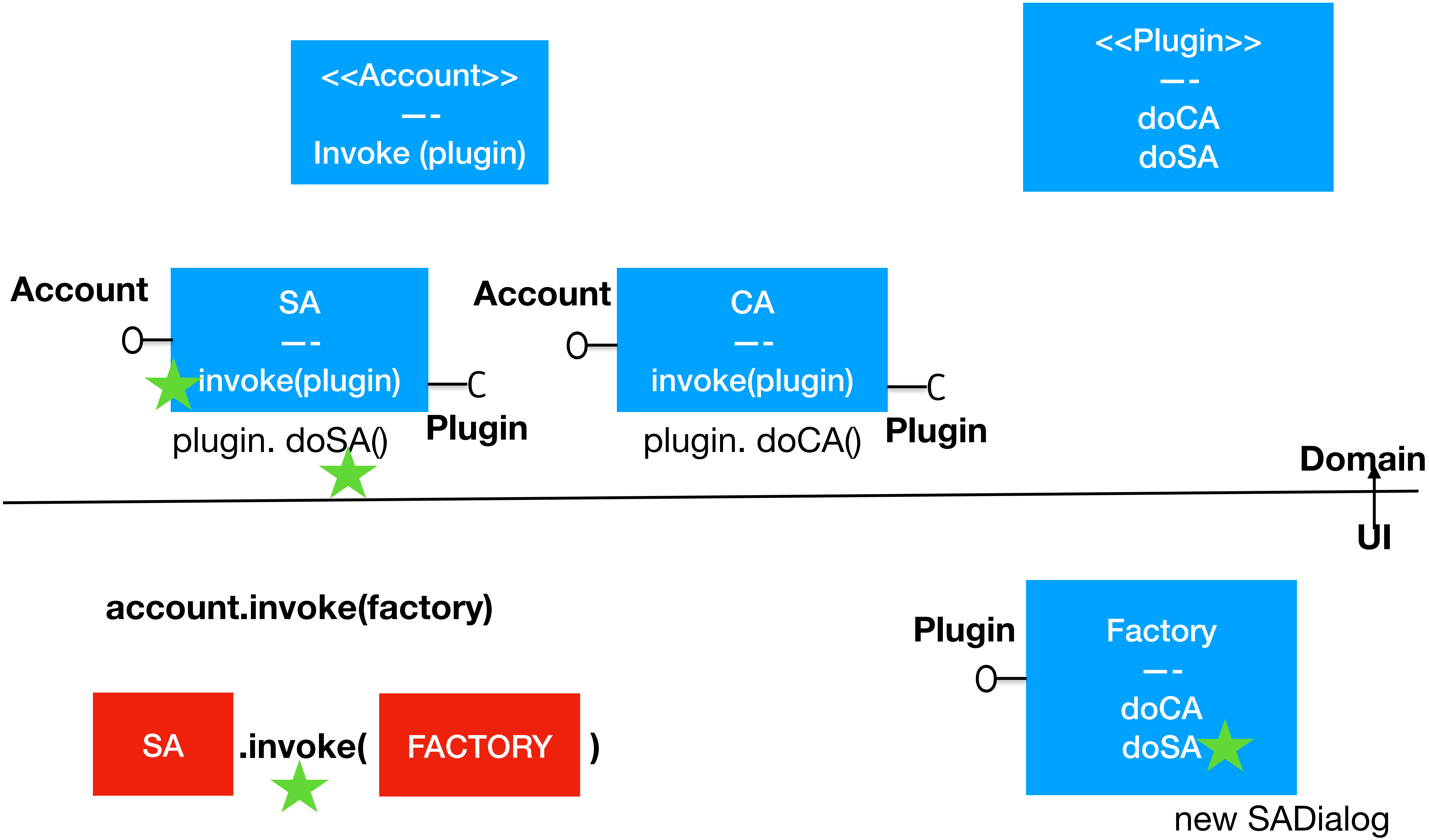
Interface/ Duck

P2

Functional
Interface

Function
Object

	1	2	3
Type of Coupling	**Method call**	**Instantiation**	**Deallocation**
Examples of coupling	Emp obj … obj.fun();	new Emp()	Emp obj … delete obj;
Approach for Low coupling	Abstraction # Interface typing * # Duck typing # Lamda	# DI * # factory	# smart pointers # virtual destructor
Xtreme Approach	# wrapper (Adapter) # reflection	# reflection	# Garbage collector



<<Account>>

—-

Invoke (plugin)

<<Plugin>>

—-

doCA

doSA

Account

SA

—-

invoke(plugin)

Plugin

plugin. doSA()

Account

CA

—-

invoke(plugin)

Plugin

plugin. doCA()

Domain

UI

account.invoke(factory)

SA .invoke(FACTORY)

Plugin

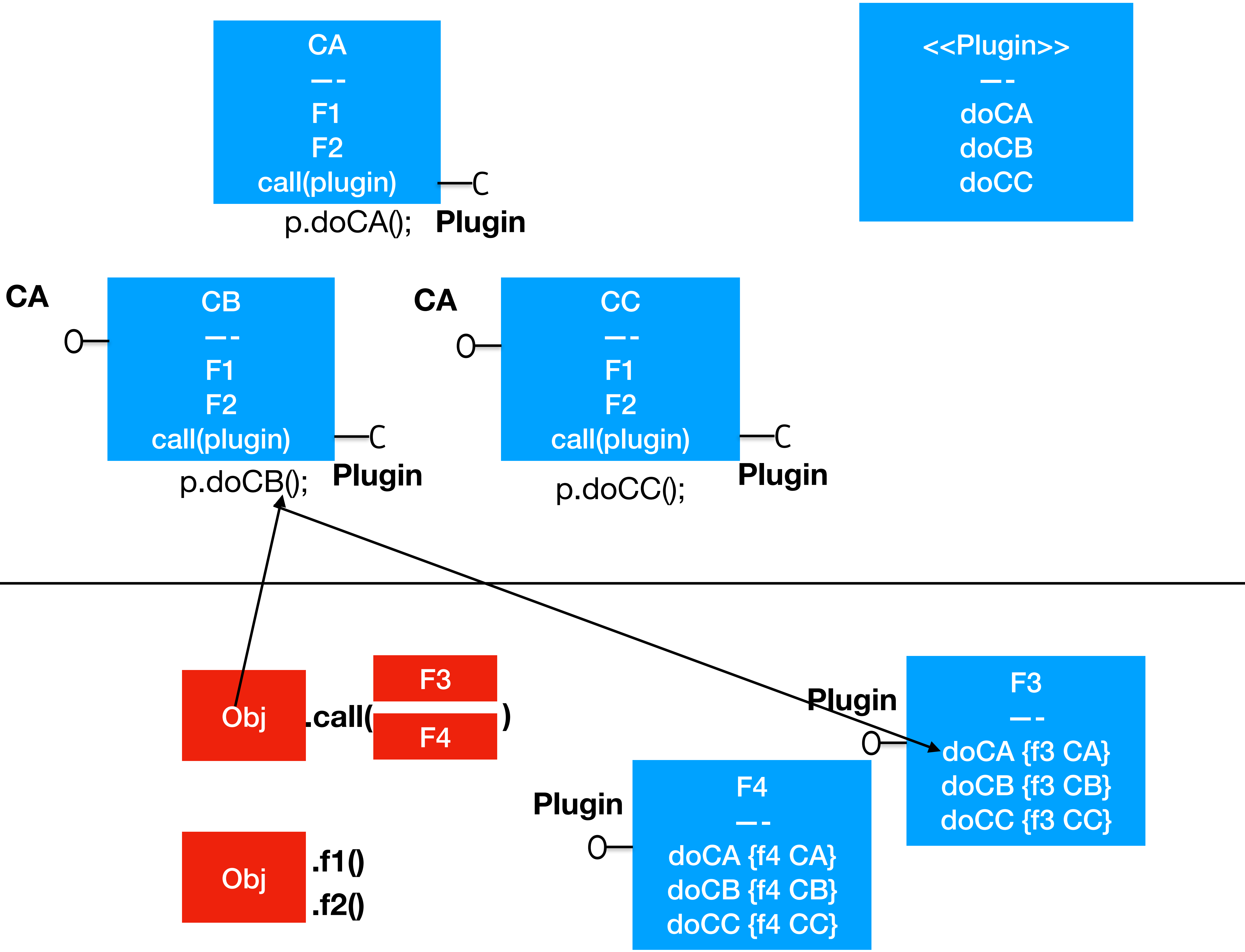
Factory

—-

doCA

doSA

new SADialog



CA
—-
F1
F2
call(plugin)

p.doCA(); Plugin

<<Plugin>>
—-
doCA
doCB
doCC

CA

CB
—-
F1
F2
call(plugin)

p.doCB(); Plugin

CA

CC
—-
F1
F2
call(plugin)

p.doCC(); Plugin

Obj .call(F3 F4)

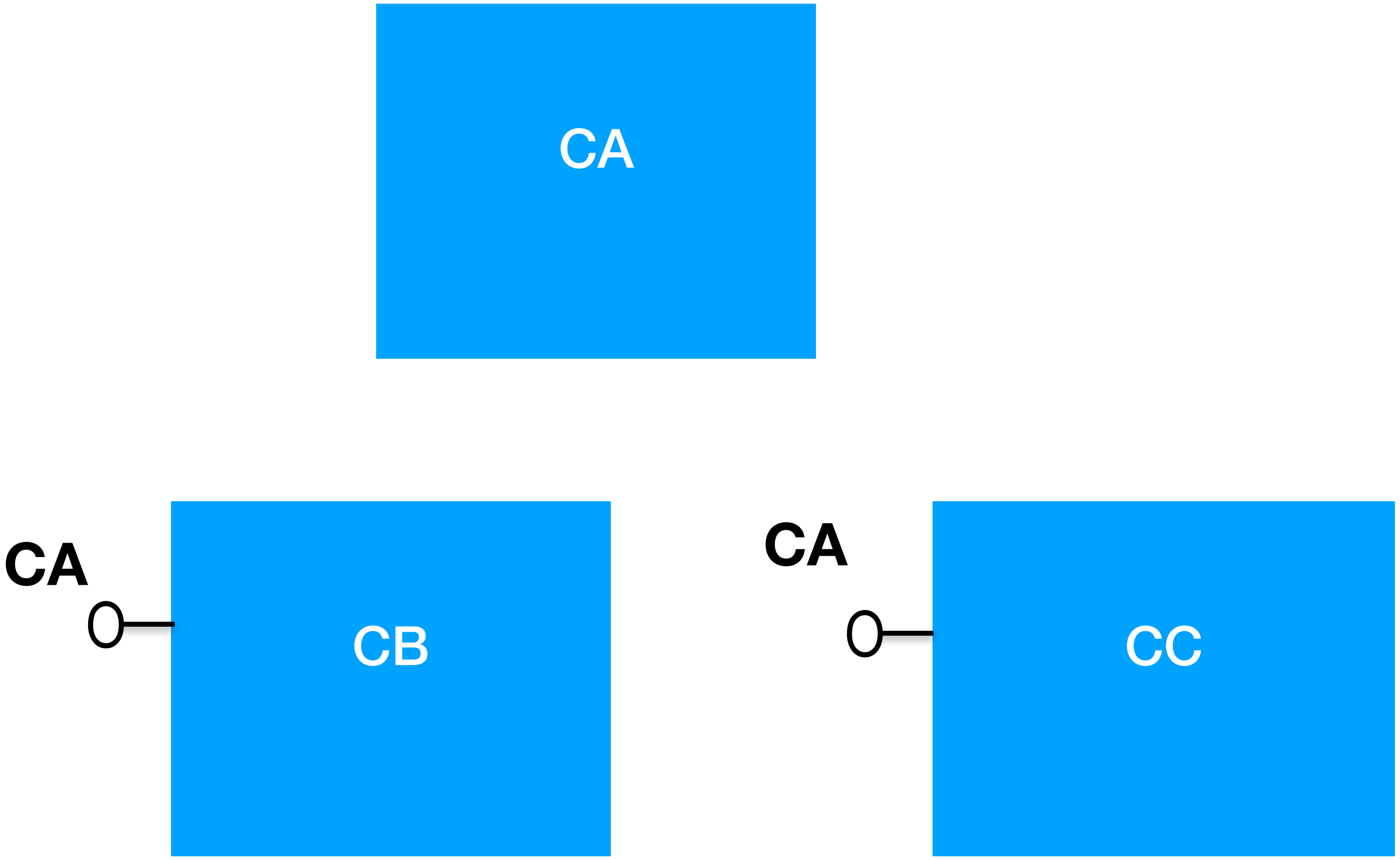
Obj .f1() .f2()

Plugin

F3
—-
doCA {f3 CA}
doCB {f3 CB}
doCC {f3 CC}

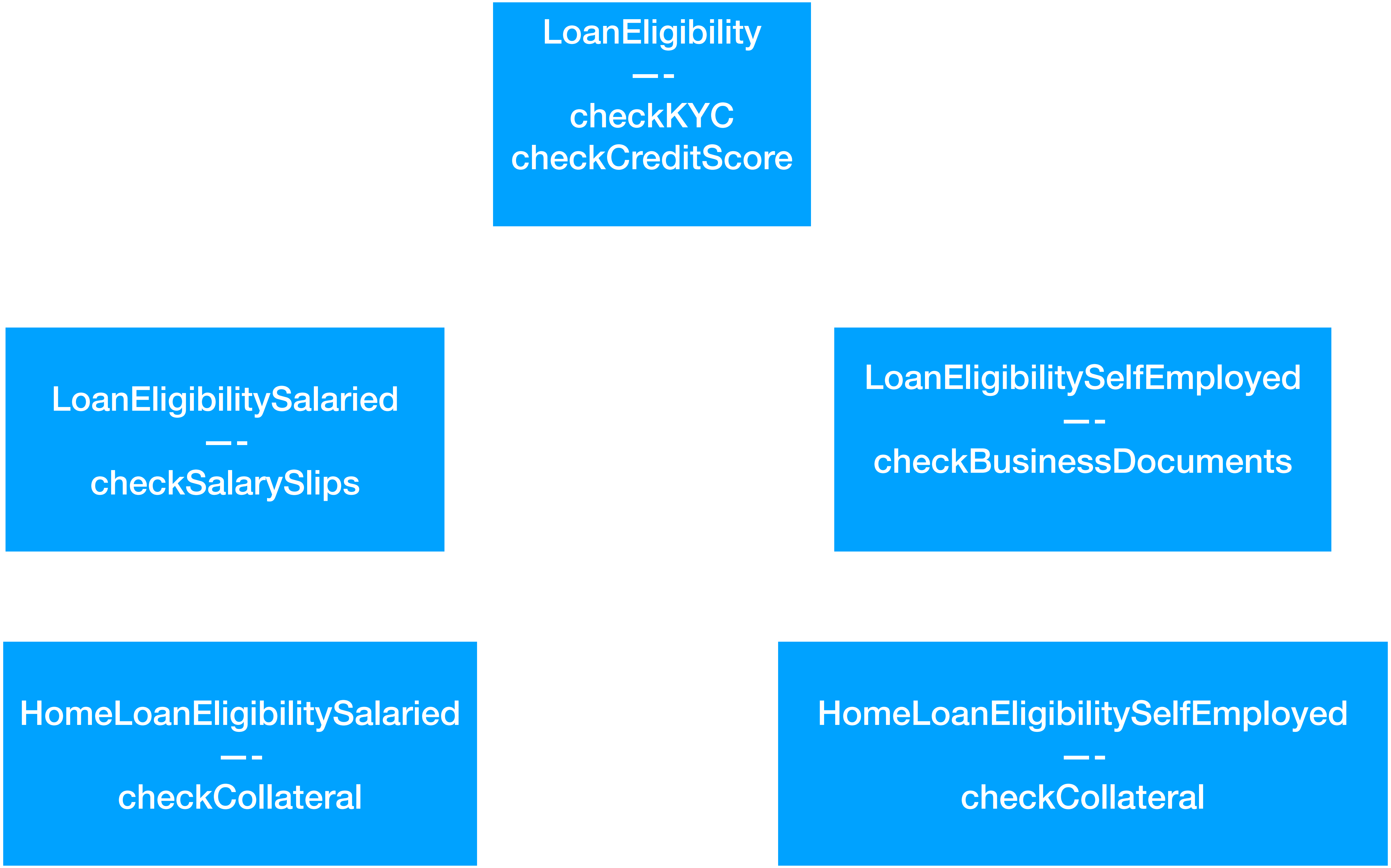
Plugin

F4
—-
doCA {f4 CA}
doCB {f4 CB}
doCC {f4 CC}



F1 (inside)
F2 (inside)
F3 (outside)
F4 (outside)

Srp
Coupling
Consumer extend



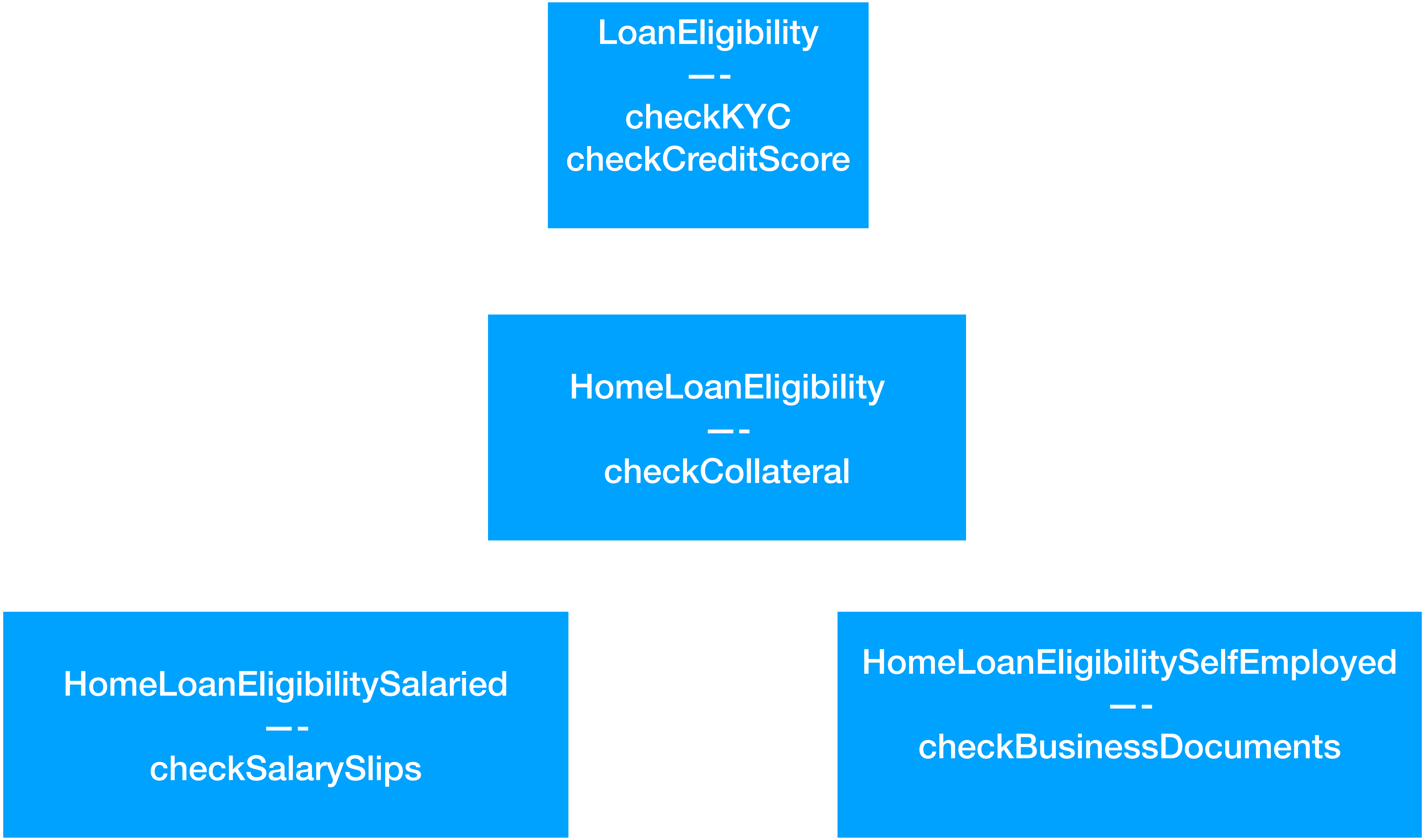
LoanEligibility
—-
checkKYC
checkCreditScore

LoanEligibilitySalaried
—-
checkSalarySlips

LoanEligibilitySelfEmployed
—-
checkBusinessDocuments

HomeLoanEligibilitySalaried
—-
checkCollateral

HomeLoanEligibilitySelfEmployed
—-
checkCollateral

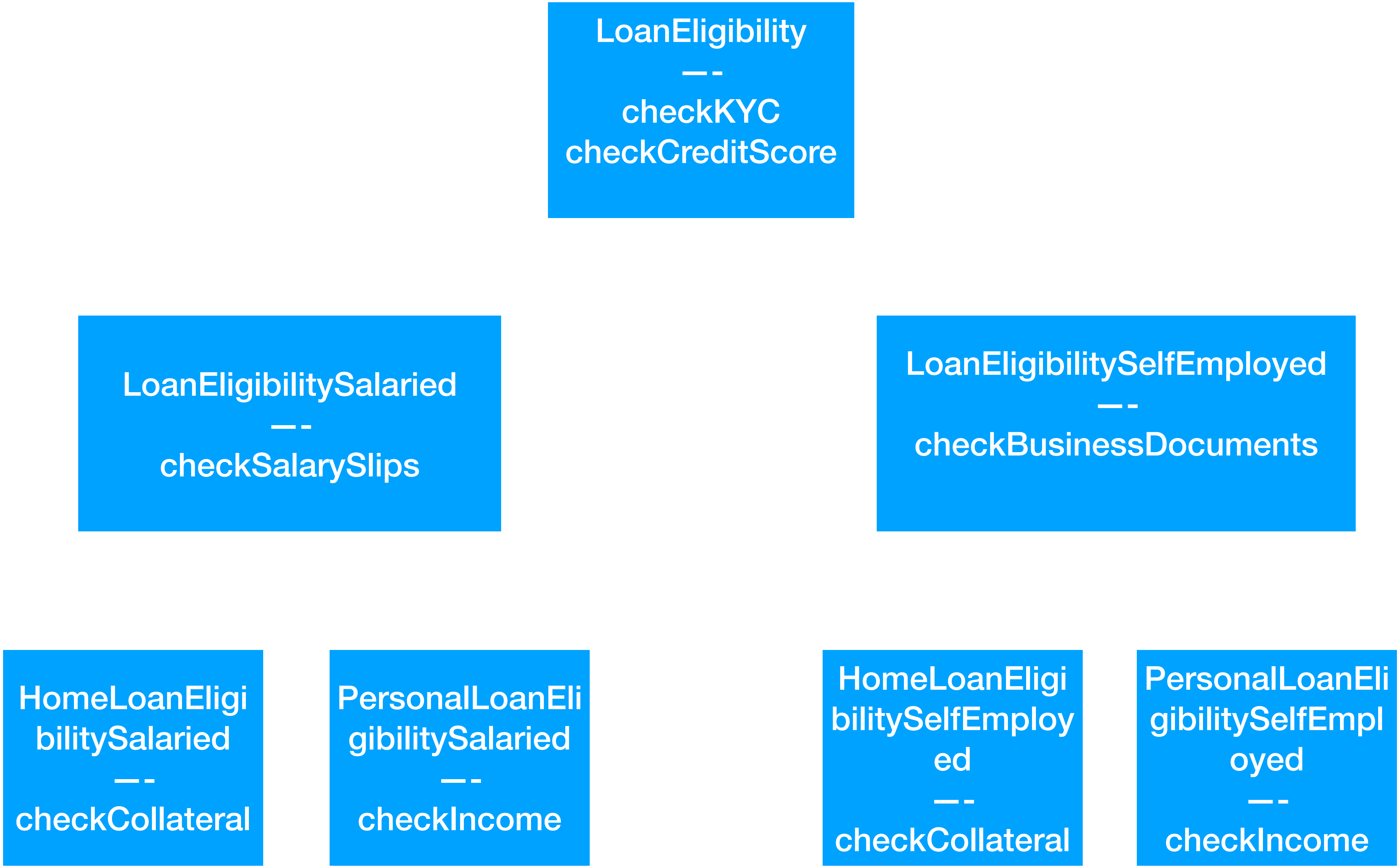


LoanEligibility
—-
checkKYC
checkCreditScore

HomeLoanEligibility
—-
checkCollateral

HomeLoanEligibilitySalaried
—-
checkSalarySlips

HomeLoanEligibilitySelfEmployed
—-
checkBusinessDocuments



LoanEligibility
—-
checkKYC
checkCreditScore

LoanEligibilitySalaried
—-
checkSalarySlips

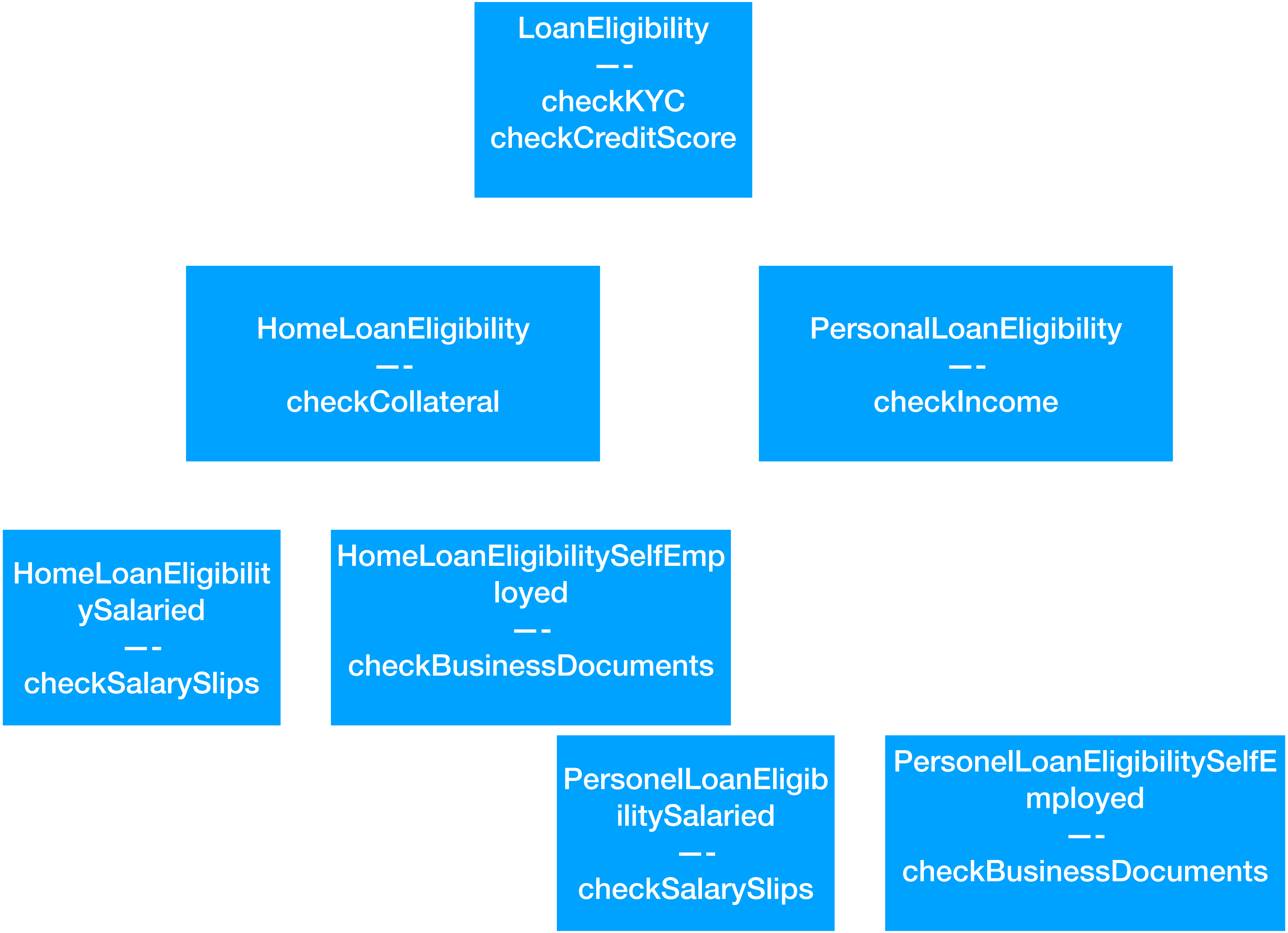
LoanEligibilitySelfEmployed
—-
checkBusinessDocuments

HomeLoanEligibilitySalaried
—-
checkCollateral

PersonalLoanEligibilitySalaried
—-
checkIncome

HomeLoanEligibilitySelfEmployed
—-
checkCollateral

PersonalLoanEligibilitySelfEmployed
—-
checkIncome



LoanEligibility
—-
checkKYC
checkCreditScore

HomeLoanEligibility
—-
checkCollateral

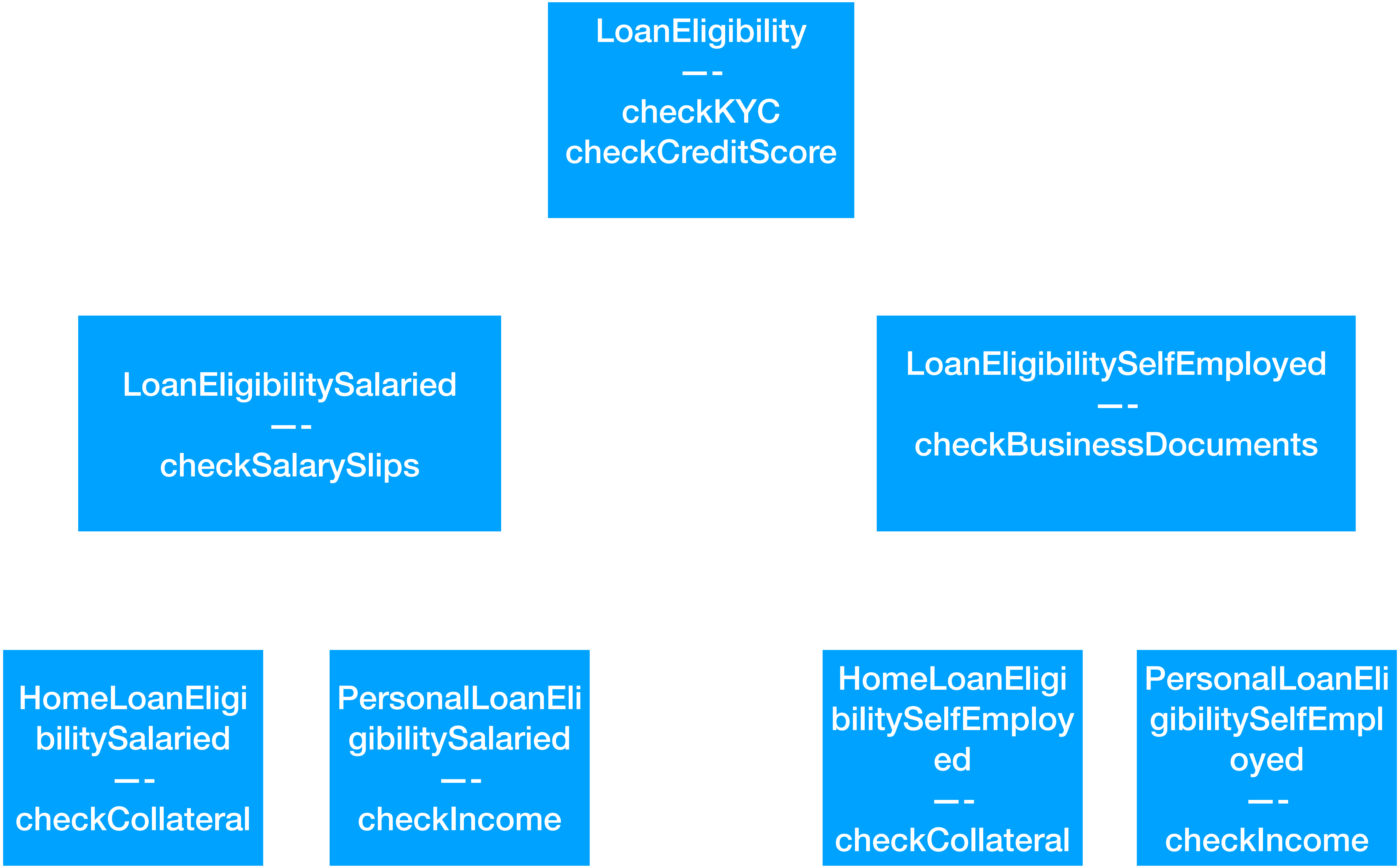
PersonalLoanEligibility
—-
checkIncome

HomeLoanEligibilitySalaried
—-
checkSalarySlips

HomeLoanEligibilitySelfEmployed
—-
checkBusinessDocuments

PersonelLoanEligibilitySalaried
—-
checkSalarySlips

PersonelLoanEligibilitySelfEmployed
—-
checkBusinessDocuments



LoanEligibility
—-
checkKYC
checkCreditScore

LoanEligibilitySalaried
—-
checkSalarySlips

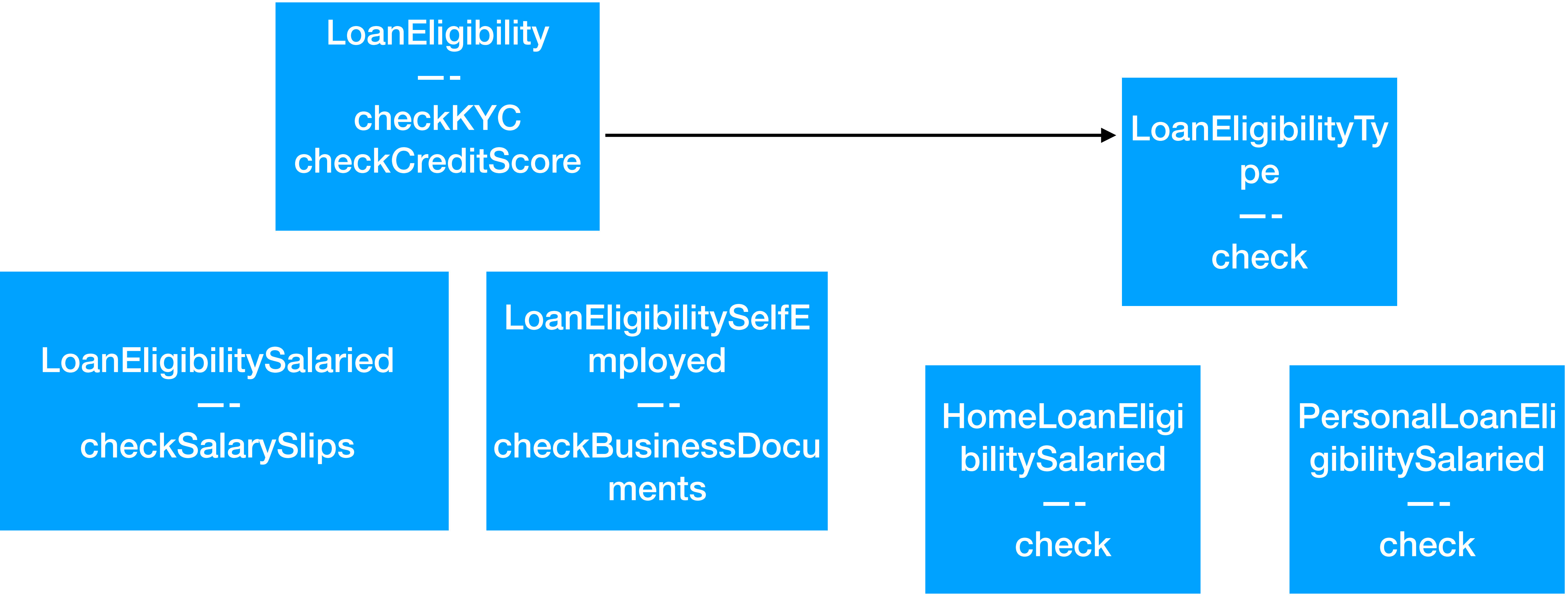
LoanEligibilitySelfEmployed
—-
checkBusinessDocuments

HomeLoanEligibilitySalaried
—-
checkCollateral

PersonalLoanEligibilitySalaried
—-
checkIncome

HomeLoanEligibilitySelfEmployed
—-
checkCollateral

PersonalLoanEligibilitySelfEmployed
—-
checkIncome



LoanEligibility
—-
checkKYC
checkCreditScore

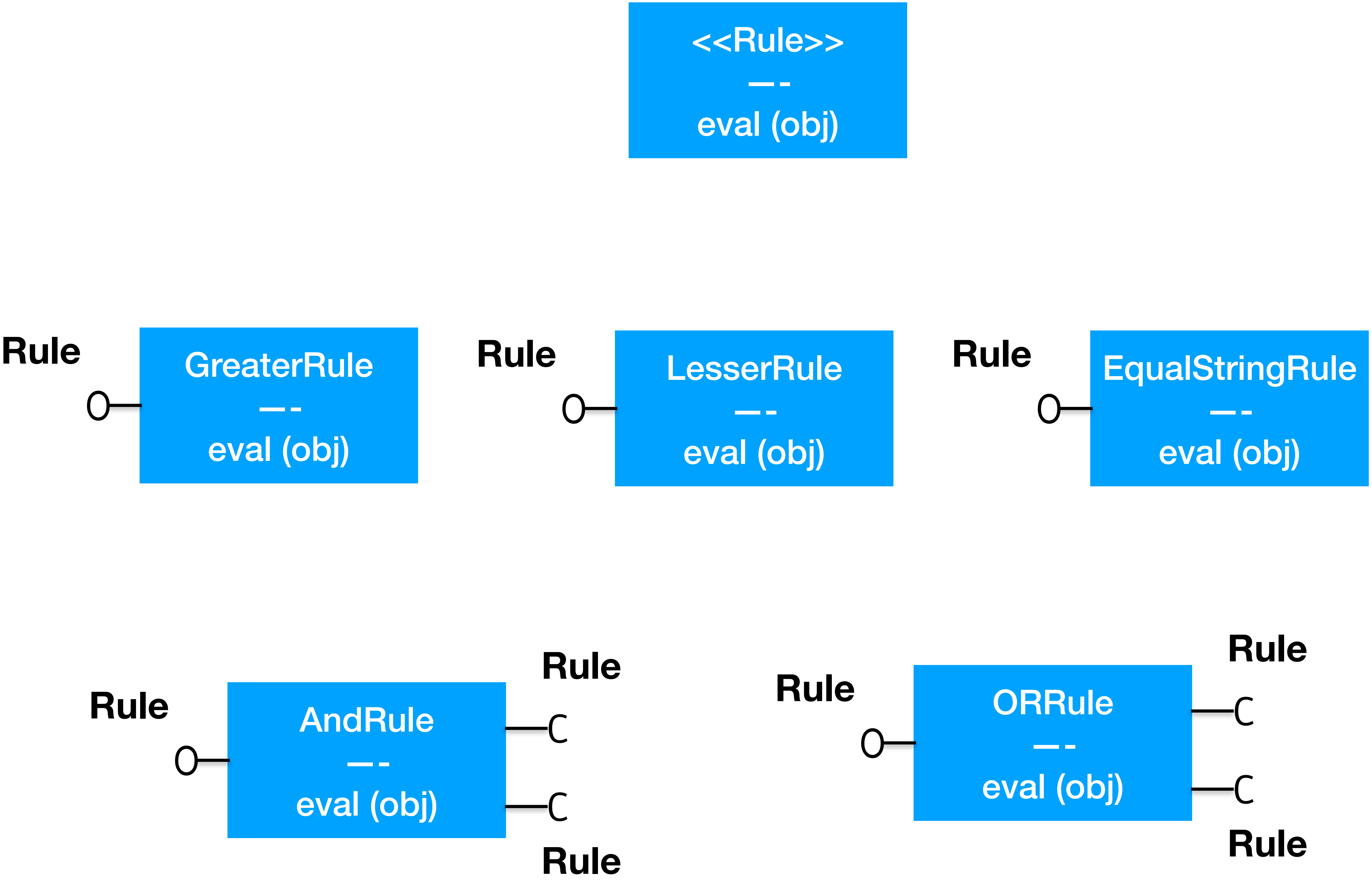
LoanEligibilityType
—-
check

LoanEligibilitySalaried
—-
checkSalarySlips

LoanEligibilitySelfEmployed
—-
checkBusinessDocuments

HomeLoanEligibilitySalaried
—-
check

PersonalLoanEligibilitySalaried
—-
check



<<Rule>>
—-
eval (obj)

Rule

GreaterRule
—-
eval (obj)

Rule

LesserRule
—-
eval (obj)

Rule

EqualStringRule
—-
eval (obj)

Rule

AndRule
—-
eval (obj)

Rule

Rule

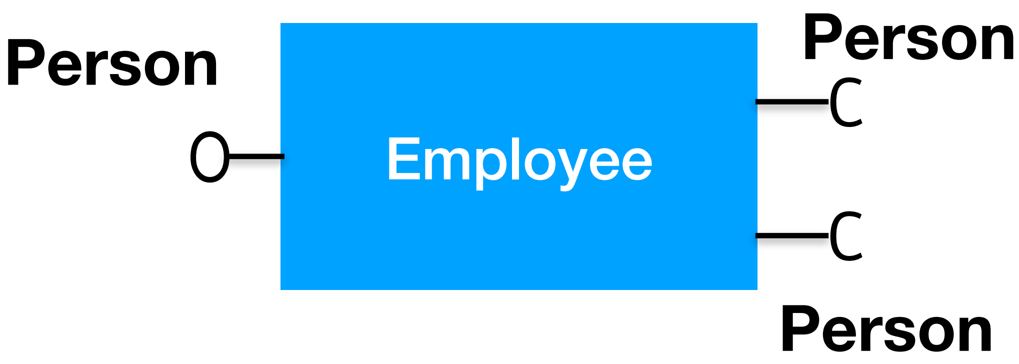
Rule

ORRule
—-
eval (obj)

Rule

Rule

<<Person>>

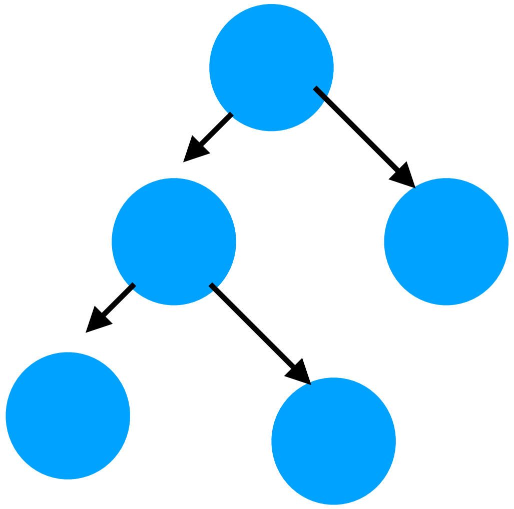


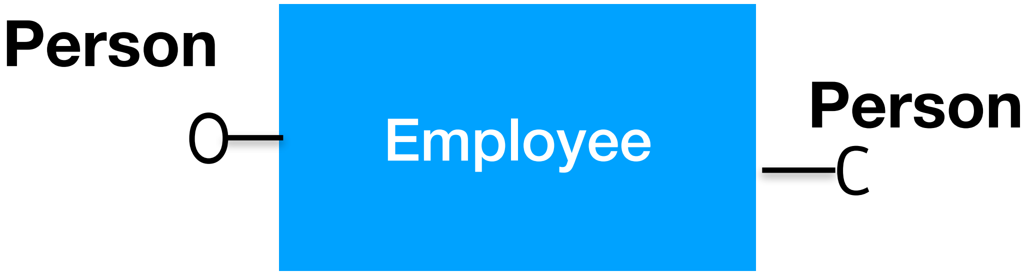
Person

Employee

Person

Person

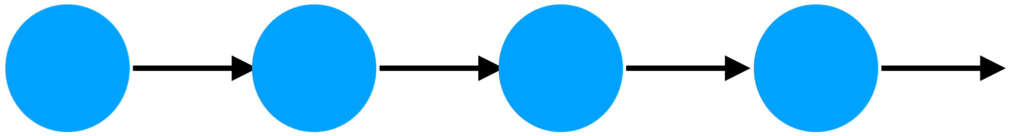




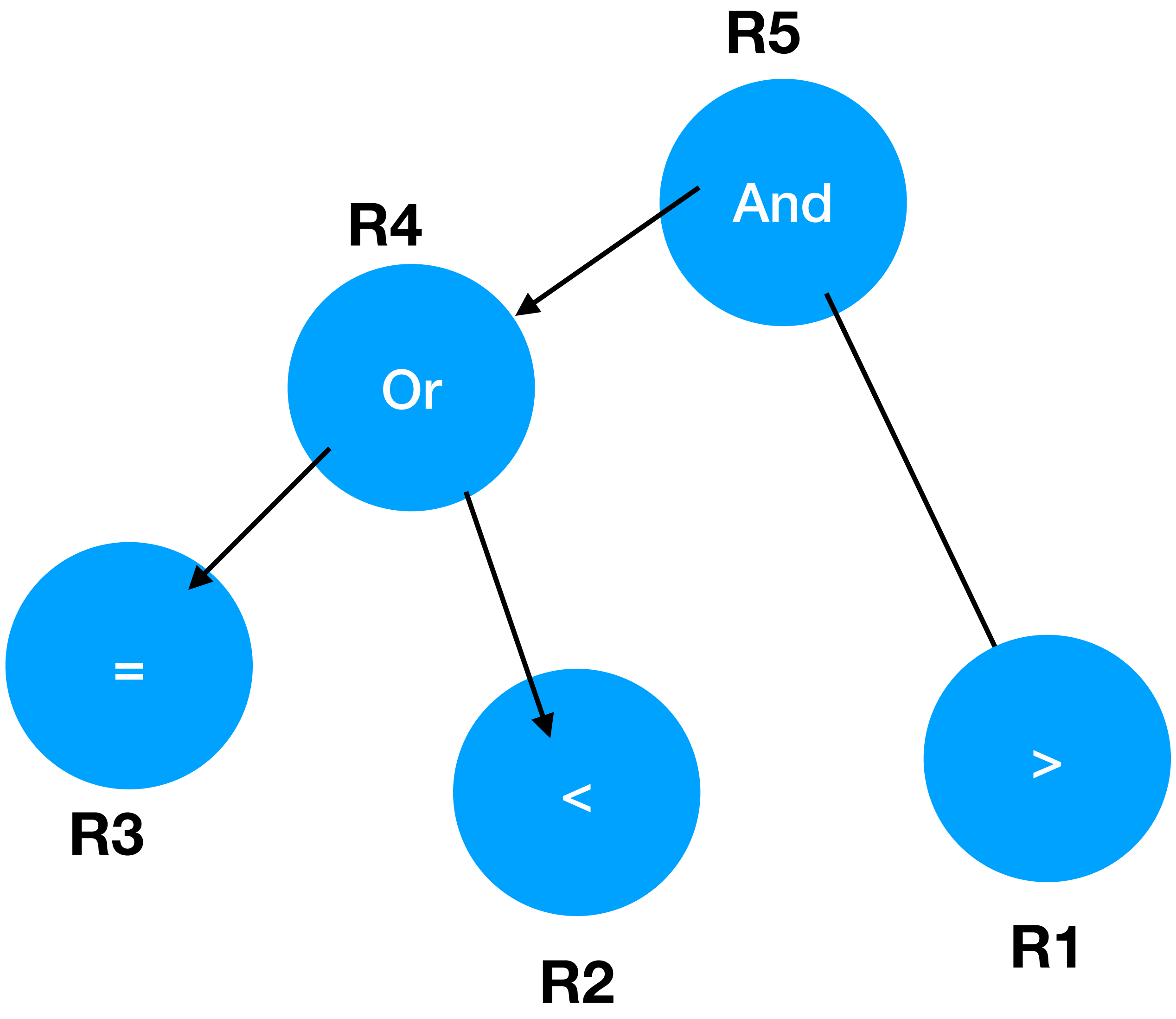
Person

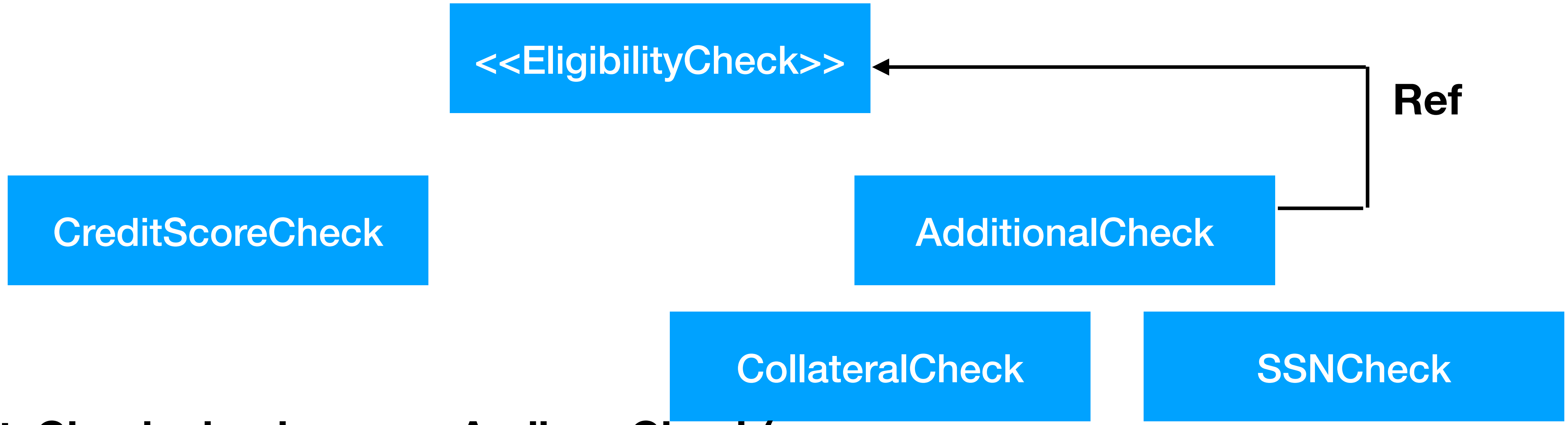
Employee

Person



```
object = {name : “jack”, salary:5000, age:10, location:”CA”}

Rule rule1 = new GreaterRule(“salary”, 5000);
Rule rule2 = new LesserRule(“age”, 35);
Rule rule3 = new StringEqual(“location”, “NY”);
Rule rule4 = new OrRule(rule2,rule3);
Rule rule5 = new AndRule(rule1,rule4);

bool res = rule5.eval(object);
```





<<EligibilityCheck>>

Ref

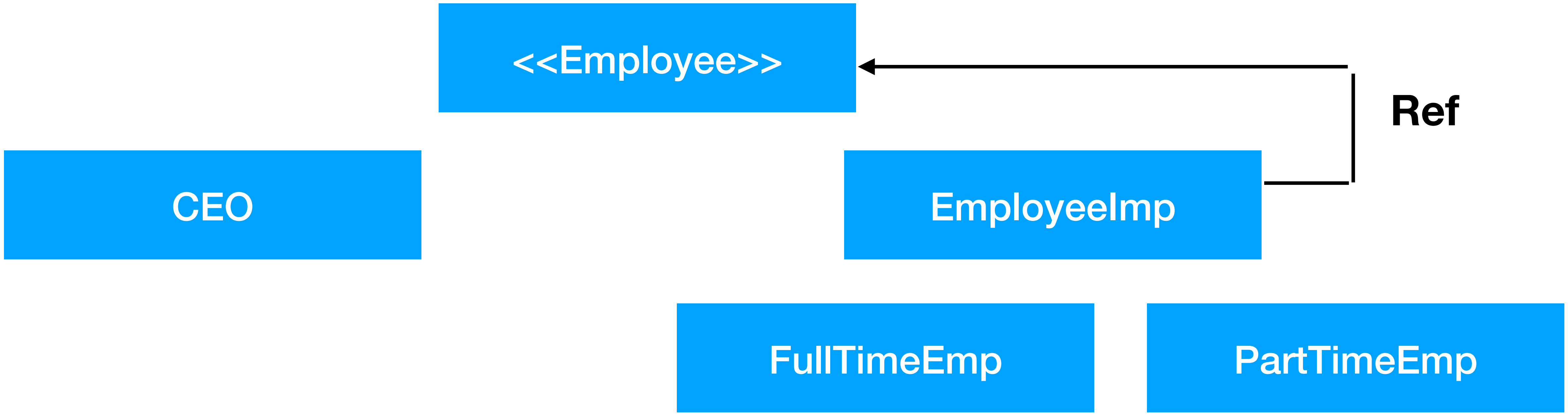
CreditScoreCheck

AdditionalCheck

CollateralCheck

SSNCheck

```
EligibilityCheck checks= new AadhaarCheck(
                         new CollateralCheck(
                          new SalarySlipCheck(
                               new KYCCheck(
                        new CreditScoreCheck())))));
```



<<Employee>>

Ref

CEO

EmployeeImp

FullTimeEmp

PartTimeEmp

```
Employee emp = New FullTimeEmp(new FullTimeEmp(new CEO()))
```

No discrimination

```
Interface Diabetes{
 ….
}
```

```
Interface Bird{
      Chirp
     Migrate
      Swim
         }
```

```
Class Parrot{
 ….
}
```

```
Interface Disease{
 ….
}
```

```
fun(Bird bird){

    If type(Bird) != type(Penguin) …
         bird.fly();
    ….

}
```

```
fun(Disease disease){
   ….

}
```

```
Interface FlyingBird extends Bird{
                  Fly
                   }
```

Open for adding new code
with out changing
existing code

GOF

- Mediator
- Command
- Visitor
- Singleton
- Builder
- Factory Method
- Abstract Factory
-

Good

- SRP (***)
- Low coupling (**)
- Unit testable (*)
- DRY
- KISS
- LSP
- ISP
- OCP
- Upcasting/ Abstraction

- DDD
  - Aggregates
- Boundary control Entity
- Program to an interface
- Prefer composition over inheritance

Bad

- Flag
- Typecheck
- Downcasting
- High CC
- Overloading for family
- Magic number/string
- God class
- Functional Interface (Lilliput classes)
- Using bool, null, int for error handling
- Duplicate code
- Commented code
- Dead Code
- Static methods
- Extends
- Tight coupling
    - Cyclic coupling
    - * to * coupling

SOLID

- SRP (**)
- OCP (?)
- LSP
- ISP
- DIP

SRP

- Things which do not change together should not be kept together
- Size : Fun size/ Class Size/Module Size
- SOC

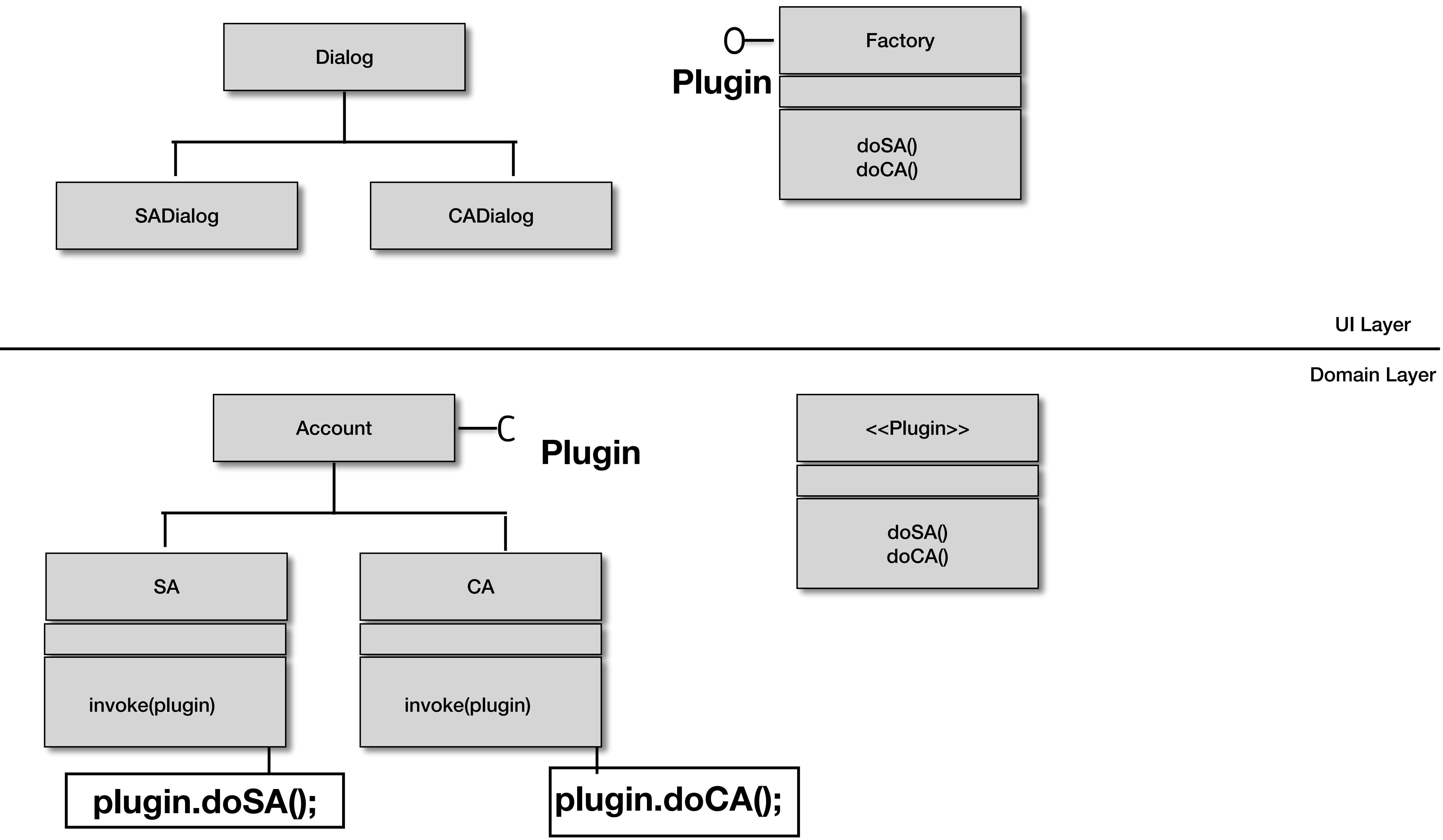
SOC

- Boundary & Entity (*)
- Error handling logic & domain logic
- Flow & steps (*)
- Domain logic & pure fabrication
- Domain logic & domain rules

500 lines of code	*(1) - dense 10 fun 50 lines each*	*(2) - sparse 50 fun 10 lines each*
Perf	~	~
Easy to Name		*
Unit test		*
Readability		*
Agility to change		*
Reusability		*

- new CA();
- New sizeof(CA)()
- new(8);
- Backward compatibility (old client + new library)

account.invoke(factory);



Dialog

SADialog

CADialog

Plugin

Factory

doSA()
doCA()

UI Layer

Domain Layer

Account

Plugin

SA

invoke(plugin)

plugin.doSA();

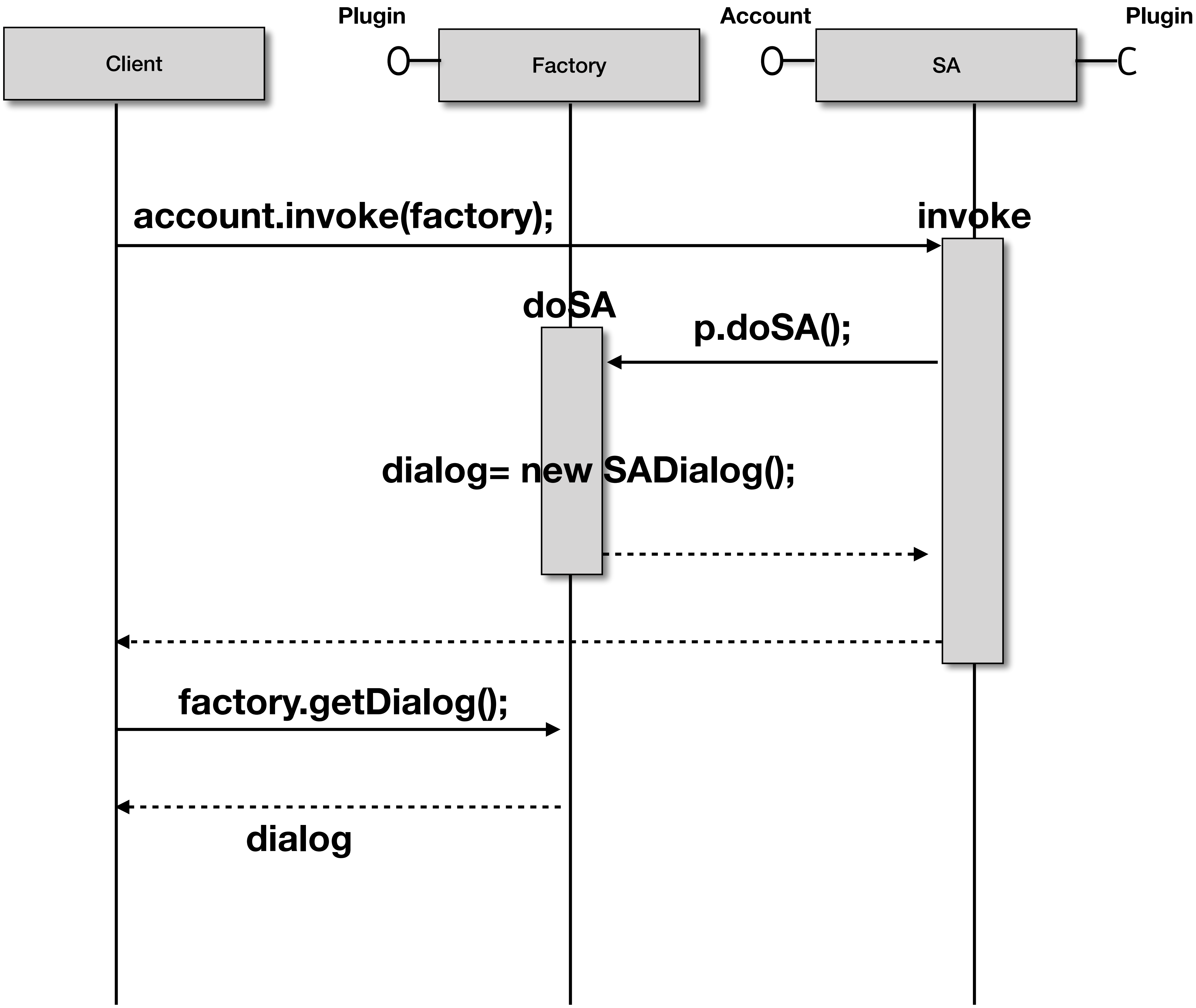
CA

invoke(plugin)

plugin.doCA();

<<Plugin>>

doSA()
doCA()



Plugin

Account

Plugin

Client

Factory

SA

account.invoke(factory);

invoke

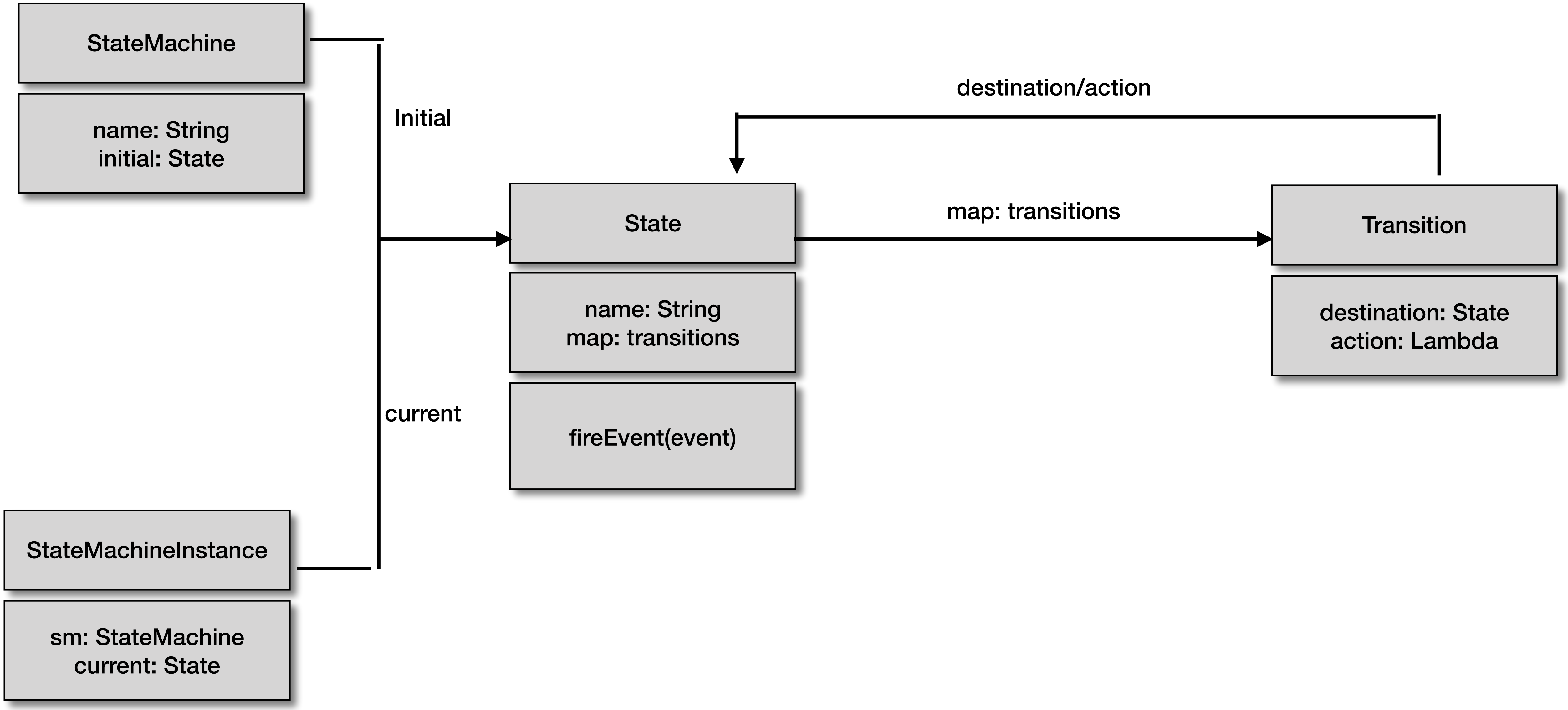
doSA

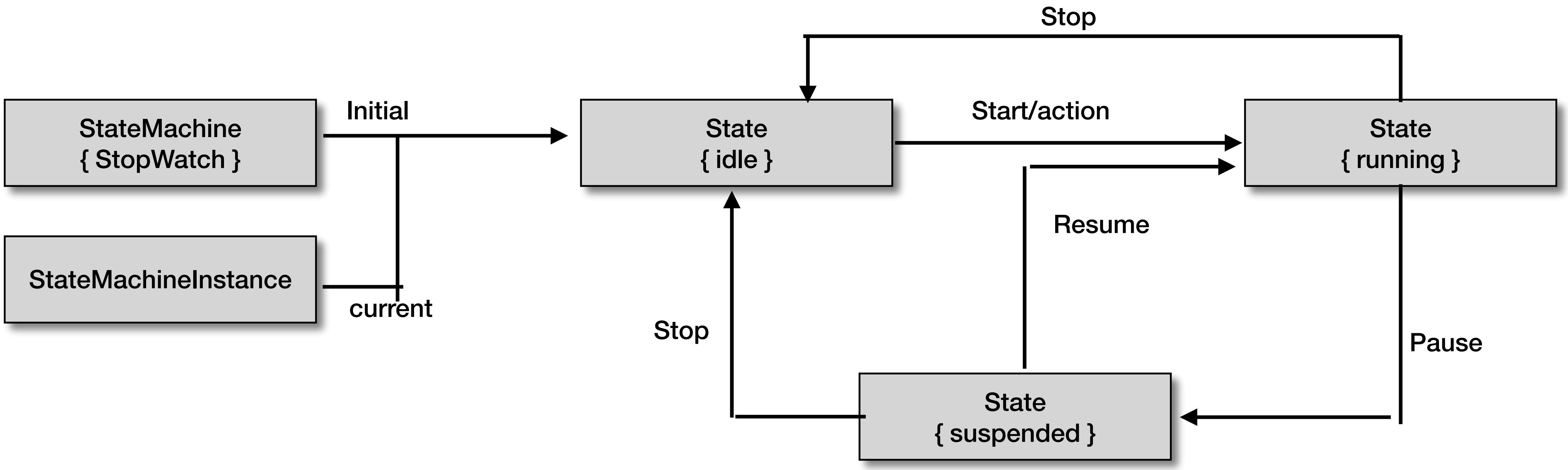
p.doSA();

dialog= new SADialog();

factory.getDialog();

dialog


StateMachine
name: String
initial: State
Initial
current
StateMachineInstance
sm: StateMachine
current: State
State
name: String
map: transitions
fireEvent(event)
map: transitions
destination/action
Transition
destination: State
action: Lambda

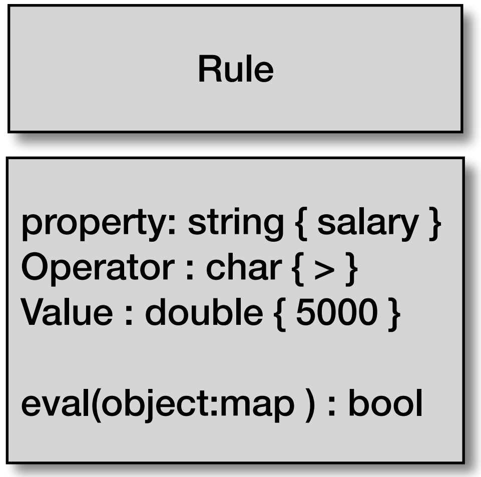

StateMachine
{ StopWatch }
StateMachineInstance
Initial
current
Stop
State
{ idle }
Start/action
State
{ running }
Resume
Stop
Pause
State
{ suspended }

- if(salary > 5000) => logic

- If(salary > 5000 & age < 21) => logic

- if(salary > 5000 & (age > 35 || location == “NY”) => logic

- if(salary > 5000)



Rule

property: string { salary }
Operator : char { > }
Value : double { 5000 }

eval(object:map) : bool

```
current = object[property]
switch(operator){
   case ‘>’ :
       return  current > value;
   case ‘=‘ :
      …
   case ‘<‘ :
       return  current < value;
       …
}
```

```
Rule rule = new Rule(“salary”,”>”,”5000”);

object = {name : “jack”, salary:5000, age:10, location:”CA”}

bool res = rule.eval(object);
```

Rule

- if(salary > 5000) => logic
- If(salary > 5000 & age < 21) => logic
- if(salary > 5000 & (age > 35 || location == “NY”) => logic

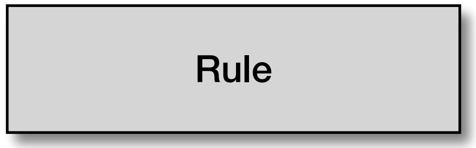
- If(salary > 5000 & age < 21) => logic

```
Rule rule1 = new Rule(“salary”,”>”,”5000”);
Rule rule2 = new Rule(“age”,”<”,”21”);

object = {name : “jack”, salary:5000, age:10, location:”CA”}

bool res1 = rule1.eval(object);
bool res2 = rule2.eval(object);
Bool res3 = rule1 & rule 2;
```

Rule

- if(salary > 5000) => logic
- If(salary > 5000 & age < 21) => logic
- if(salary > 5000 & (age > 35 || location == “NY”) => logic

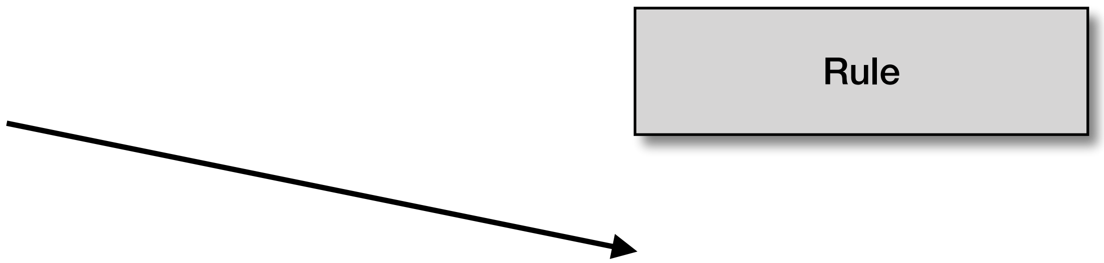
```
object = {name : “jack”, salary:5000, age:10, location:”CA”}

Rule rule = new Rule(“salary”,”>”,”5000”); //1
bool res = rule.eval(object);

Rule rule = new GreaterRule(“salary”, 5000); //2
bool res = rule.eval(object);
```



Rule

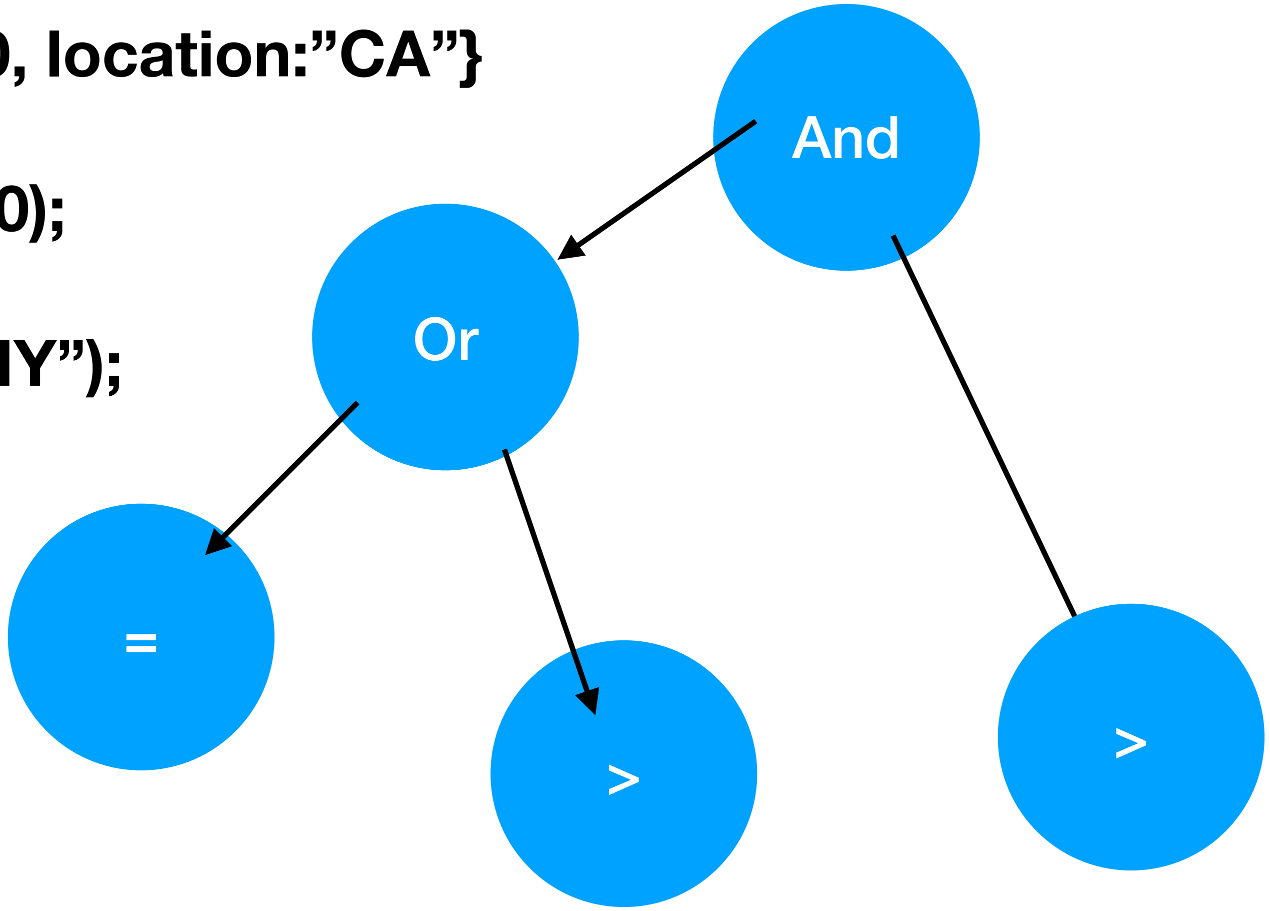
- If(salary > 5000 & age < 21)

```
object = {name : “jack”, salary:5000, age:10, location:”CA”}

Rule rule1 = new Greater(“salary”, 5000);
Rule rule2 = new Lesser(“age”, 21);
Rule rule3 = new And(rule1,rule2);

Rule rule3 = new And(new Greater(“salary”, 5000) ,new Lesser(“age”, 21));


bool res = rule3.eval(object);
```



- if(salary > 5000) => logic
- If(salary > 5000 & age < 21) => logic
- if(salary > 5000 & (age > 35 || location == “NY”) => logic

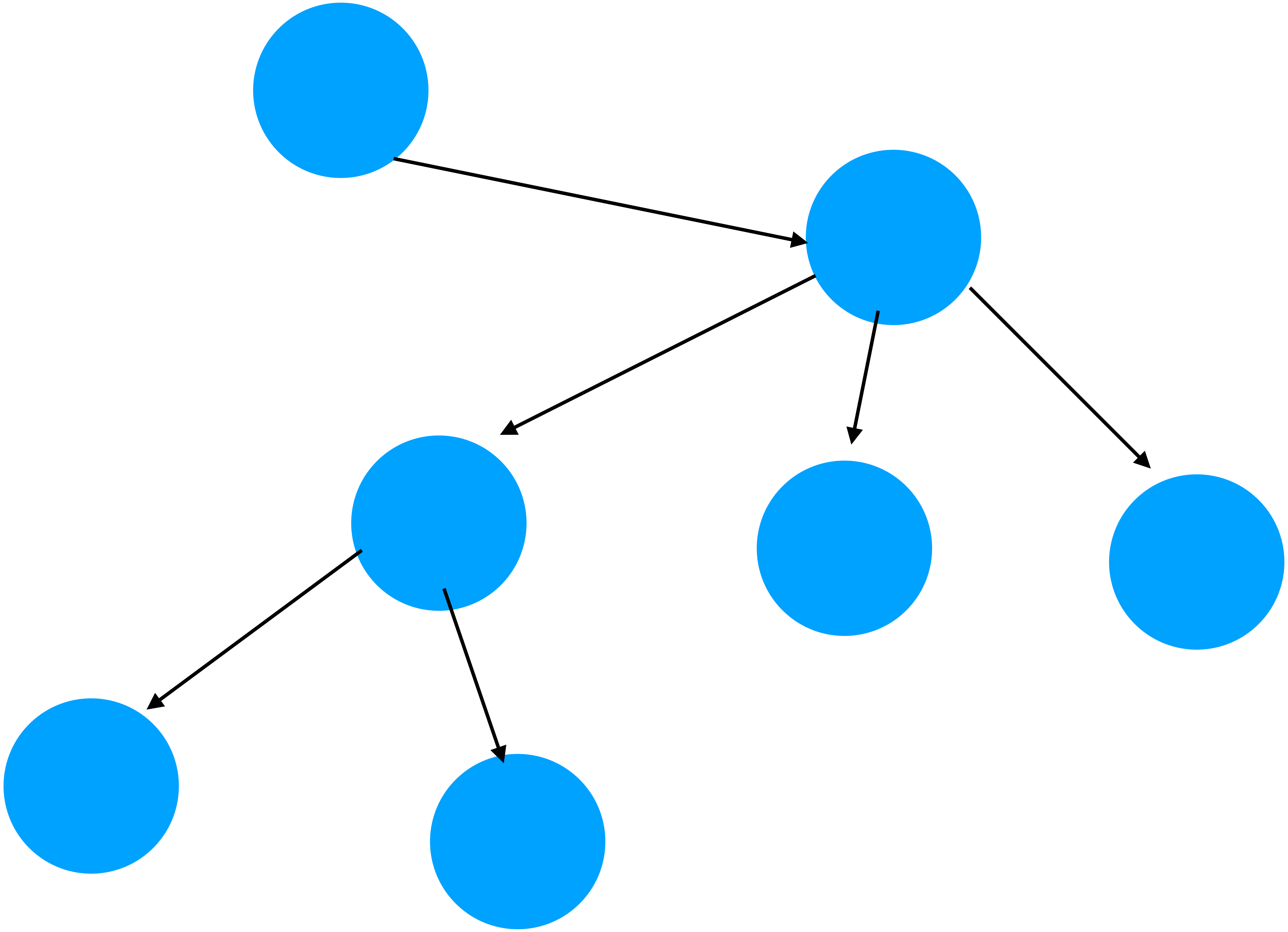
```
object = {name : “jack”, salary:5000, age:10, location:”CA”}

Rule rule1 = new GreaterRule(“salary”, 5000);
Rule rule2 = new GreaterRule(“age”, 35);
Rule rule3 = new StringEqual(“location”, “NY”);
Rule rule4 = new OrRule(rule2,rule3);
Rule rule5 = new AndRule(rule1,rule4);

bool res = rule5.eval(object);
```



```
class Emp{
List<Emp> emps;

}
```

```
class Emp{
Emp o1;
Emp o2;

}
```

```
class Emp{
Emp mgr;

}
```



#1

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

#2

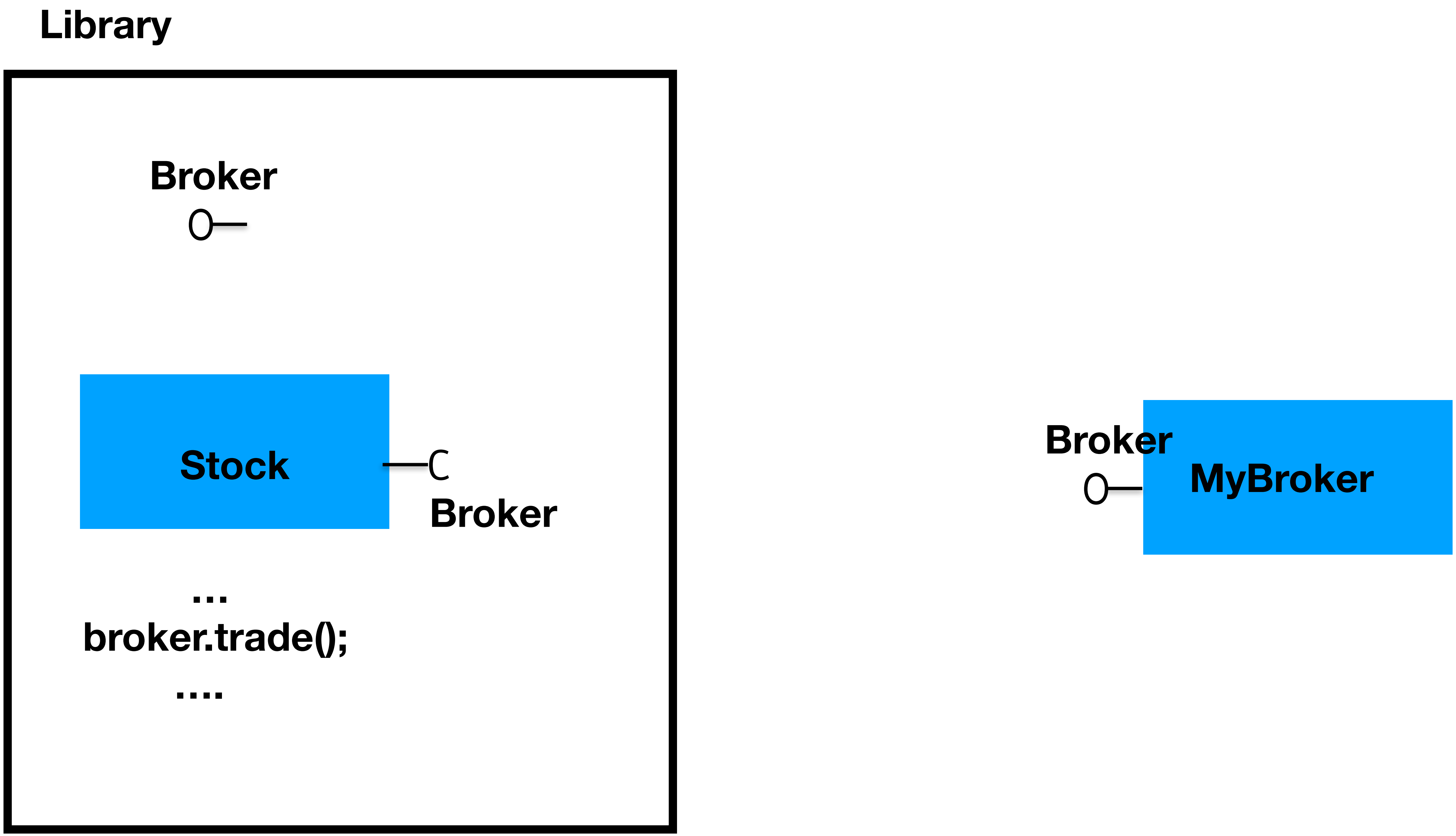
```
class Emp{
   Emp manager;
}
```

#3

```
class Emp{
   Emp a;
   Emp b;
}
```

Linked list

Graph /tree



Library

Broker

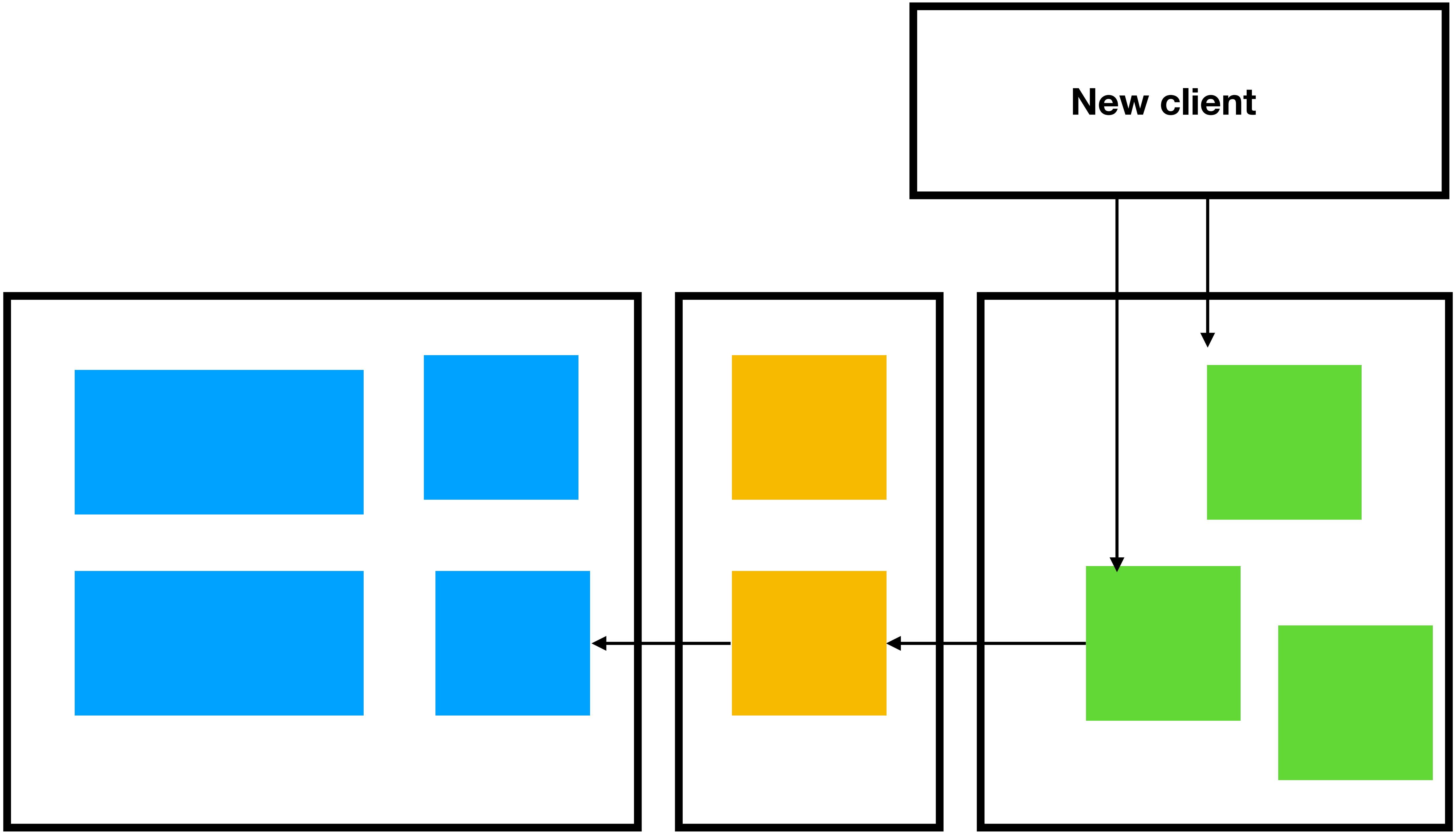
Stock

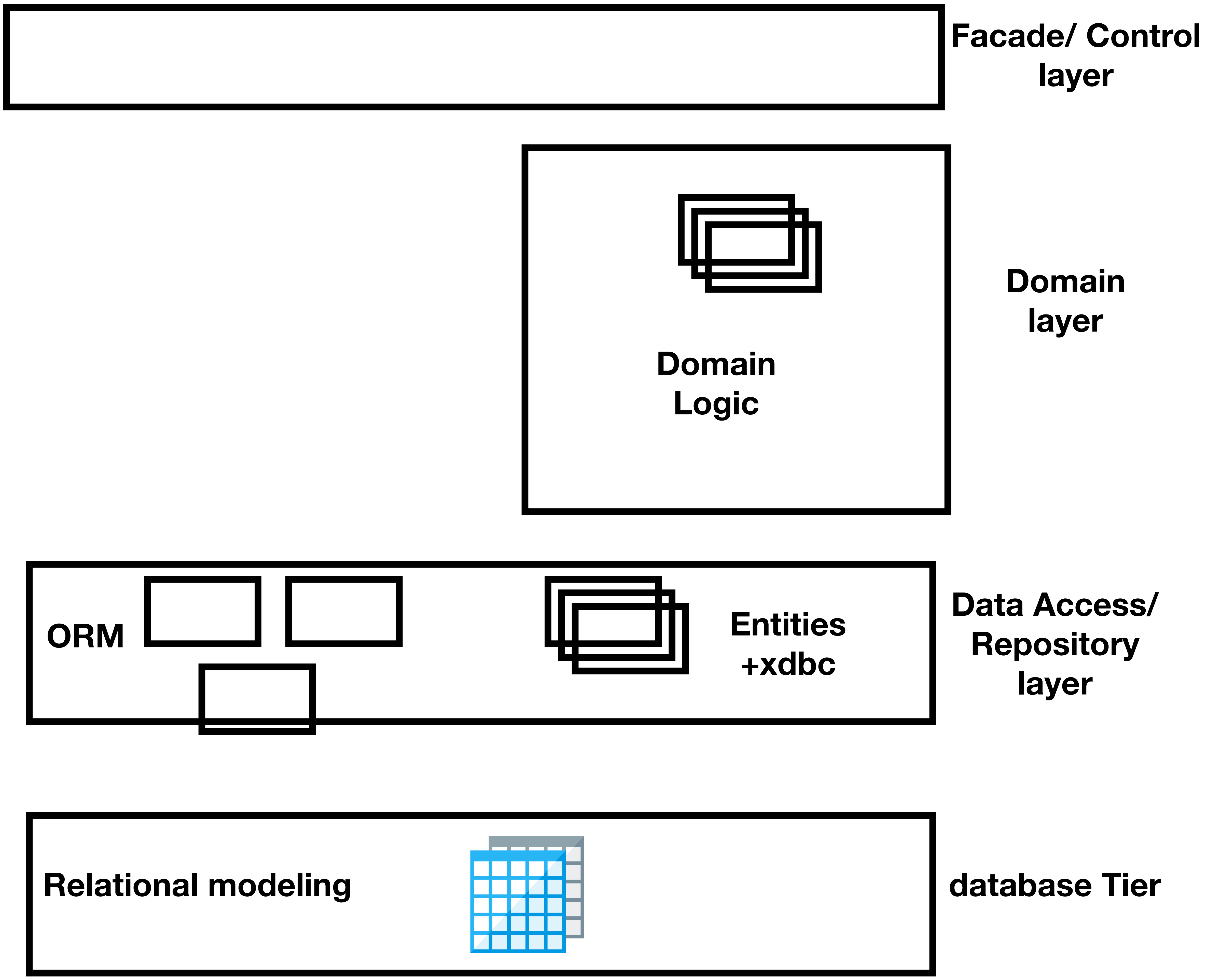
Broker

```
...
broker.trade();
....
```

Broker

MyBroker


New client



Facade/ Control layer

Domain layer

Domain Logic

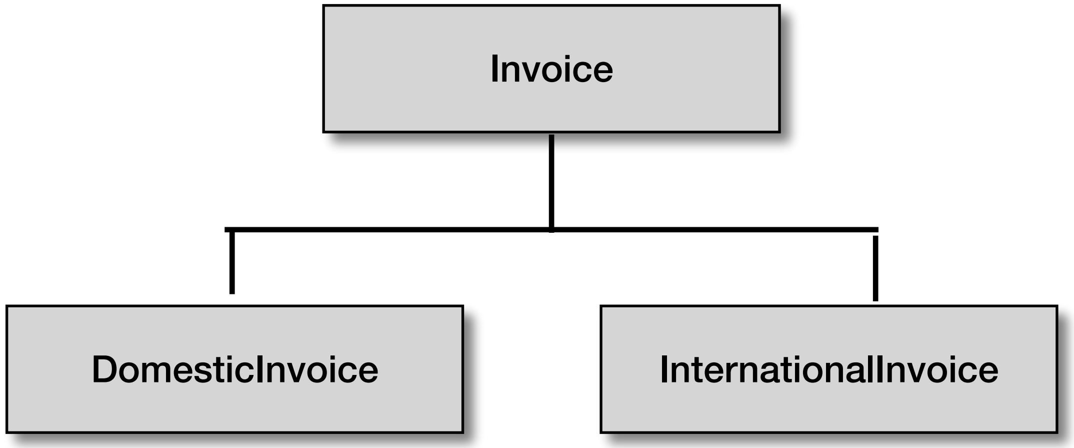
ORM

Entities +xdbc

Data Access/ Repository layer

Relational modeling

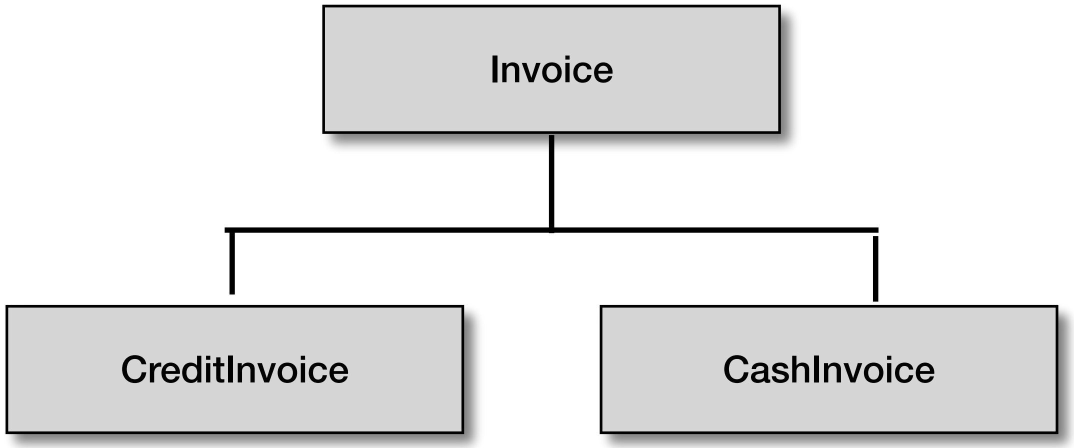
database Tier



Invoice

DomesticInvoice

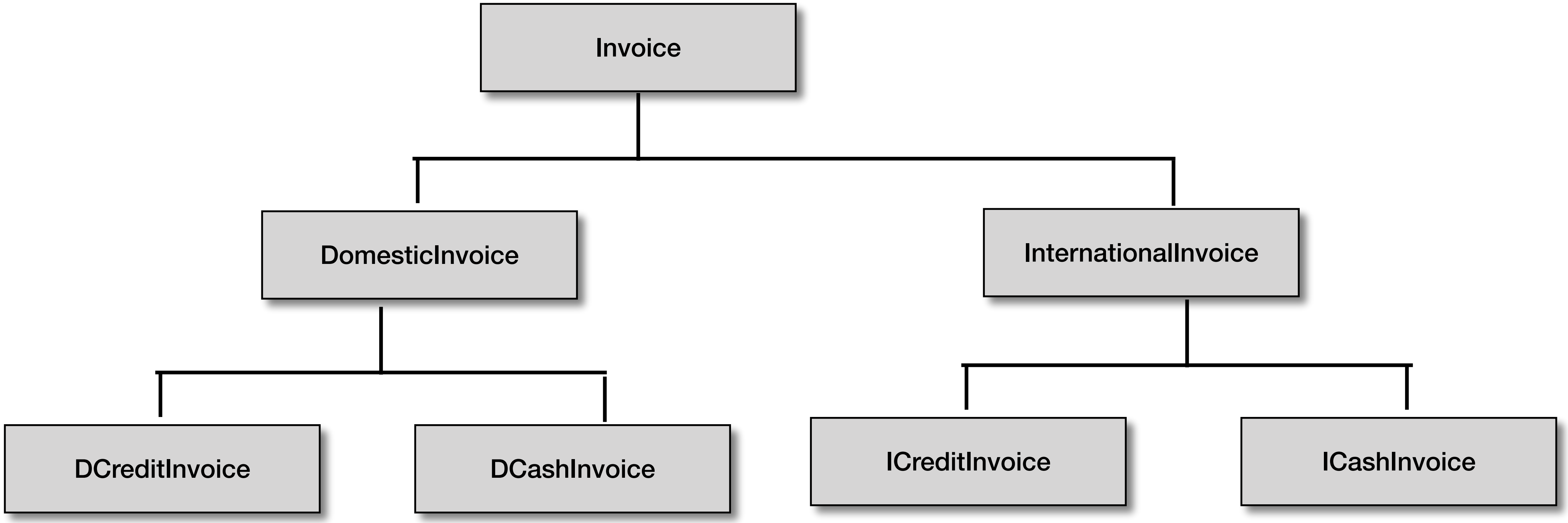
InternationalInvoice



Invoice

CreditInvoice

CashInvoice



Invoice

DomesticInvoice

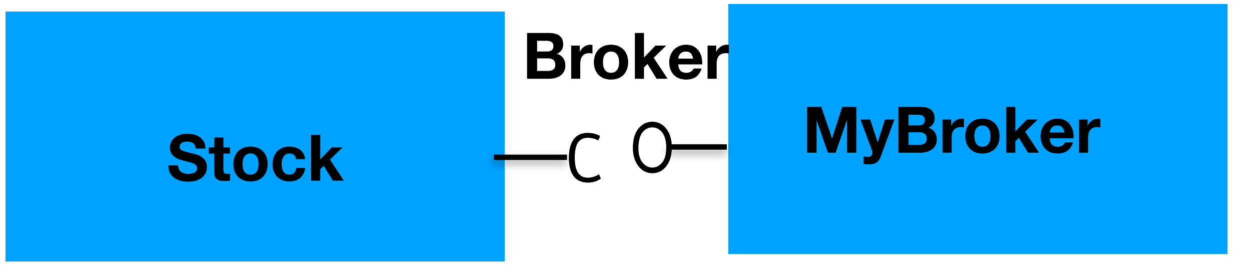
InternationalInvoice

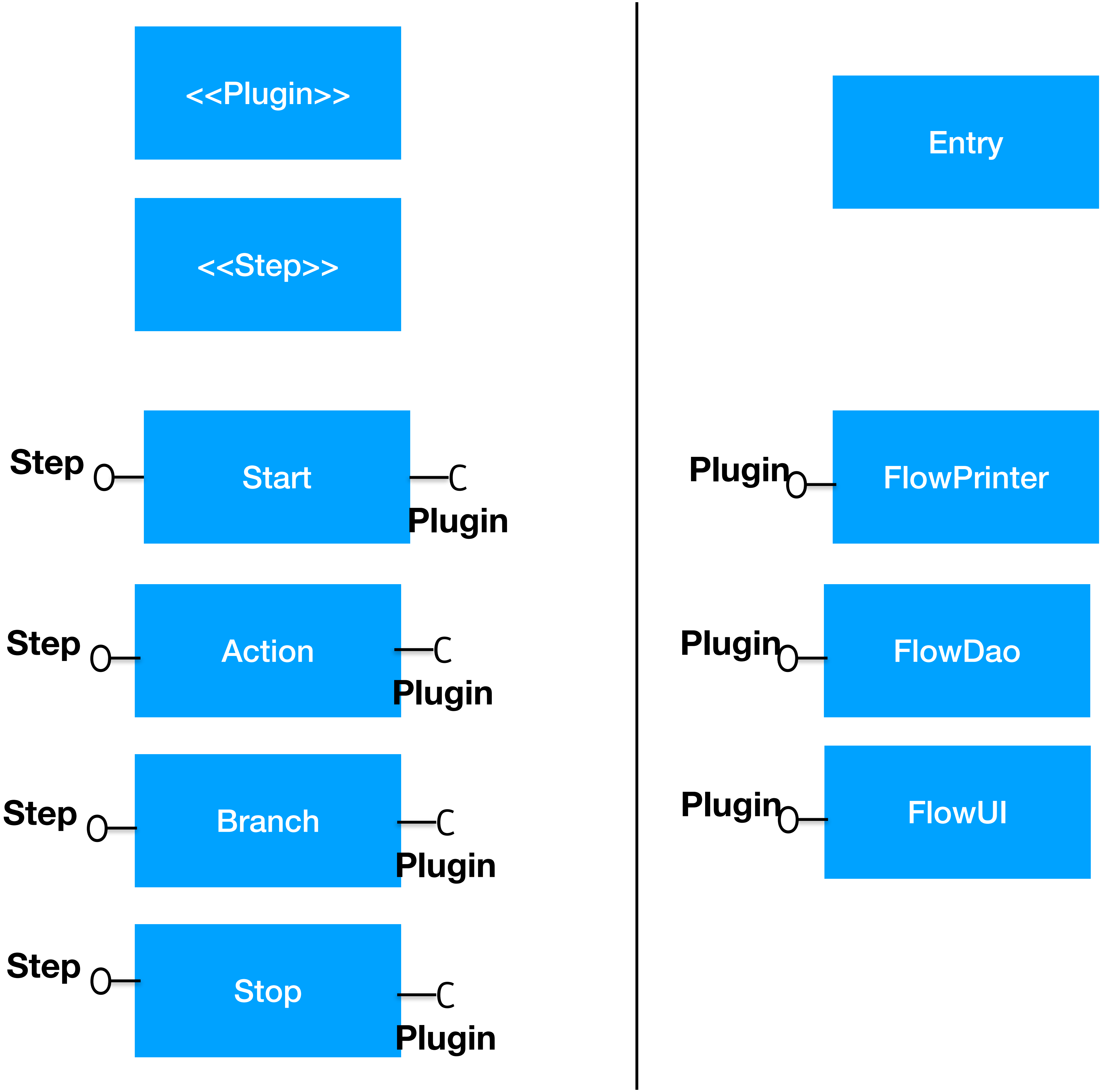
DCreditInvoice

DCashInvoice

ICreditInvoice

ICashInvoice


Stock
Broker
MyBroker



<<Plugin>>

<<Step>>

Step

Start

Plugin

Step

Action

Plugin

Step

Branch

Plugin

Step

Stop

Plugin

Entry

Plugin

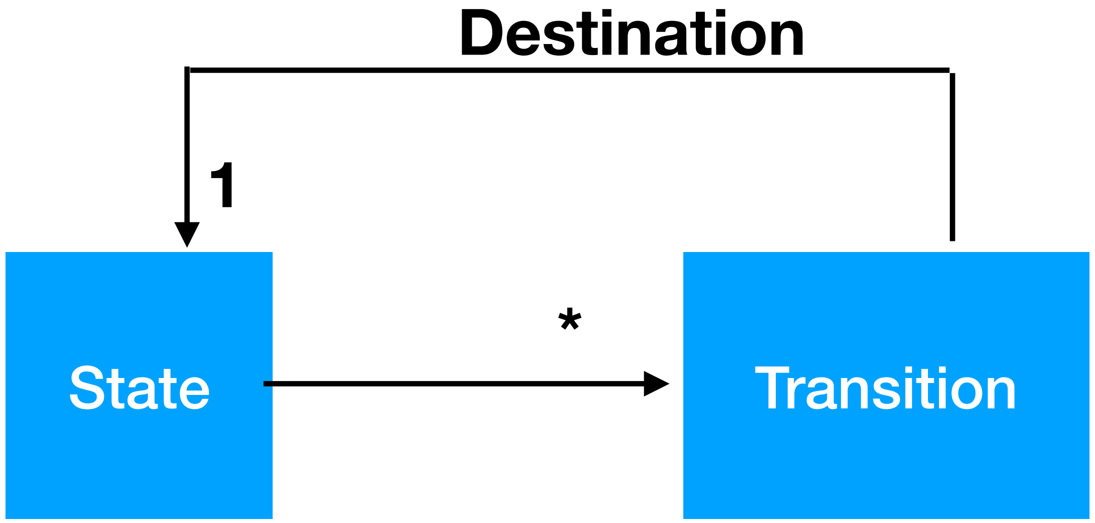
FlowPrinter

Plugin

FlowDao

Plugin

FlowUI



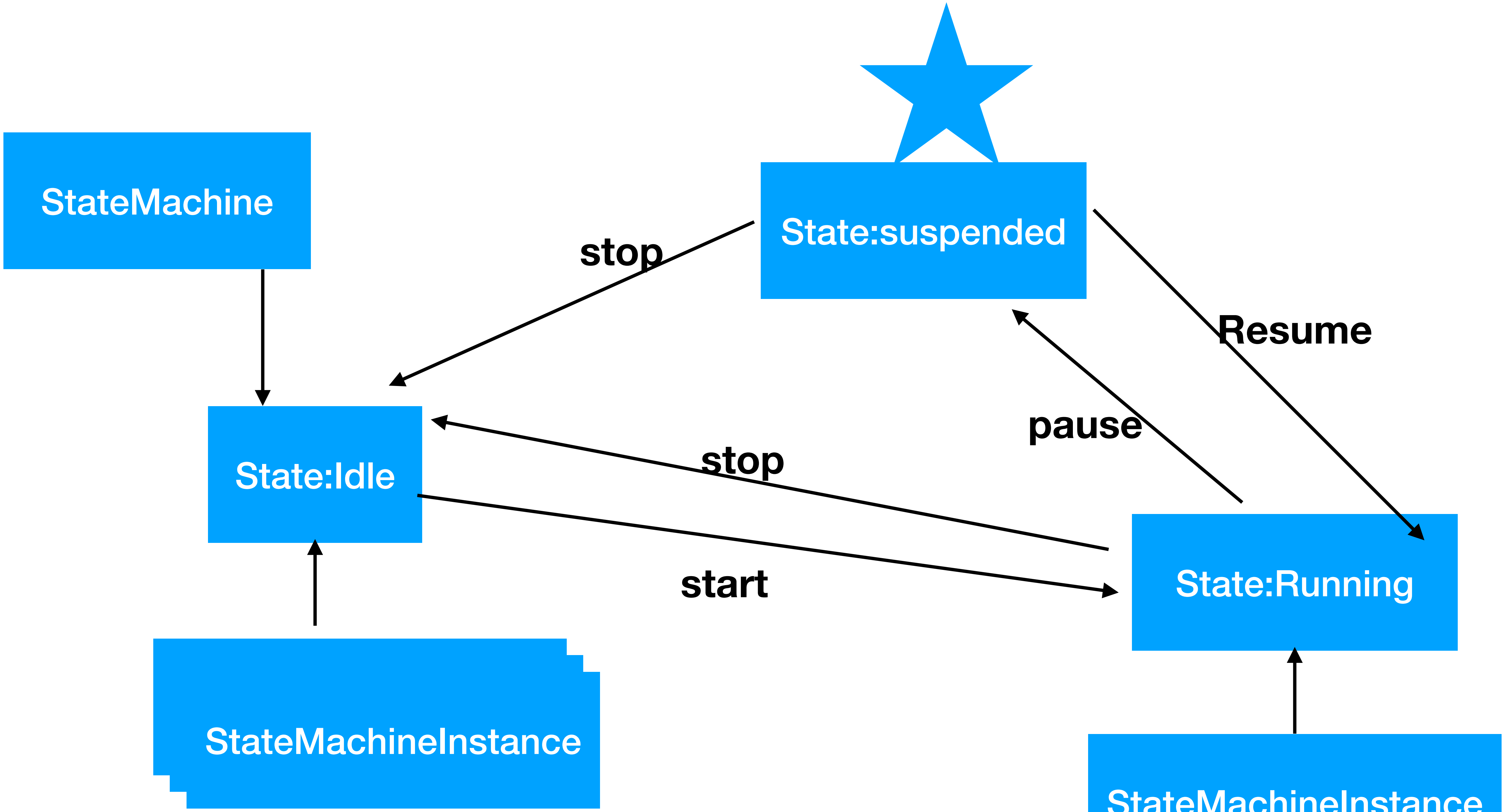
Destination

1

State

*

Transition



StateMachine

State:suspended

stop

Resume

pause

State:Idle

stop

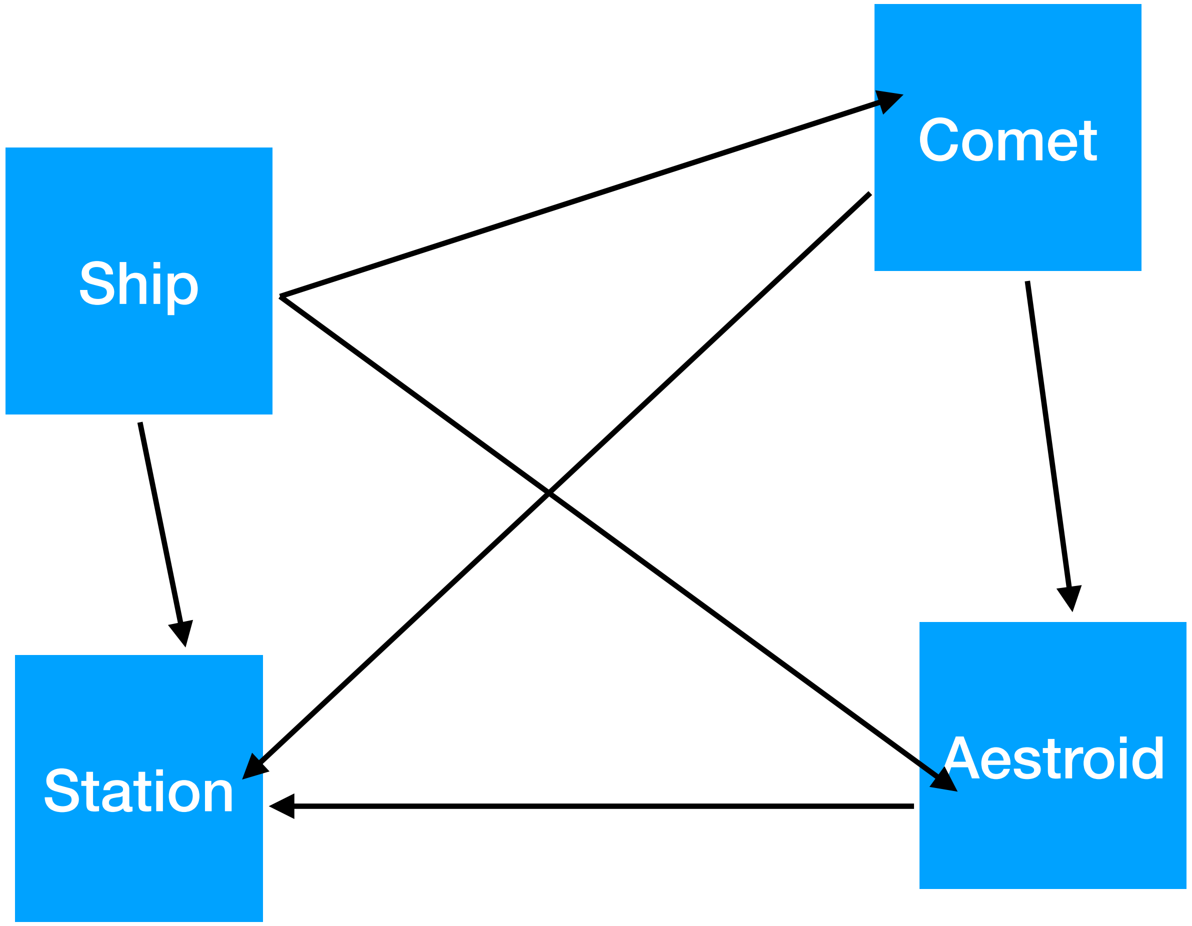
start

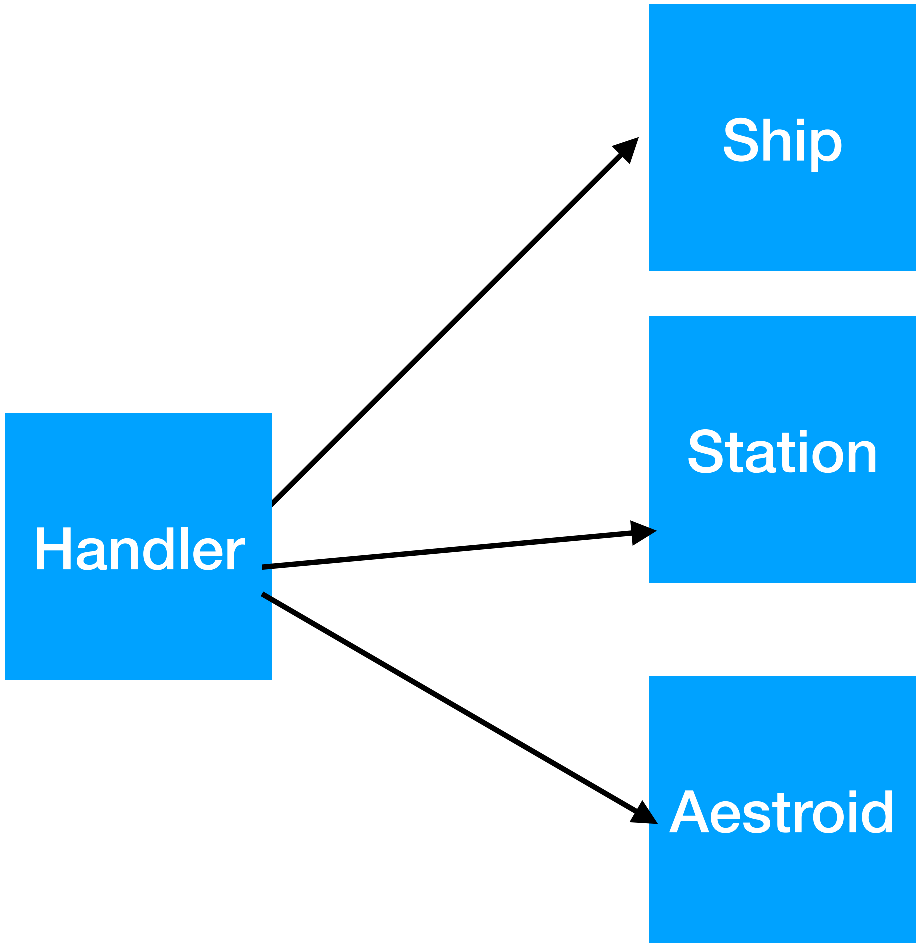
State:Running

StateMachineInstance

StateMachineInstance

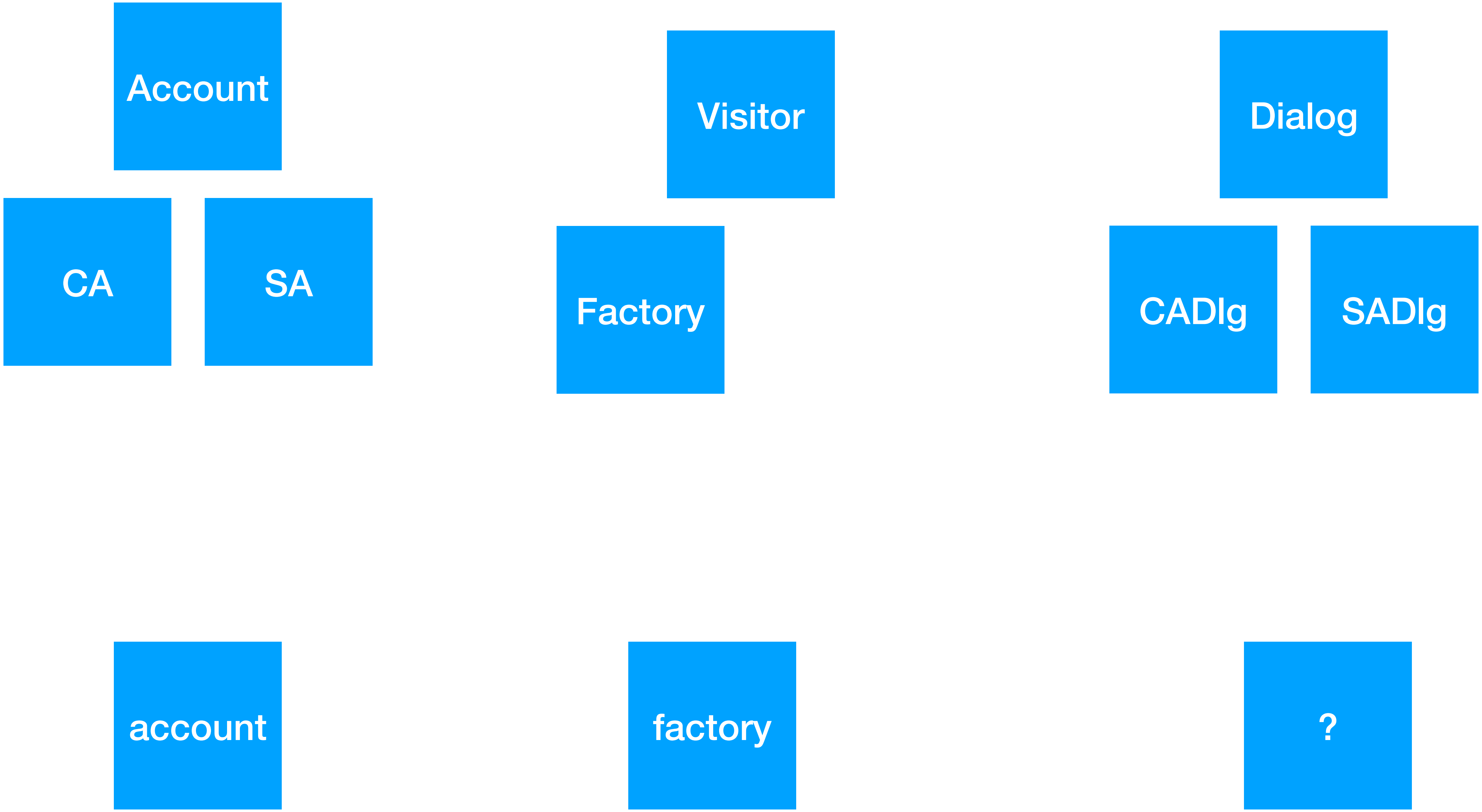
	5A (static)	3 (dynamic)
OCP		***
KISS	***	


Comet
Ship
Station
Aestroid


Ship
Station
Handler
Aestroid

Factory

- Creator method
- Factory method
- Class factory
- Abstract factory



Account

CA

SA

Visitor

Factory

Dialog

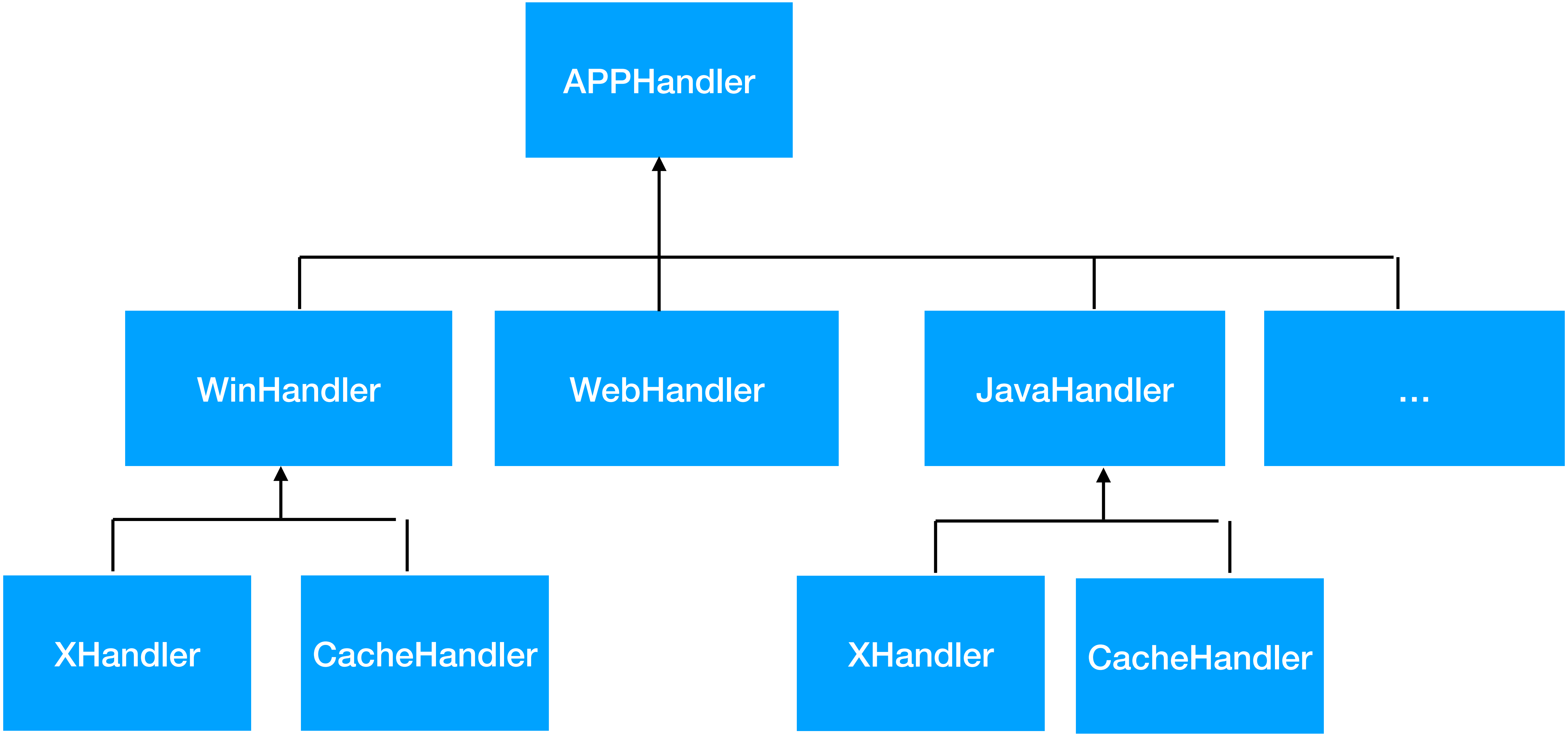
CADlg

SADlg

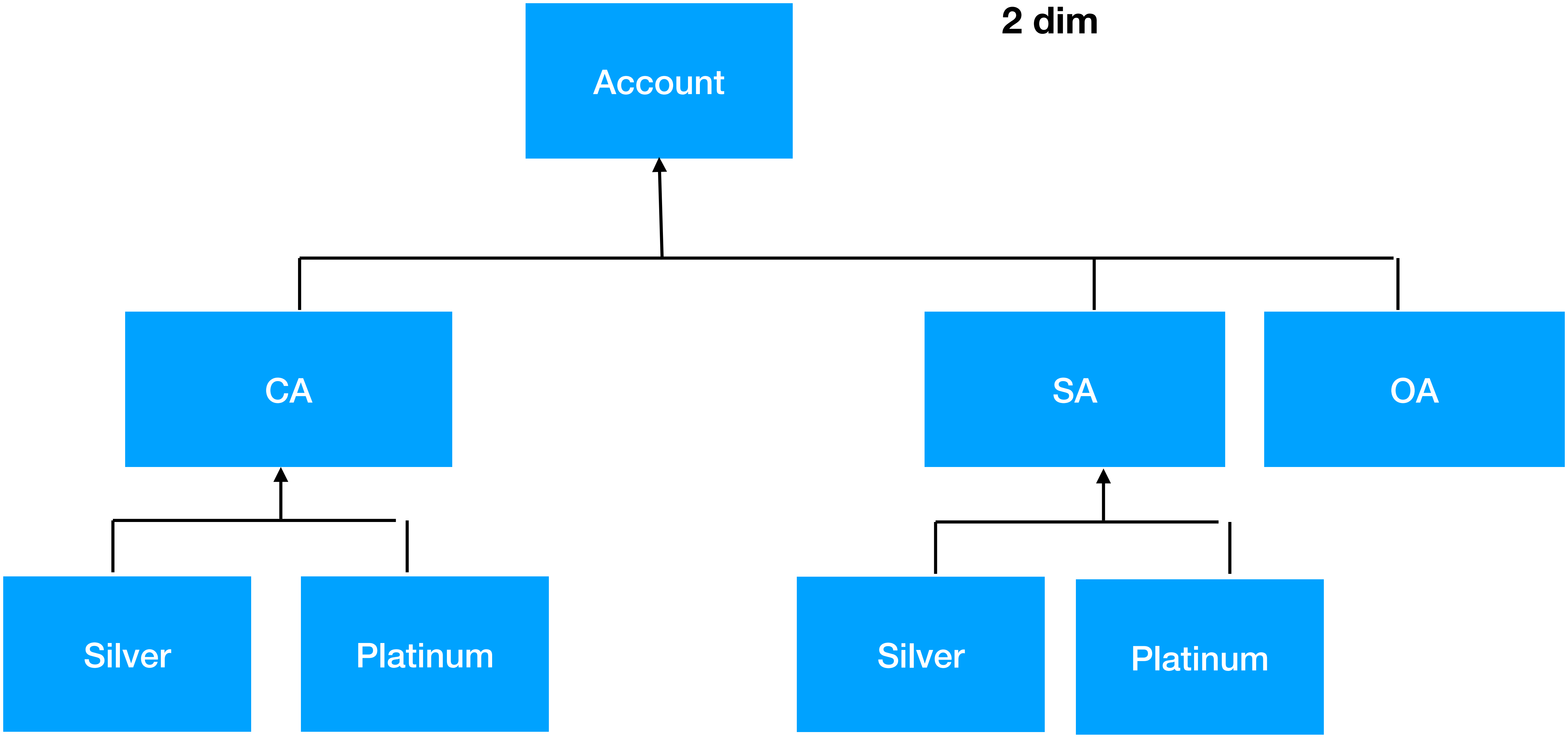
account

factory

?


APPHandler
WinHandler
WebHandler
JavaHandler
...
XHandler
CacheHandler
XHandler
CacheHandler

2 dim



Account

CA

SA

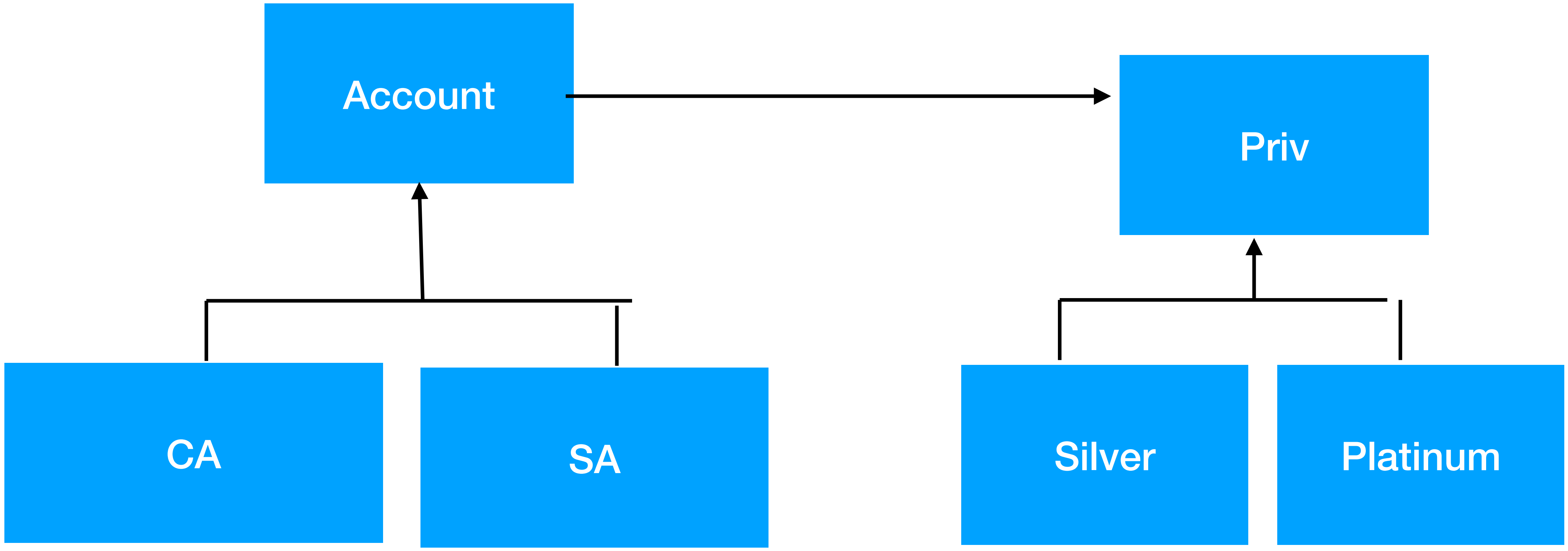
OA

Silver

Platinum

Silver

Platinum


Account
Priv
CA
SA
Silver
Platinum

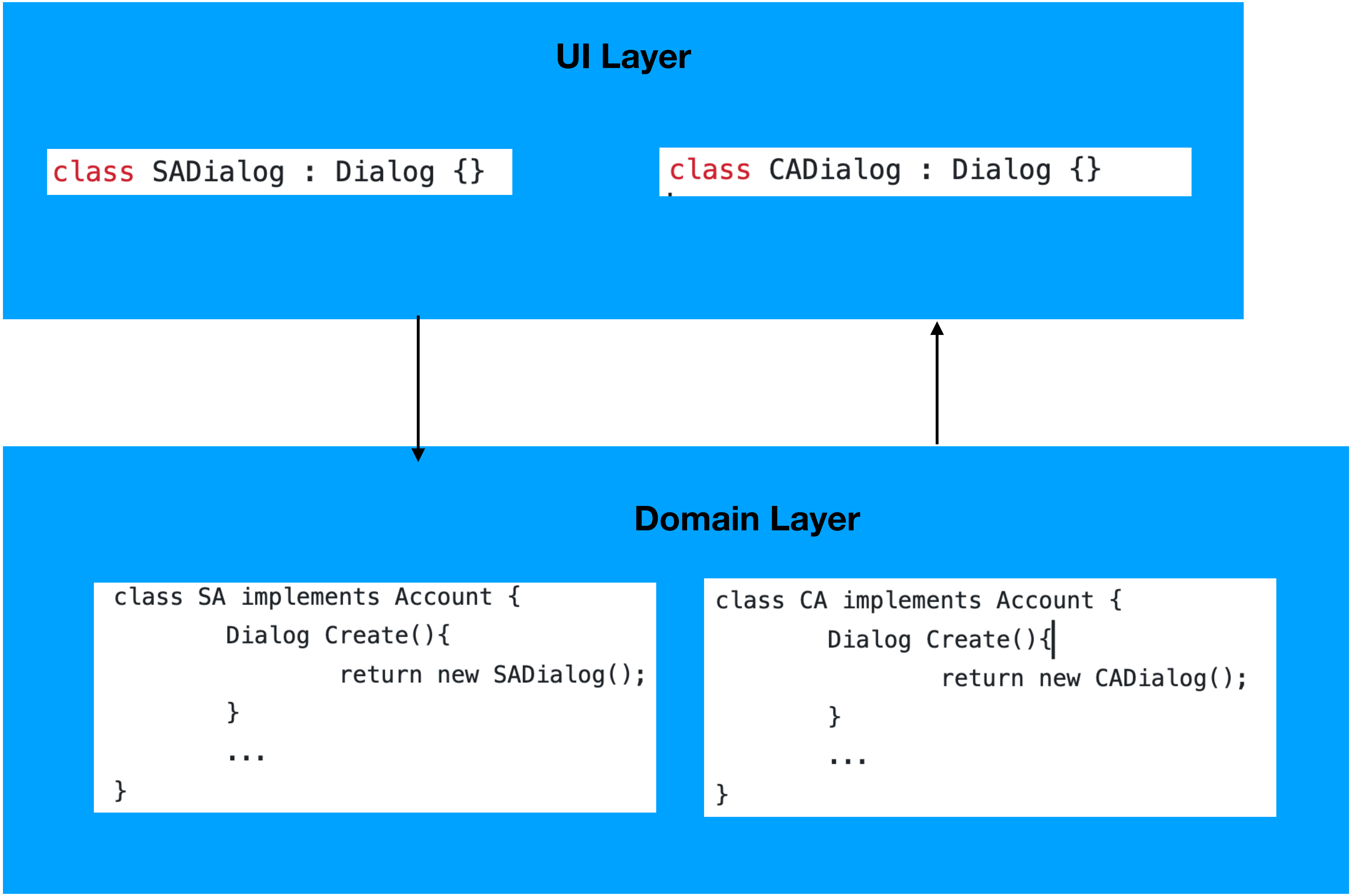
```
Account{}
class SA : Account{}
class CA : Account{}

class SilverSA : SA{}
class PlatinumSA : SA{}
class BronzeSA : SA{}

class SilverCA : CA{}
class PlatinumCA : CA{}
class BronzeCA : CA{}

class SilverOverdraft : Overdraft{}
class PlatinumOverdraft :Overdraft{}
class BronzeOverdraft : Overdraft{}
```

```
Account acc = new SilverSA();

acc.withdraw();
```

```
Priv{}
class Silver : Priv{}
class Platinum : Priv{}
class Bronze : Priv{}


Account{
  priv;
}
class SA : Account{}
class CA : Account{}
class OD : Account{}
```

```
Account acc = New SA(New Silver());

acc.withdraw();
```

Account type “ Saving/ Current
Priv Type “ Silver / Platinum



UI Layer

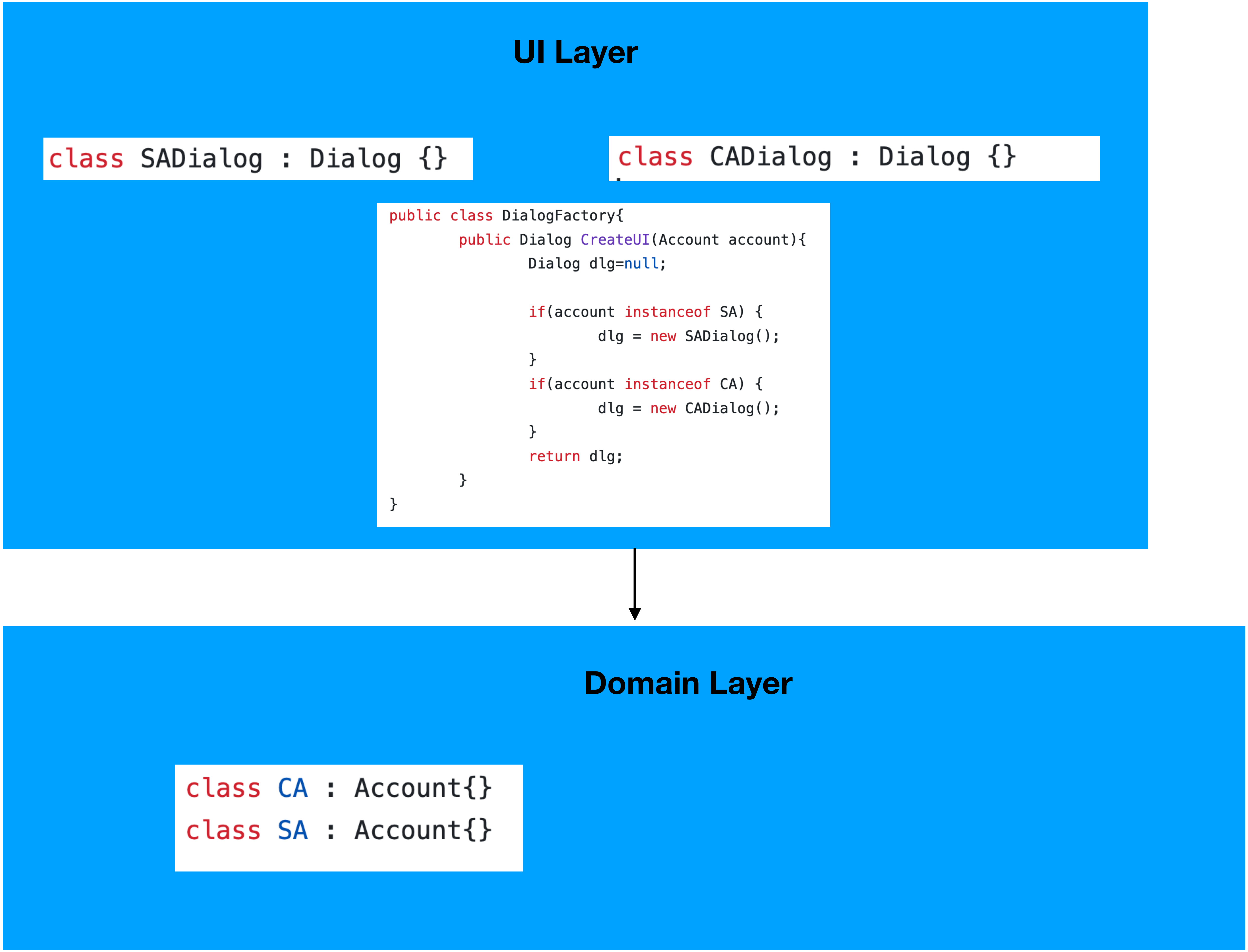
```
class SADialog : Dialog {}
```

```
class CADialog : Dialog {}
```

Domain Layer

```
class SA implements Account {
        Dialog Create(){
                return new SADialog();
        }
        ...
}
```

```
class CA implements Account {
        Dialog Create(){
                return new CADialog();
        }
        ...
}
```



UI Layer

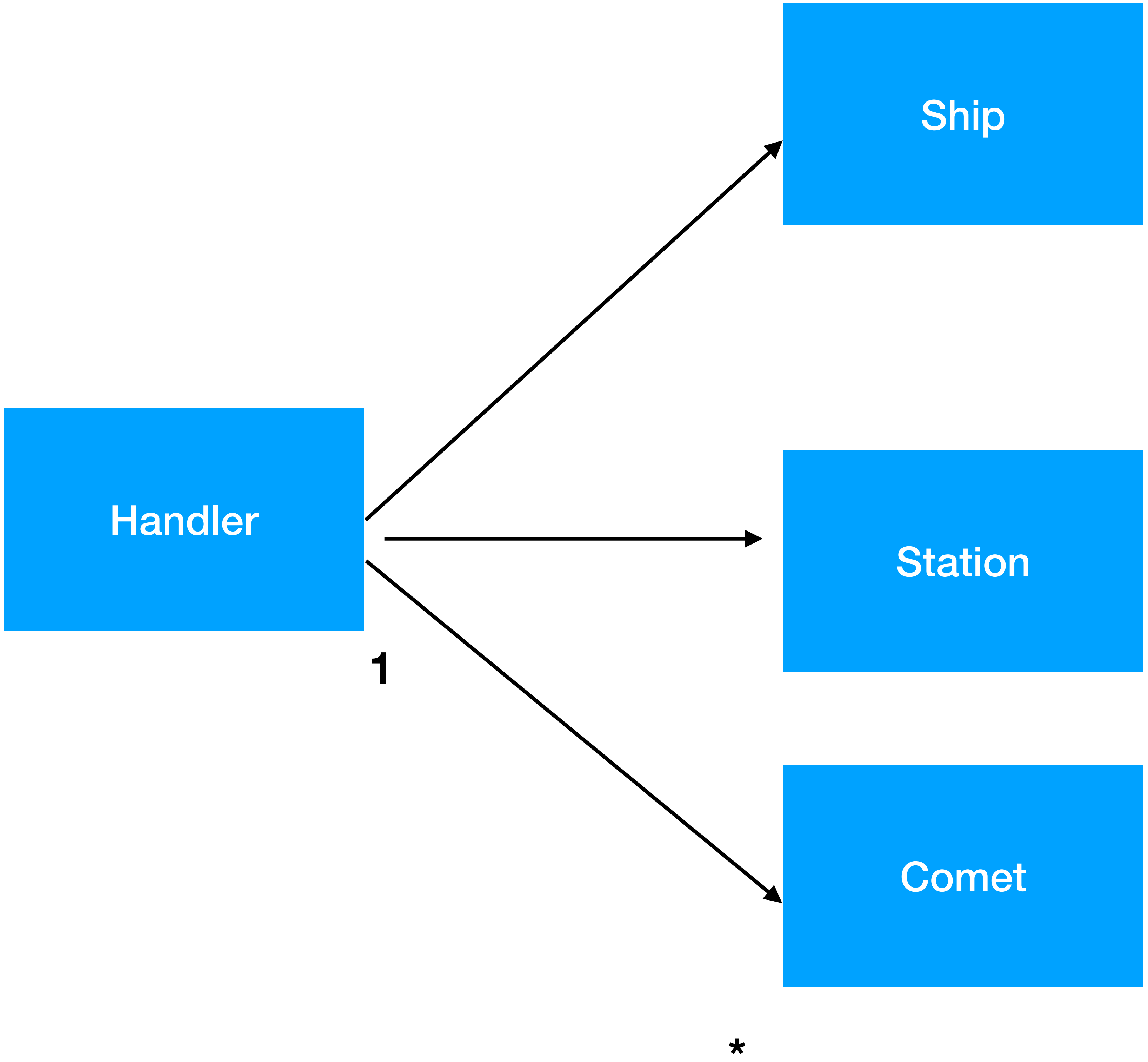
```
class SADialog : Dialog {}
```

```
class CADialog : Dialog {}
```

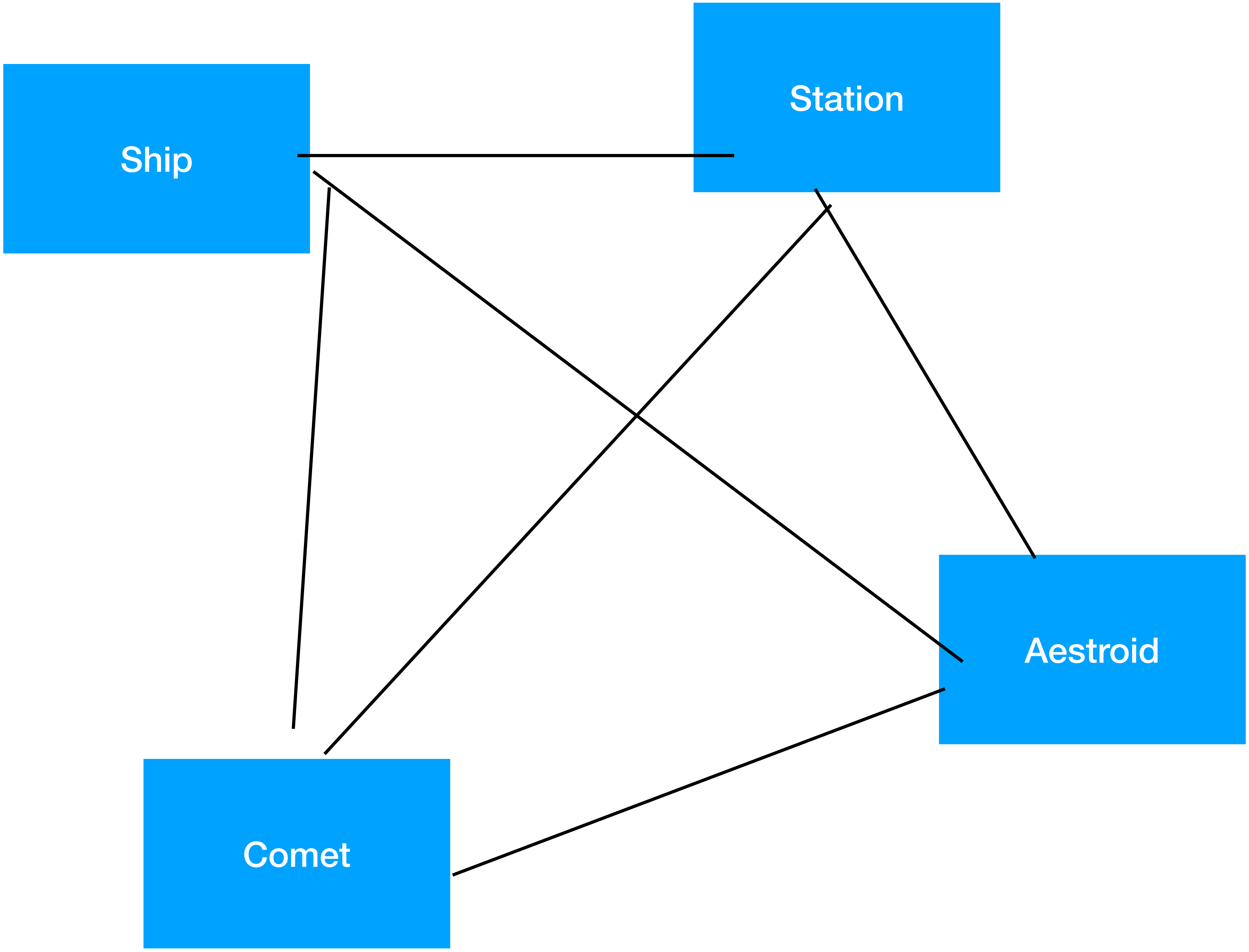
```
public class DialogFactory{
        public Dialog CreateUI(Account account){
                Dialog dlg=null;

                if(account instanceof SA) {
                        dlg = new SADialog();
                }
                if(account instanceof CA) {
                        dlg = new CADialog();
                }
                return dlg;
        }
}
```

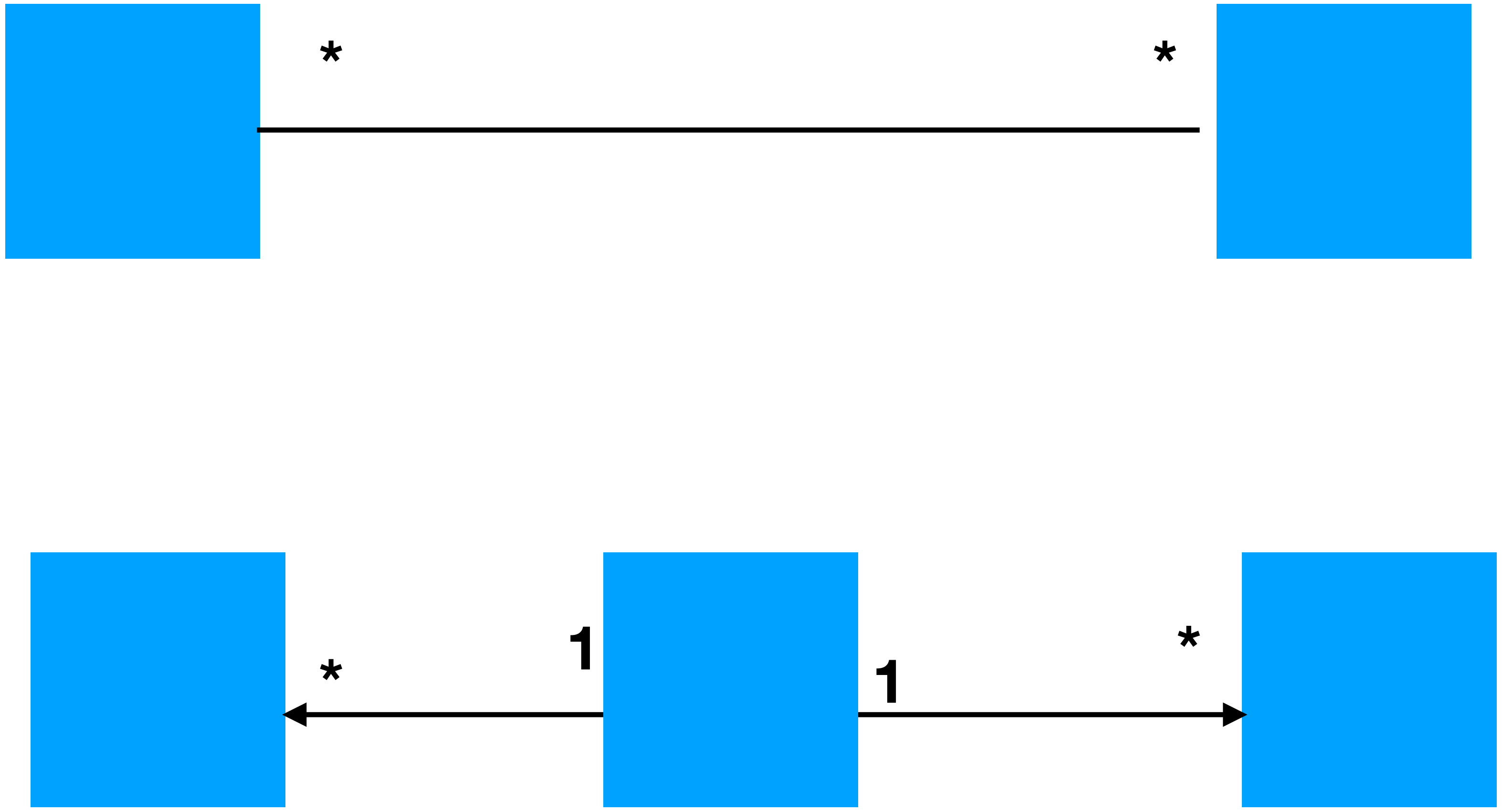
Domain Layer

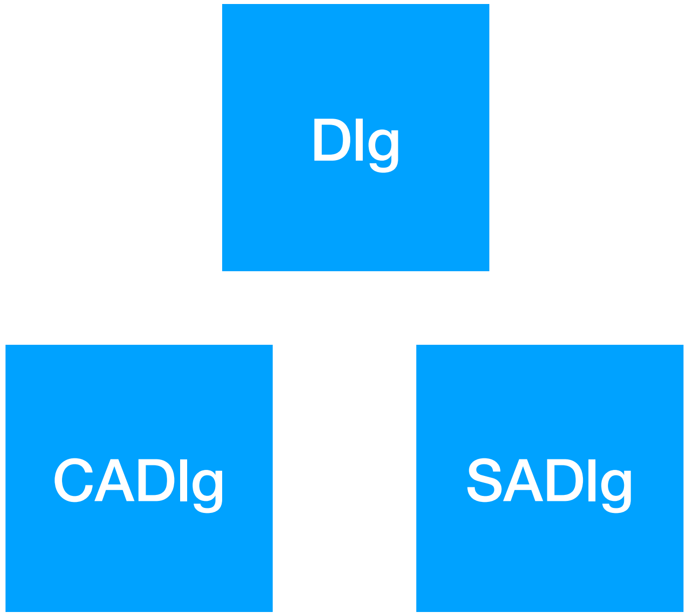
```
class CA : Account{}
class SA : Account{}
```



Ship
Handler
Station
1
Comet
*


Ship
Station
Aestroid
Comet

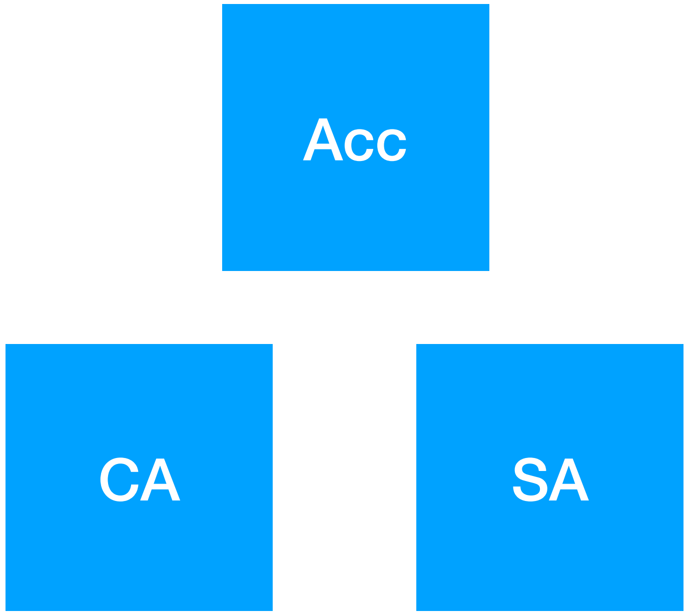

*
*
*
1
1
*



Dlg

CADlg

SADlg



Acc

CA

SA

	0 (Idle)	1 (Running)	2(Suspended)
Start	Logic	Error	Error
Stop	Error	Logic	Logic
Pause	Error	Logic	Error
Resume	Error	Error	Logic

Review of code

class is having multiple resp like managing game, game logic. We can have seperate classes
Yes. There should be more classes;
Collecting user input
Run game logic
Store board state
Print board state
have constants
Remove multiple nested if else
cannot be extended if we want to play 4X4
code duplications for checking wining lines and can improve redability
Duplicate code in printBoard

Good

- SRP (***)
- Low coupling (***)
- Unit testability
- LSP
- ISP
- Upcasting/abstraction
- DRY
-

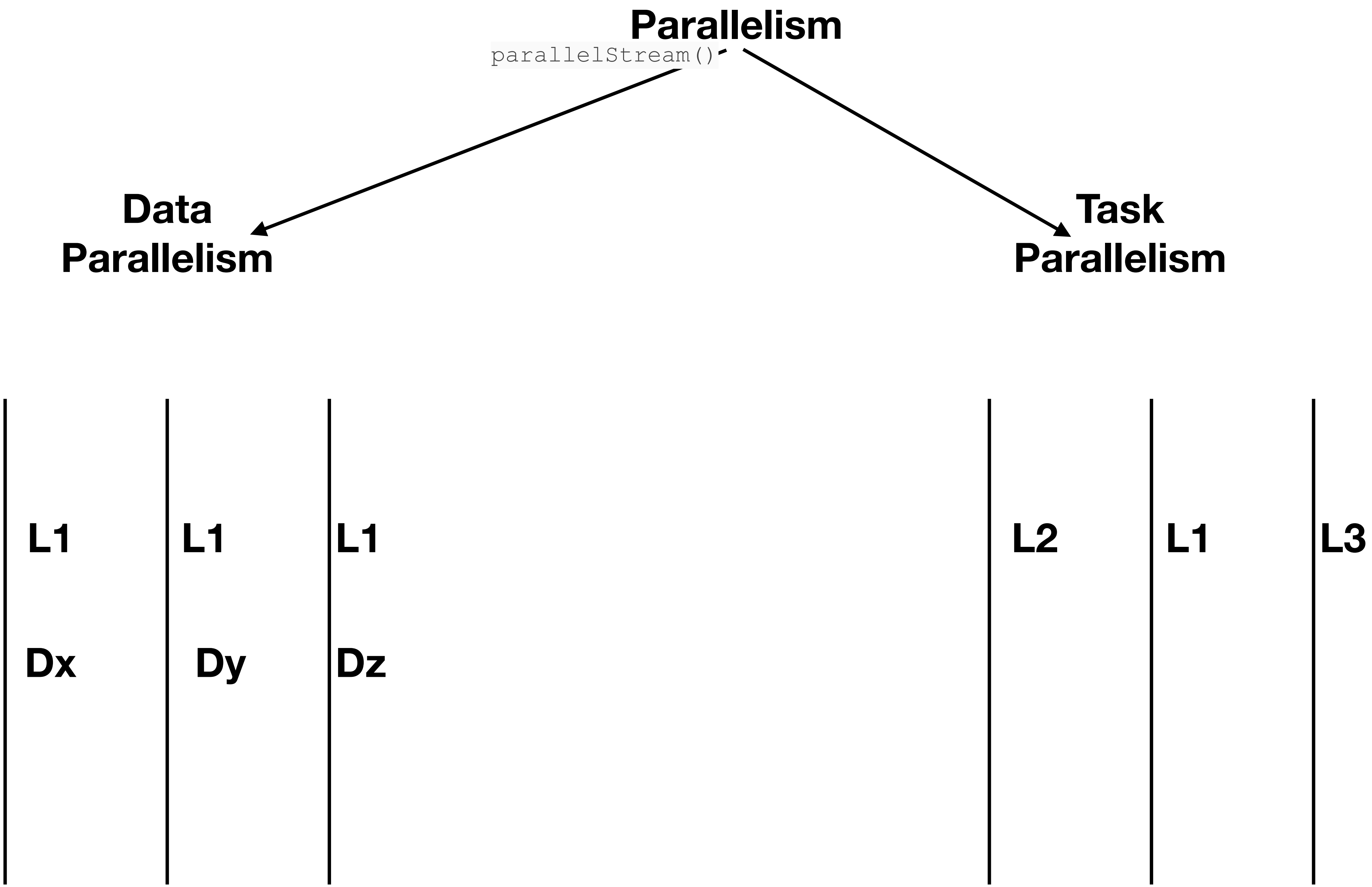
- Prefer composition over inheritance
- Boundary control entity (*)
- YAGNI
- KISS
- Program to an Interface
- DDD
    - Aggregates

Bad

- Type check
- Flag check
- dont use overloading on Family of types
- Downcasting
- Arrow code
- Magic numbers/strings
- Tight coupling across units
- Cyclic coupling
- * to * coupling

- Duplicate code
- Dead code
- Commented code
- bool/ null/ int for error handling
- Static methods
- Singleton GOF pattern
- Functional interface
- God class
- Avoid Inheritance (extends)

Good (concurrency)



Parallelism

parallelStream()

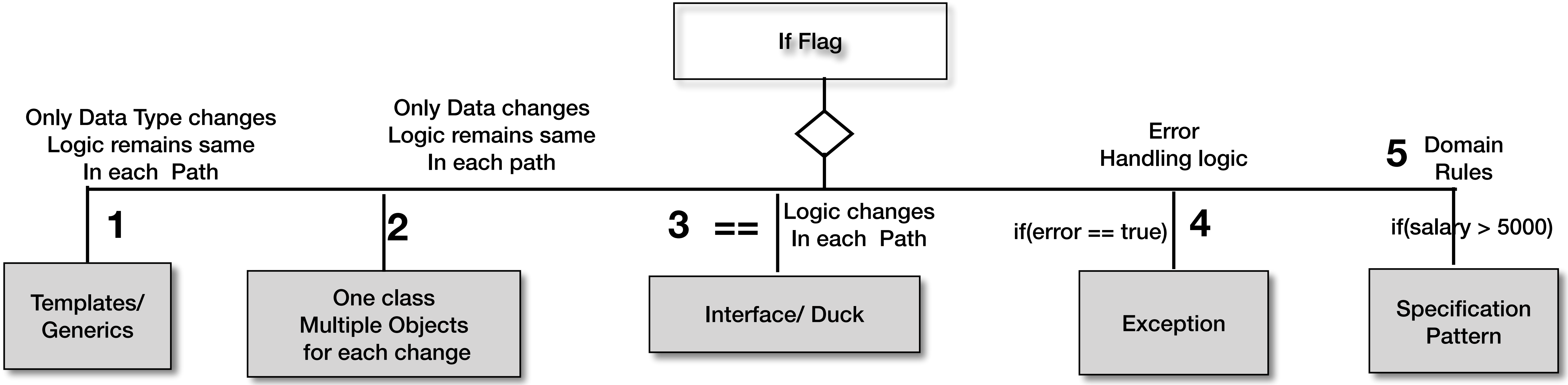
Data Parallelism

Task Parallelism

L1 L1 L1

Dx Dy Dz

L2 L1 L3



If Flag

Only Data Type changes
Logic remains same
In each Path

1

Templates/
Generics

Only Data changes
Logic remains same
In each path

2

One class
Multiple Objects
for each change

3 ==

Logic changes
In each Path

Interface/ Duck

Error
Handling logic

if(error == true)

4

Exception

5 Domain
Rules

if(salary > 5000)

Specification
Pattern

```
Class Emp{
Emp ref;

}
```

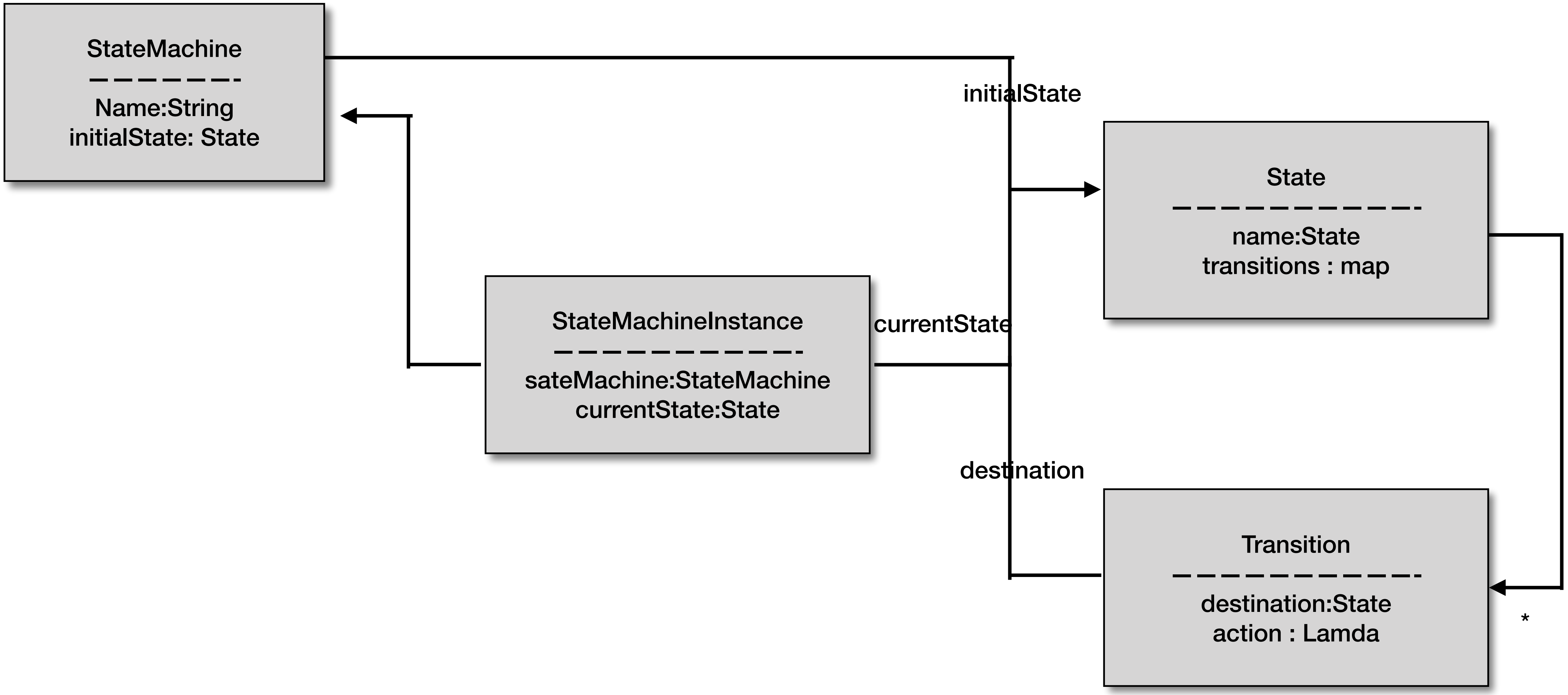
```
Class Emp{
List<Emp> emps;

}
```

```
Class Emp{
Emp e1;
Emp e2;

}
```

```
class State{


}
```

```
State idle = new State();
State running = new State();
State suspended = new State();
```

```
Class Idle implements State{}
Class Running implements State{}
Class Suspended implements State{}
```



StateMachine

Name:String
initialState: State

StateMachineInstance

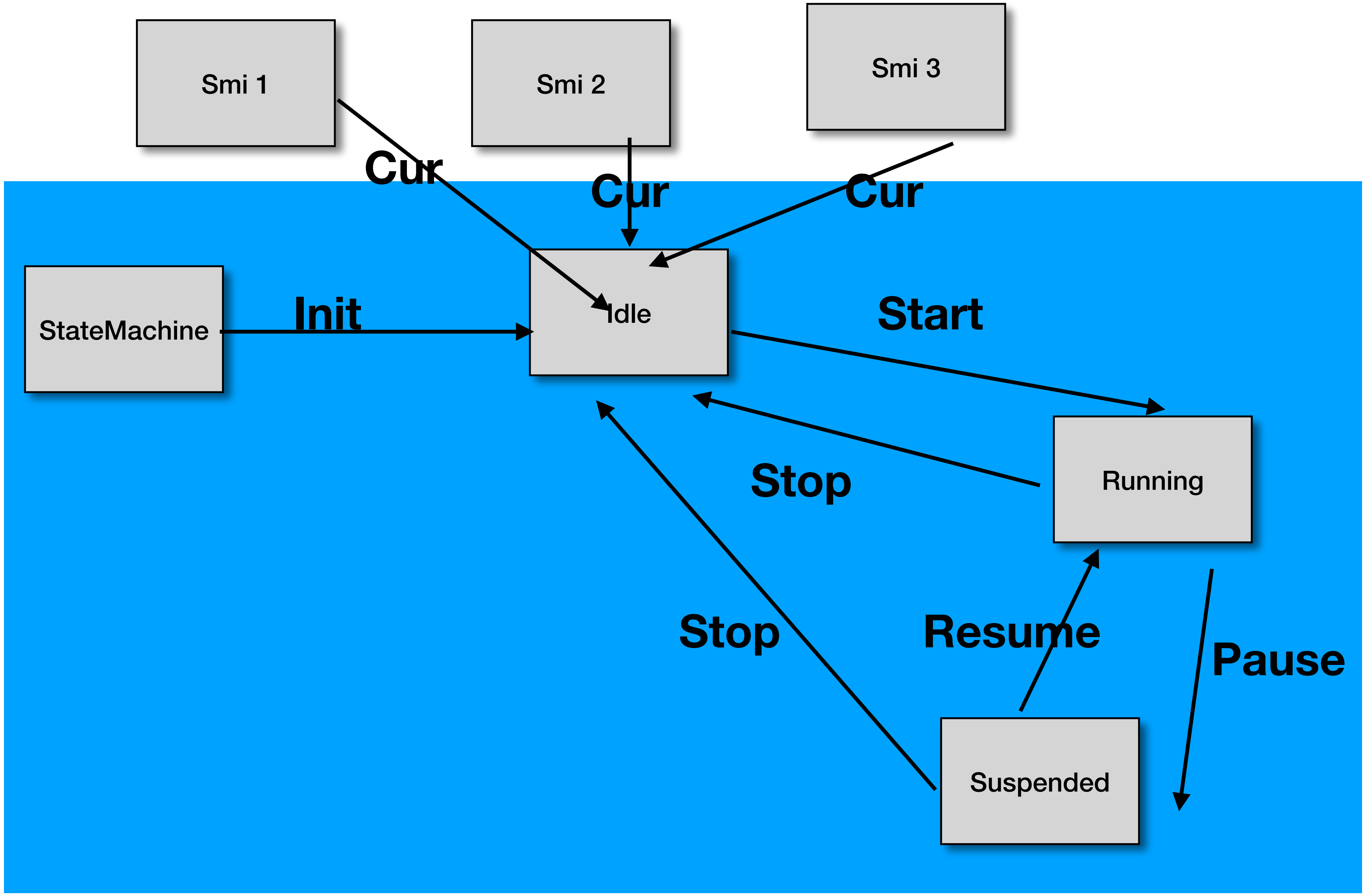
sateMachine:StateMachine
currentState:State

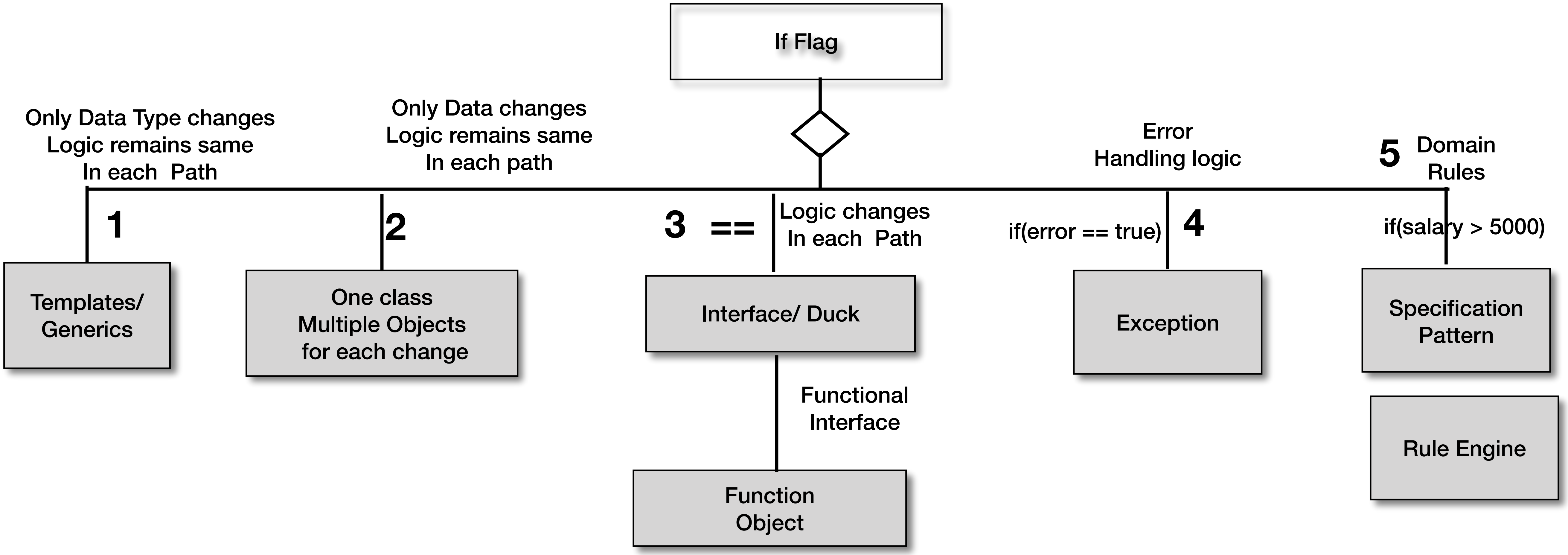
State

name:State
transitions : map

Transition

destination:State
action : Lamda


Smi 1
Smi 2
Smi 3
Cur
Cur
Cur
StateMachine
Init
Idle
Start
Stop
Running
Stop
Resume
Pause
Suspended



If Flag

1

Only Data Type changes
Logic remains same
In each Path

Templates/
Generics

2

Only Data changes
Logic remains same
In each path

One class
Multiple Objects
for each change

3 ==

Logic changes
In each Path

Interface/ Duck

Functional
Interface

Function
Object

4

Error
Handling logic

if(error == true)

Exception

5

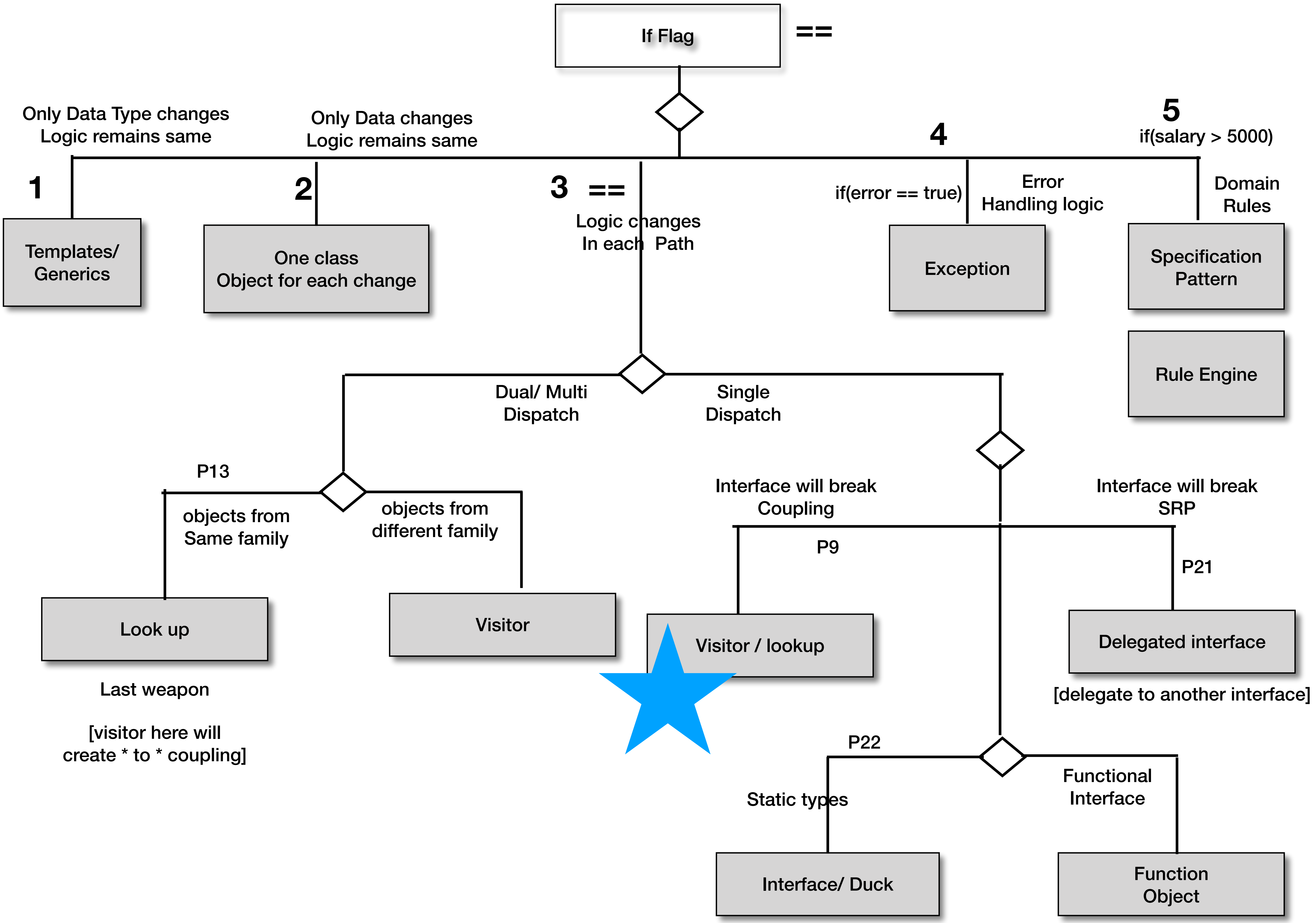
Domain
Rules

if(salary > 5000)

Specification
Pattern

Rule Engine

Lookup



If Flag ==

Only Data Type changes
Logic remains same

1

Templates/
Generics

Only Data changes
Logic remains same

2

One class
Object for each change

3 ==

Logic changes
In each Path

4

if(error == true)

Error
Handling logic

Exception

5
if(salary > 5000)

Domain
Rules

Specification
Pattern

Rule Engine

Dual/ Multi
Dispatch

P13

objects from
Same family

Look up

Last weapon

[visitor here will
create * to * coupling]

objects from
different family

Visitor

Single
Dispatch

Interface will break
Coupling

P9

Visitor / lookup

Interface will break
SRP

P21

Delegated interface

[delegate to another interface]

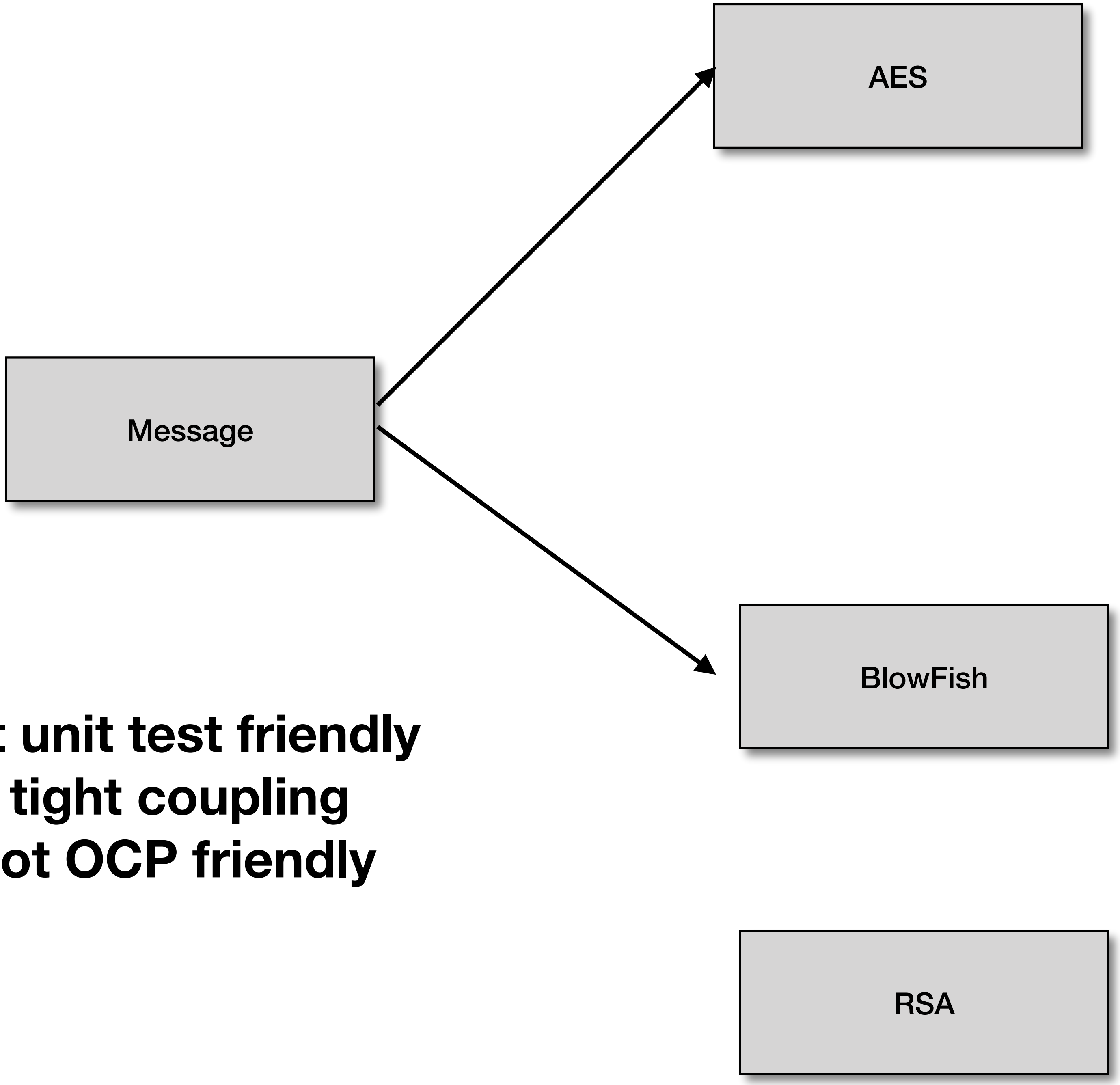
P22

Static types

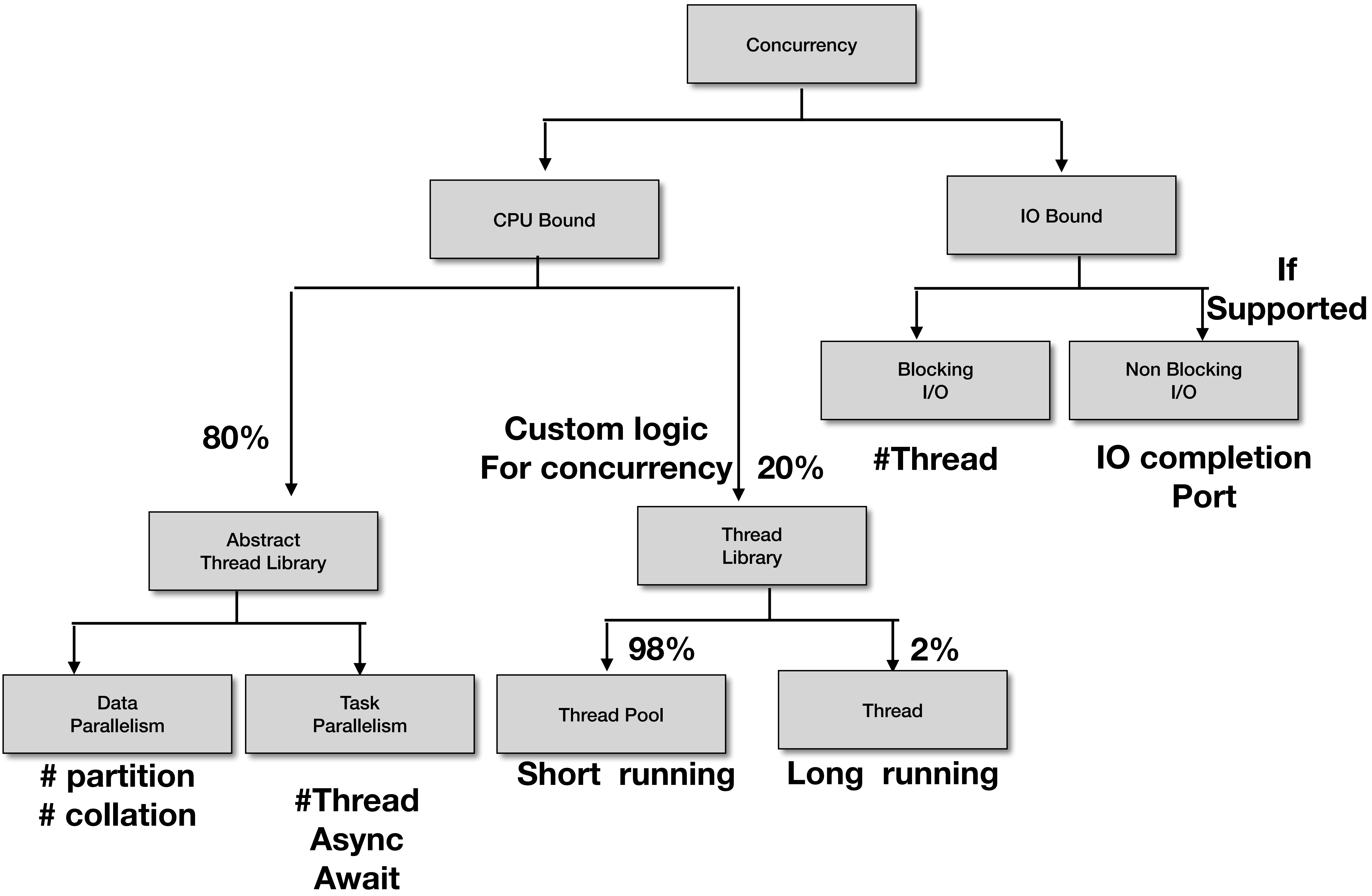
Interface/ Duck

Functional
Interface

Function
Object



# not unit test friendly
# tight coupling
# not OCP friendly



Concurrency

CPU Bound

IO Bound

If Supported

Blocking I/O

Non Blocking I/O

80%

Custom logic For concurrency

20%

#Thread

IO completion Port

Abstract Thread Library

Thread Library

98%

2%

Data Parallelism

Task Parallelism

Thread Pool

Thread

# partition
# collation

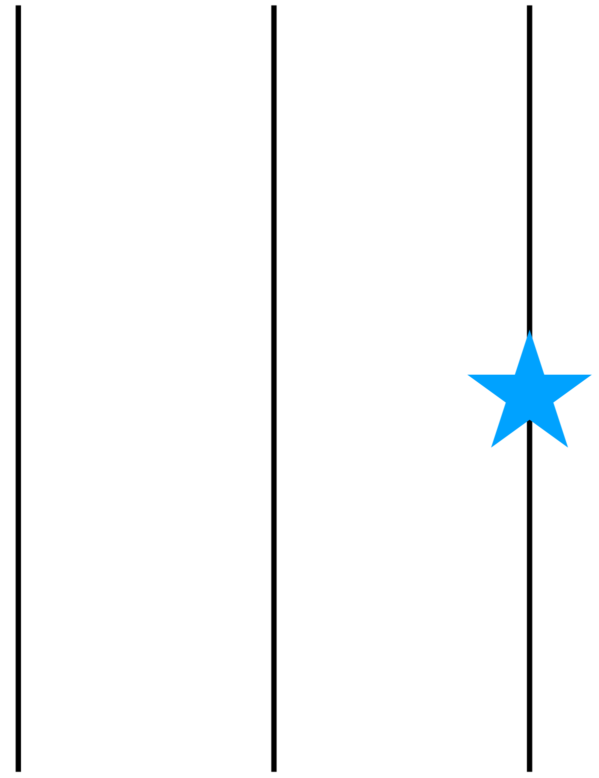
#Thread
Async
Await

Short running

Long running

Bad (concurrency)

- Abort
- Suspend
- Sleep
- SetThreadPriority
- Static / shared data (Global state)



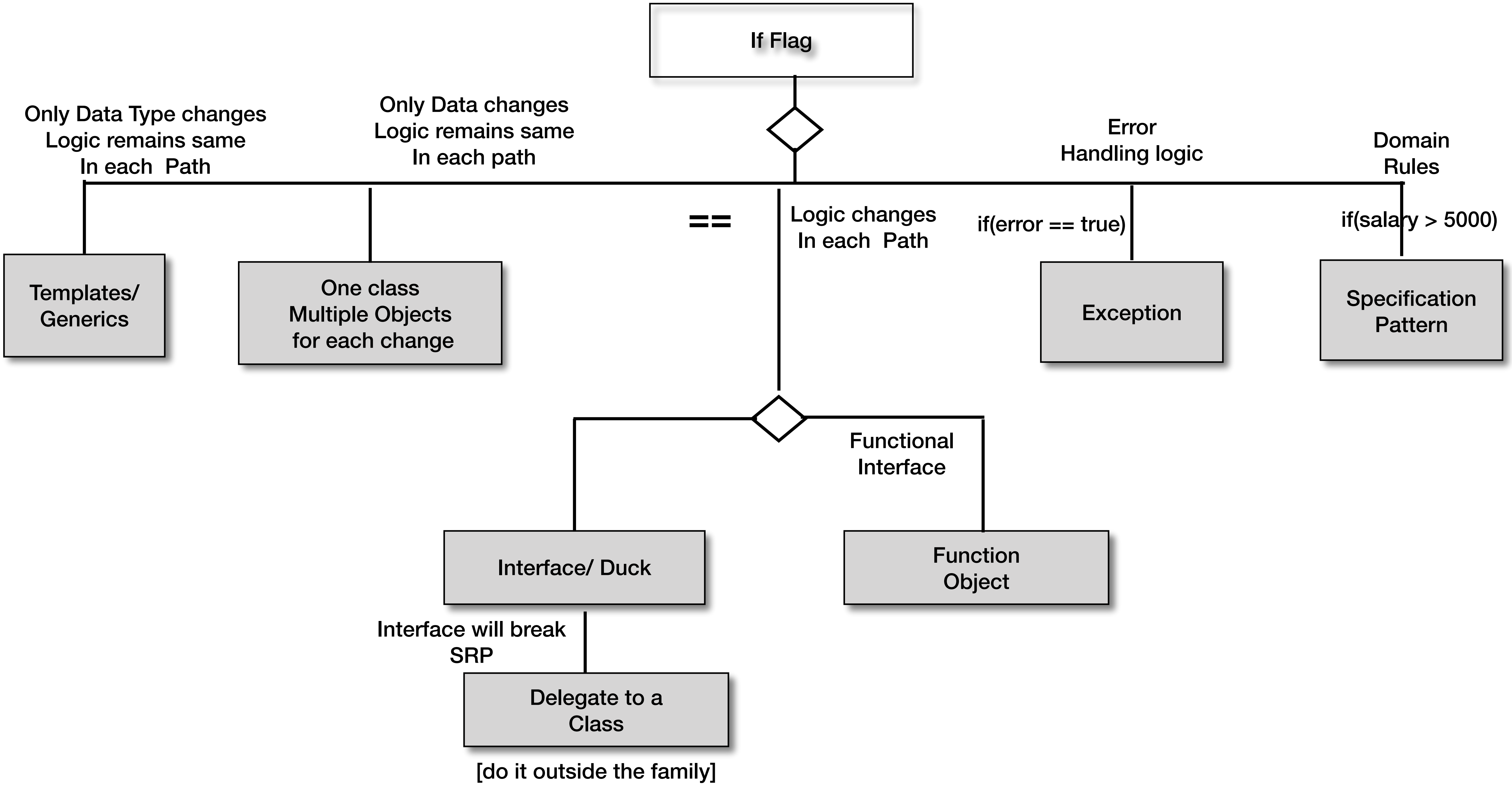
SOC

- Things which do not change together should not be kept together
- Logic and error handling
- Domain logic and domain rules
- Boundary logic and domain logic
-

Size **

- Fun size
  - Max : fit screen
  - Avg : < 10 lines

- Class size
  - Max fun : 12
  - Avg fun: 4



If Flag

Only Data Type changes
Logic remains same
In each Path

Templates/
Generics

Only Data changes
Logic remains same
In each path

One class
Multiple Objects
for each change

==

Logic changes
In each Path

Interface/ Duck

Interface will break
SRP

Delegate to a
Class

[do it outside the family]

Functional
Interface

Function
Object

Error
Handling logic

if(error == true)

Exception

Domain
Rules

if(salary > 5000)

Specification
Pattern

	10 fun 100 lines each	100 fun 10 lines each
Naming fun		***
Unit test		***
Refactoring		***
Understand Flow	?	With correct abstraction ***

```
If
{
Type changes
}
If
{
Type changes
}
If
{
Type changes
}
```

```
If
{
Value changes
}
If
{
Value changes
}
If
{
Value changes
}
```

Architecture
vs
Design

- Performance
- Scalability
- Reliability
- Availability
- Maintainability
- Security
- Robustness
- Portability
- Resilience
-

- Concurrency
- Cache
- Lazy loading
- Virtualization
- Polling
-

Architecture [Design]
vs
[Code] Design

Upcast
vs
Downcast

```
Bird bird  = new parrot / penguin;
do(bird);

do(Bird bird)
{
      bird.flab();
      ……
}
```

```
Bird bird  = ( Bird) parrot;
```

```
Bird bird  = new parrot / penguin;
…
…
…
..
…
If type(bird) == type(Parrot)
   Parrot parrot  = ( Parrot) bird;
   parrot….
```

```
Parrot parrot  = ( Parrot) bird;
```

Proc style coding
vs
OO style coding

Quality

- Performance
- Security
- Maintainability
- Reliability
- Availability
- Robustness
- …

Approach

- Caching
- Indexing
- Concurrency
- Pooling
- Data Virtualization
- Lazy Loading
- Reusability
- Extensible

1

5

If => Interface

# easy to code

# low cyclomatic complexity
# readablity
# unit test
# OCP

1

5

Flag => Interface

# easy to code

# low cyclomatic complexity
# readablity
# unit test
# OCP

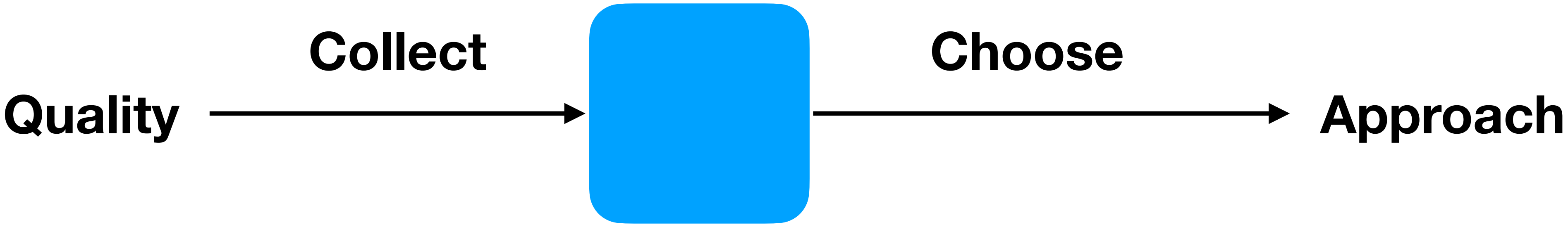
1

Flag => Polymorphism/
Abstraction/
Interface

Coupling => Polymorphism/
Abstraction/
Interface

Type check => Polymorphism/
Abstraction/
Interface

Down casting => Polymorphism/
Abstraction/
Interface



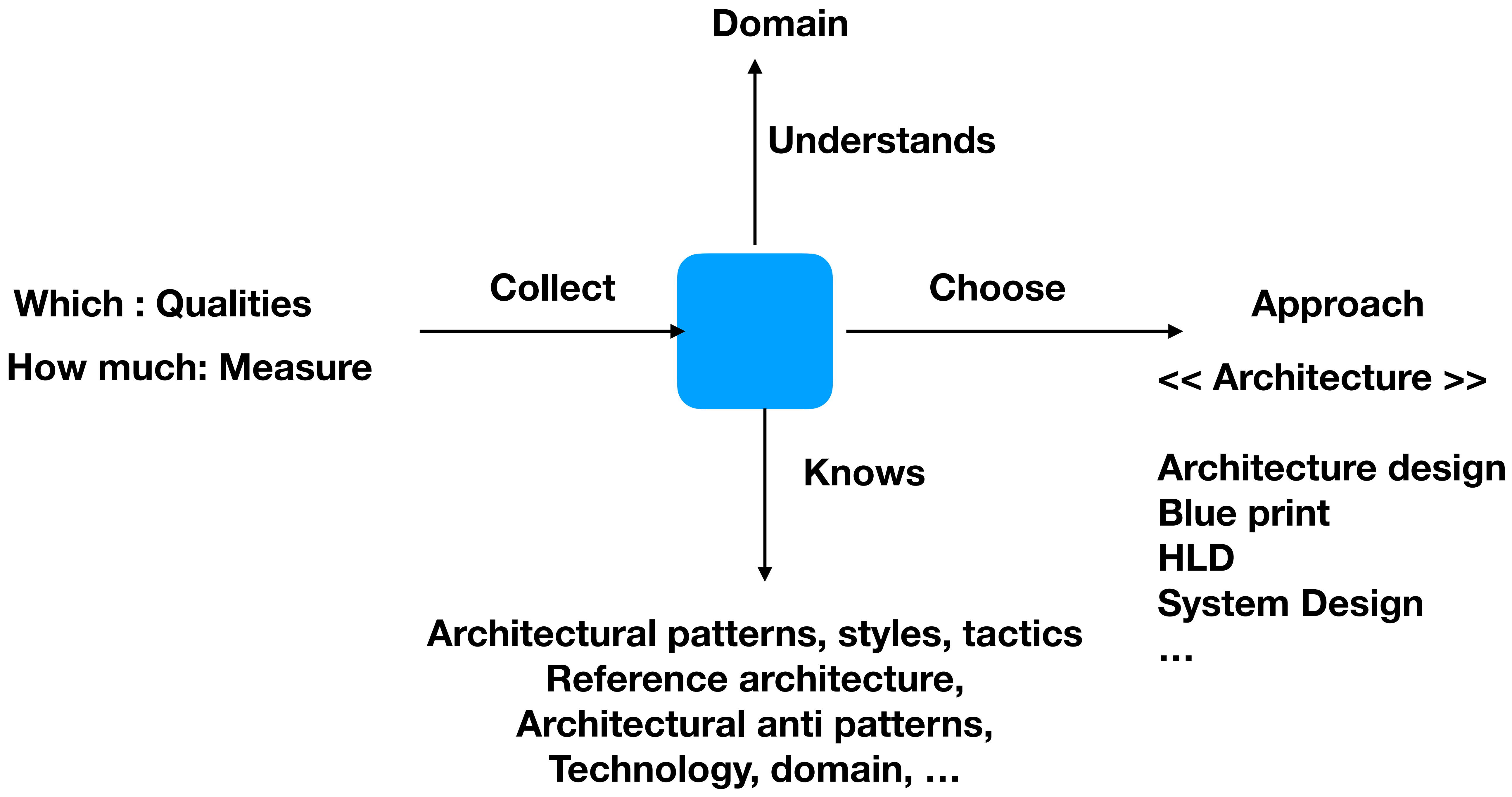
Quality

Collect

Choose

Approach

“system quality”



Domain

Understands

Which : Qualities

How much: Measure

Collect

Choose

Approach

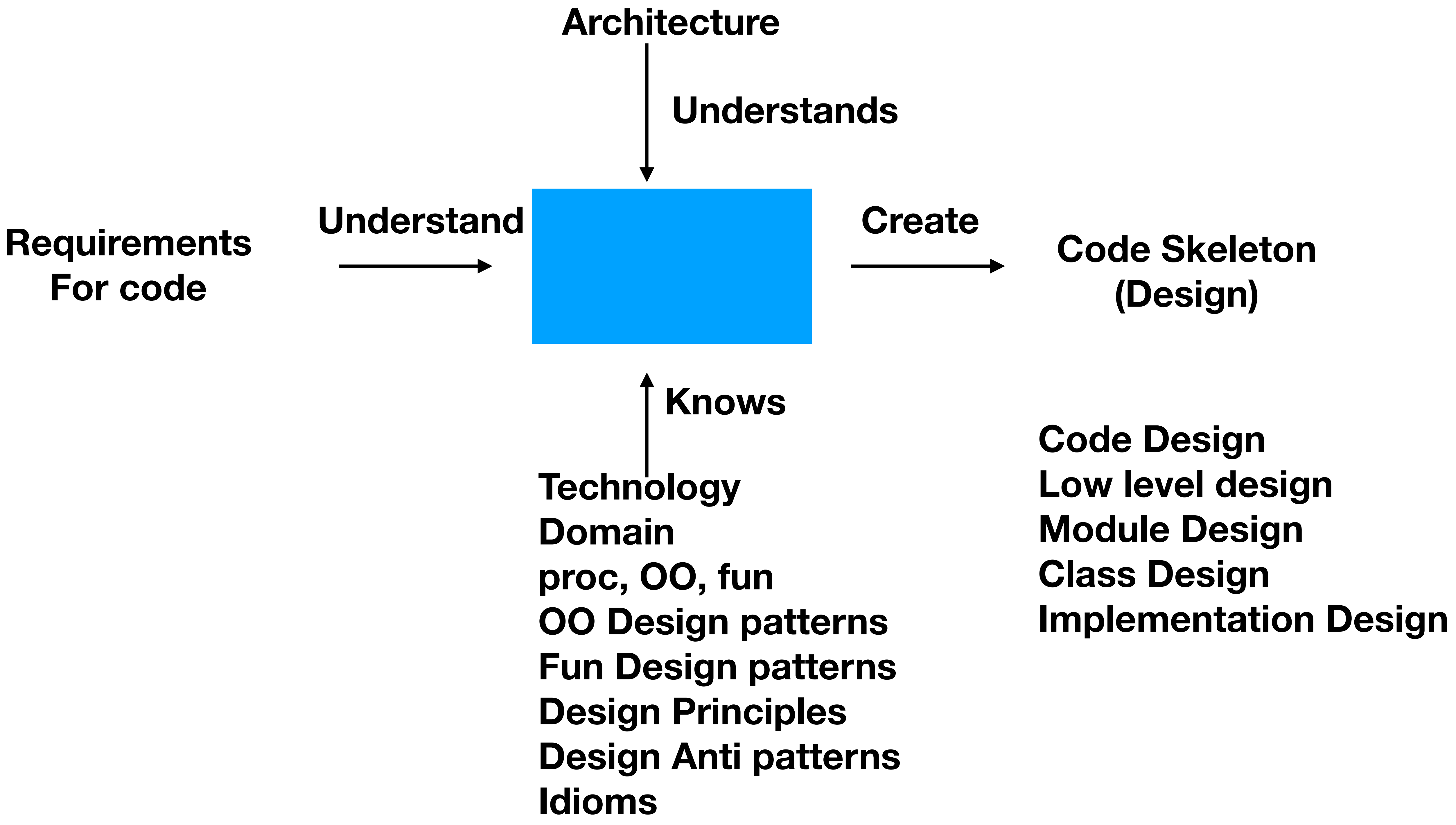
<< Architecture >>

Knows

Architecture design
Blue print
HLD
System Design
…

Architectural patterns, styles, tactics
Reference architecture,
Architectural anti patterns,
Technology, domain, …

Code Maintainability



Architecture

Understands

Requirements
For code

Understand

Create

Code Skeleton
(Design)

Knows

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

Code Design
Low level design
Module Design
Class Design
Implementation Design

- Interface Bird
    - Fly
    - Quack
    - Flap
    - Chirp

No discriminate in the family

```
fun(Bird bird){

    If type(…)
        bird.fly();
    ….

}
```

- Interface LivingThing
  - Walk
  - breathe
- Interface Bird extends LivingThing
  - Flap
  - Chirp
  -

No discriminate in the family

```
fun(Bird bird){
      ….
}
```

- Class Parrot

- Interface Bird

-

```
Interface Bird{
     fly
     sing
     buildNest
}
```

```
fun(Bird bird){

    bird.fly();
    ....

}
```

```
Interface LivingThing{
    eat
}
Interface Bird extends LivingThing{
    ?
}
```

```
fun(Bird bird){

       ....

}
```

```
Interface Bird{
    fly
    sing
    buildNest
}
```

```
fun(Bird bird){

    bird.fly();
    ….

}
```

```
Interface LivingThing{
}
Interface Bird extends LivingThing{
   ?
}
```

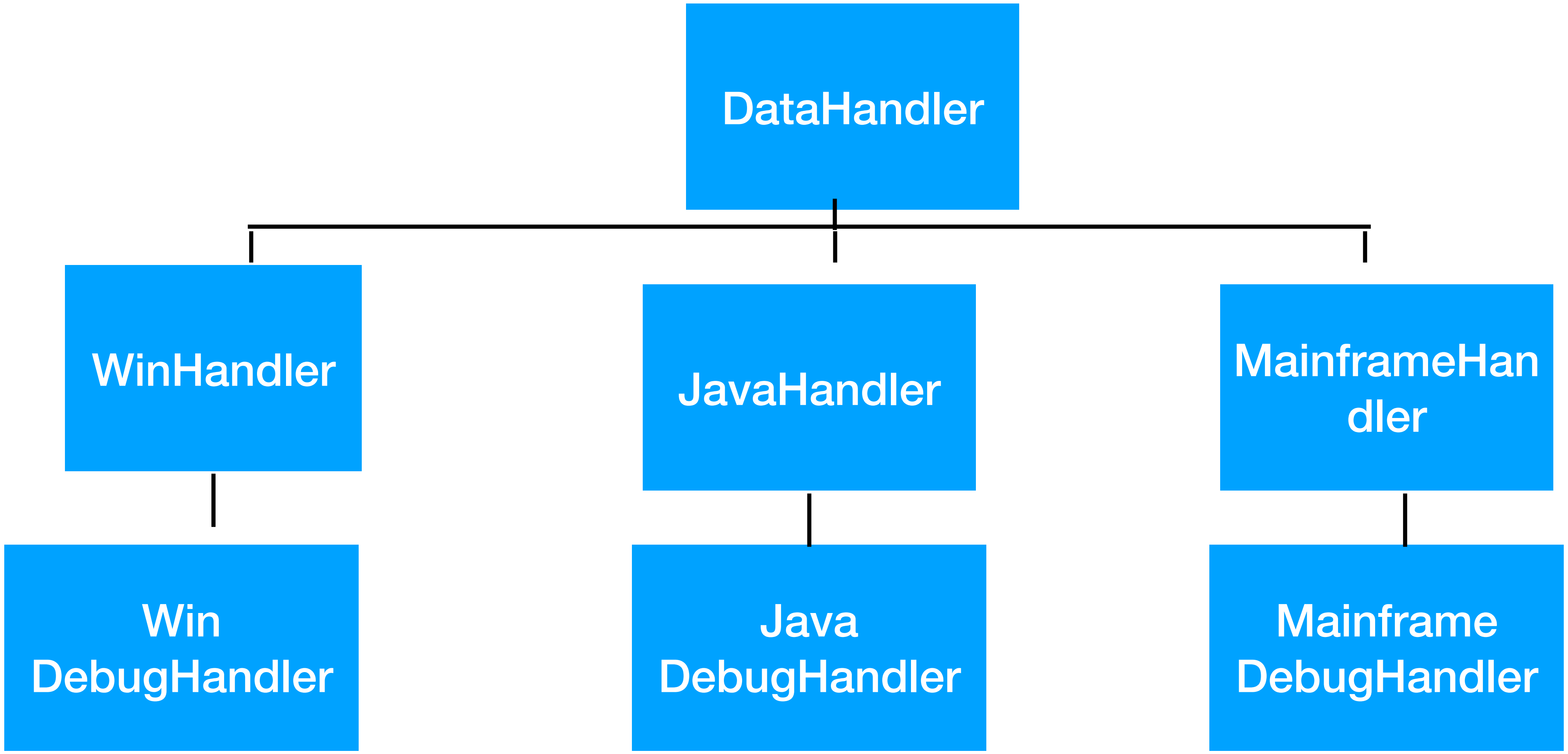
```
Interface Bird extends LivingThing{

}
```

```
fun(Bird bird){


     ….

}
```

```
Class Parrot{
}
```

```
Interface Bird{
}
```

```
interface LivingThing{
    …
}

interface Bird extends LivingThing{
    chirp
    sound()
}
Interface FlyingBird extends Bird{
    fly()
}
Interface NestBuildingBird extends Bird{
    makeNest()
}

…
   layEggs()
    swim()
   }
```



DataHandler

WinHandler

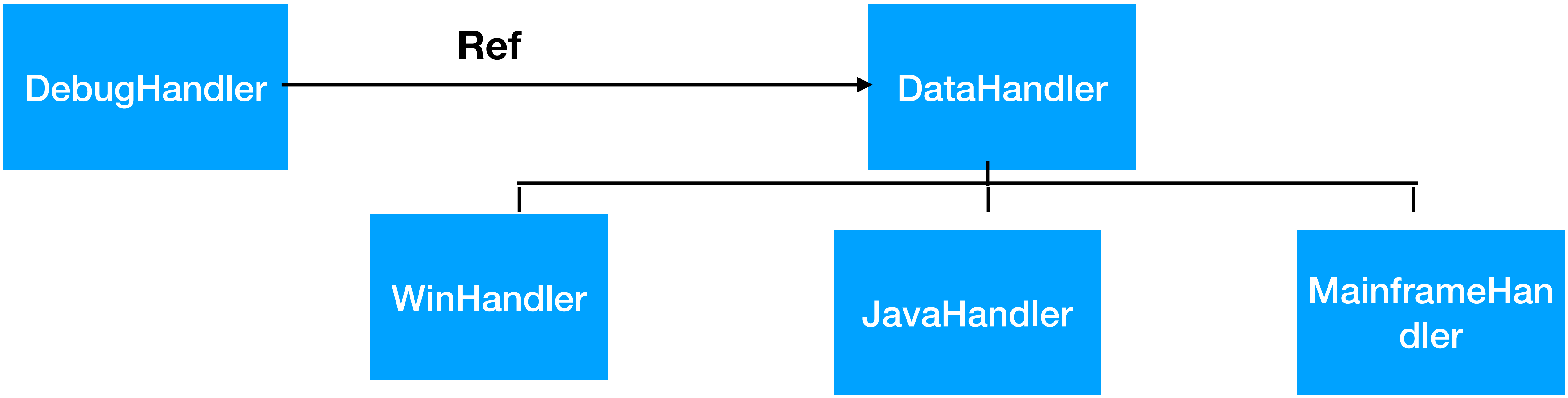
JavaHandler

MainframeHandler

Win DebugHandler

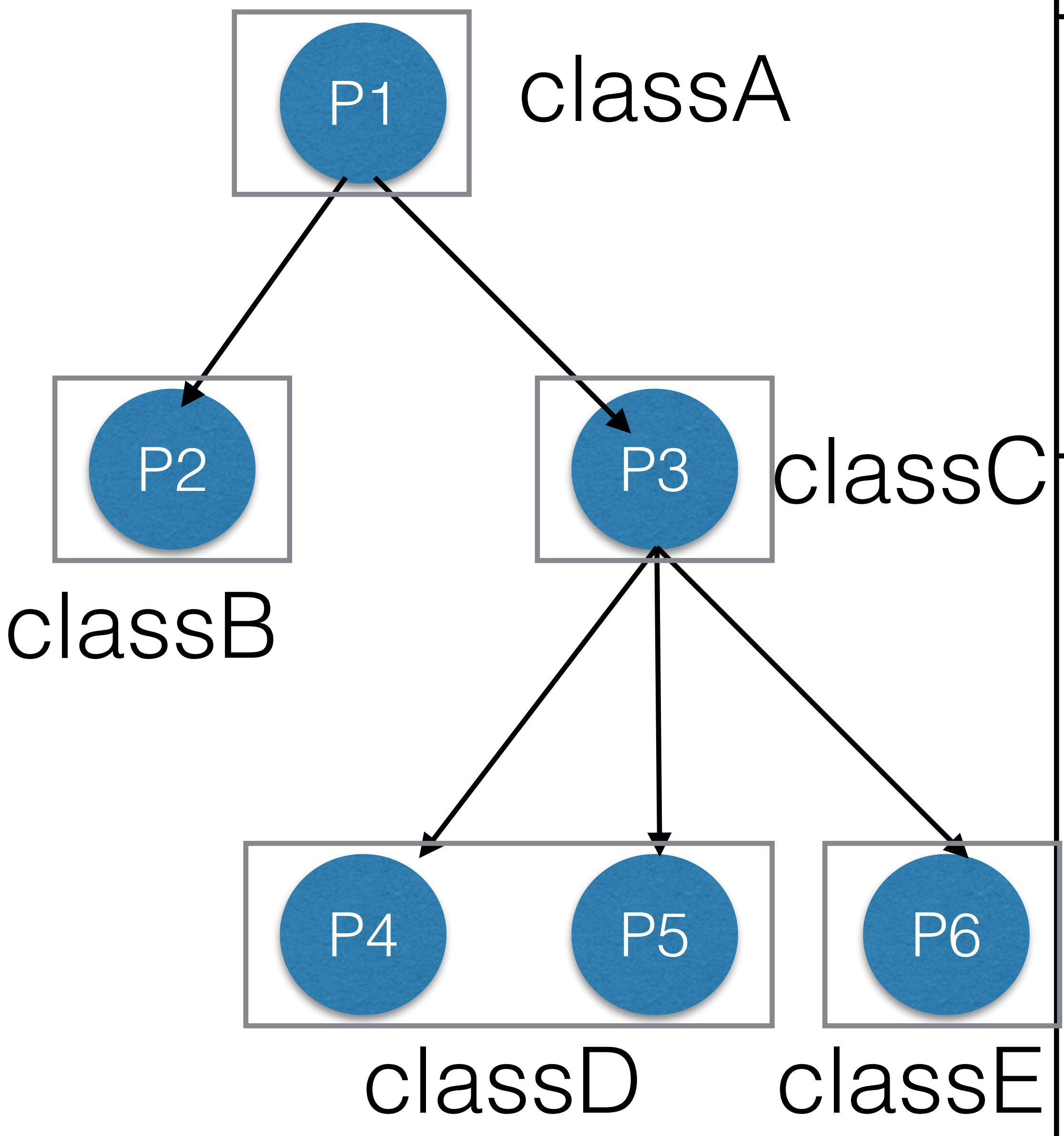
Java DebugHandler

Mainframe DebugHandler


DebugHandler
Ref
DataHandler
WinHandler
JavaHandler
MainframeHan
dler

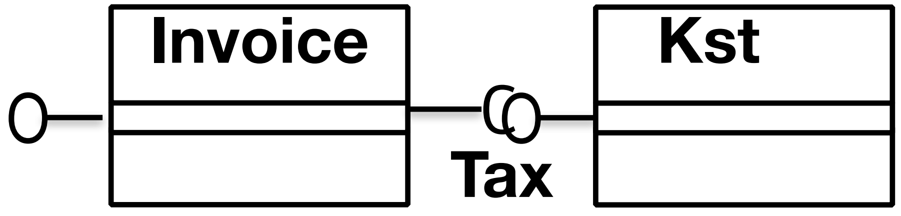
a+b	3 cpu cycles
Fun call	10 cpu cycles
Exception handling	1000 cpu cycles
Create thread	200,000 cpu cycles
Write to file	10,00,000 cpu cycles
Db call	40,00,000 cpu cycles

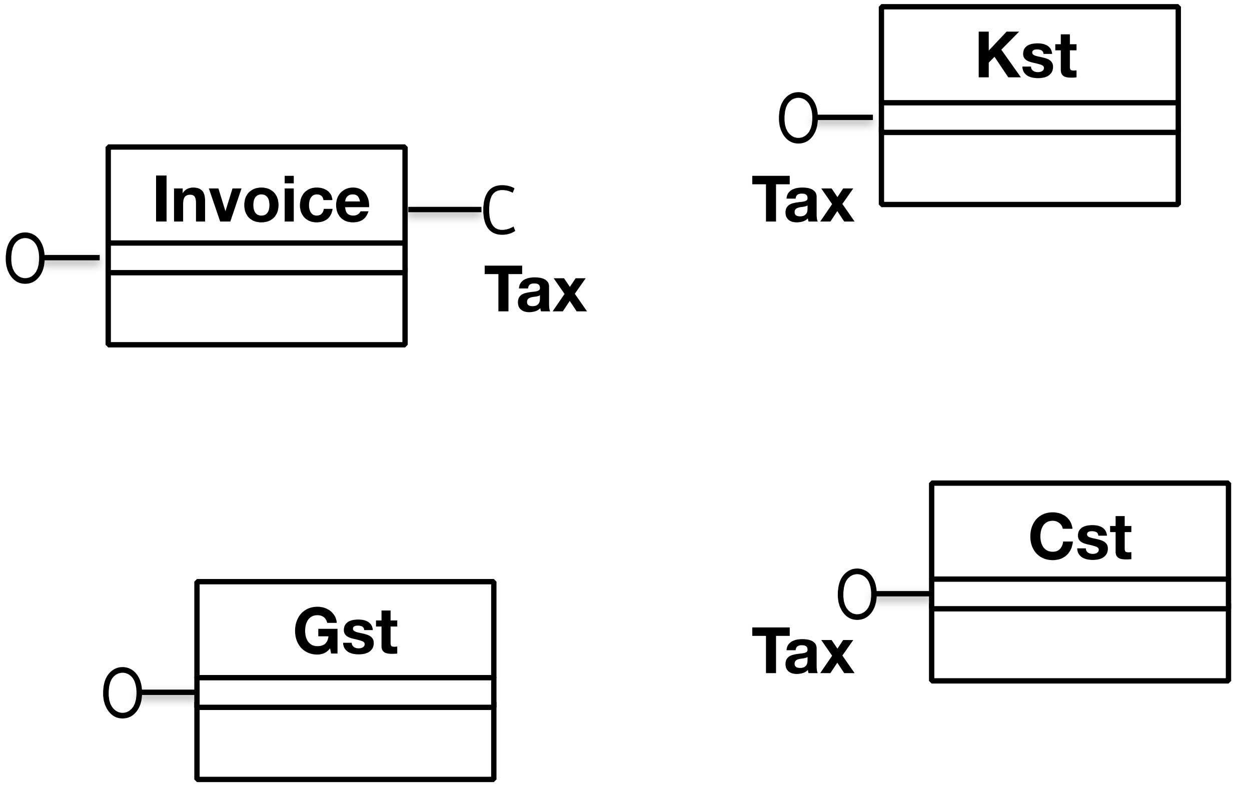
Procedural Prog
(tree)



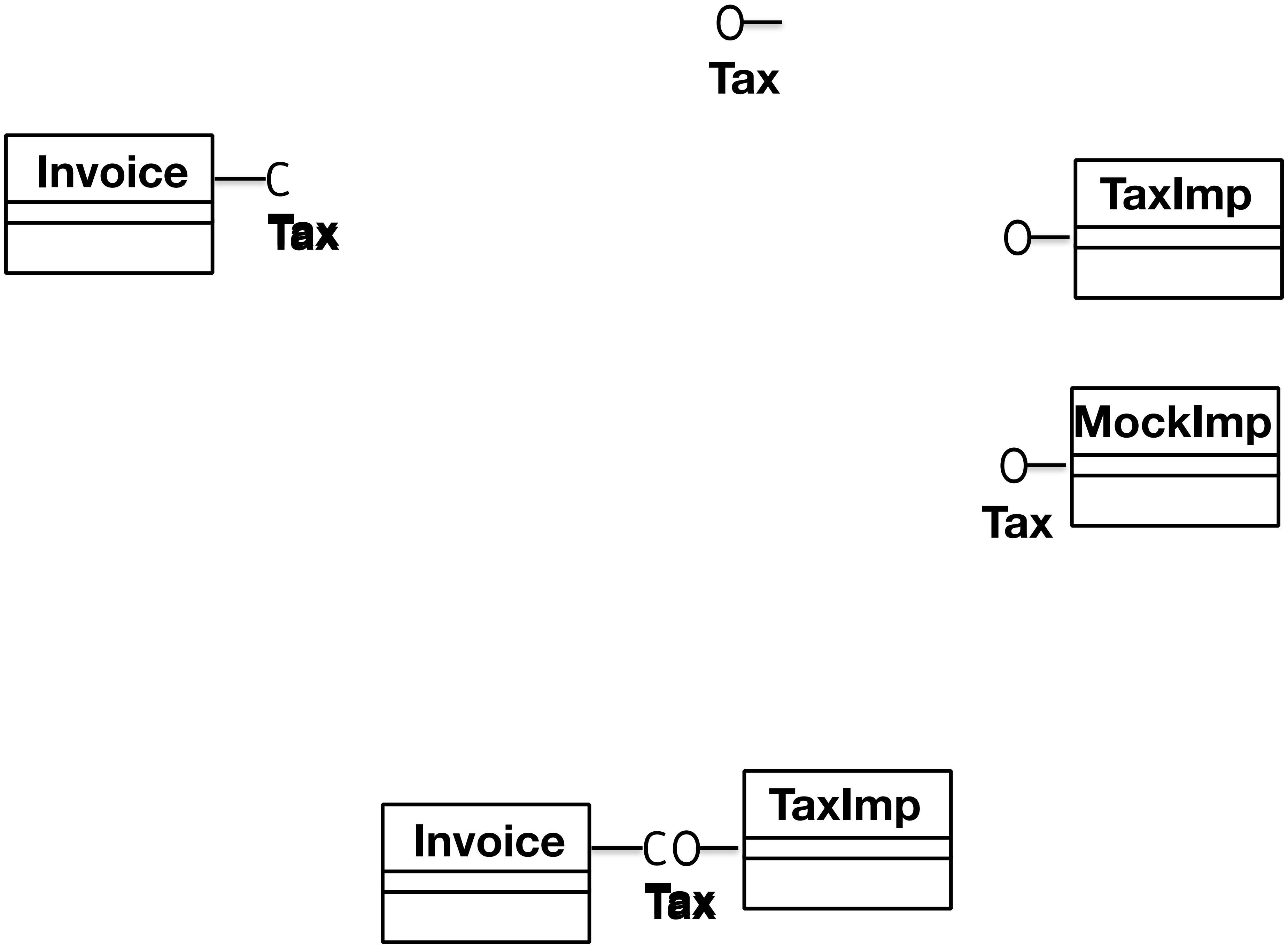
Left

OO Prog
(Lego)

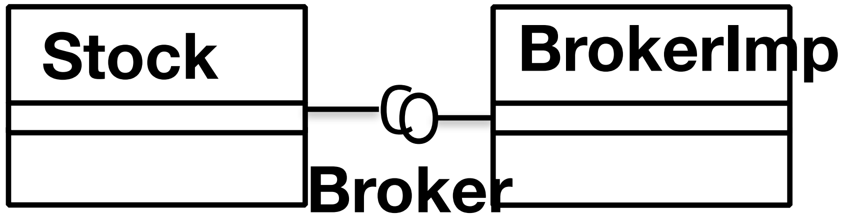




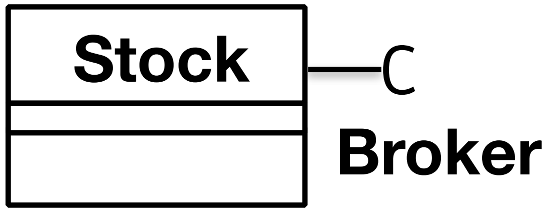
Right

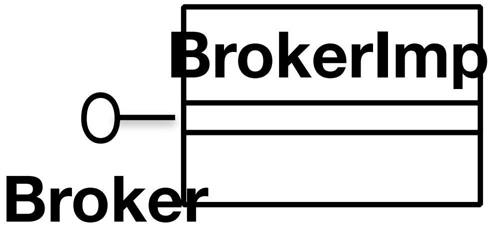

Tax
Invoice
Tax
TaxImp
MockImp
Tax
Invoice
Tax
TaxImp

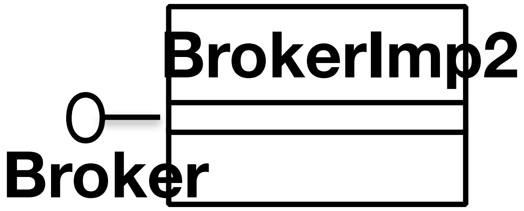
Flow


Stock
BrokerImp
Broker

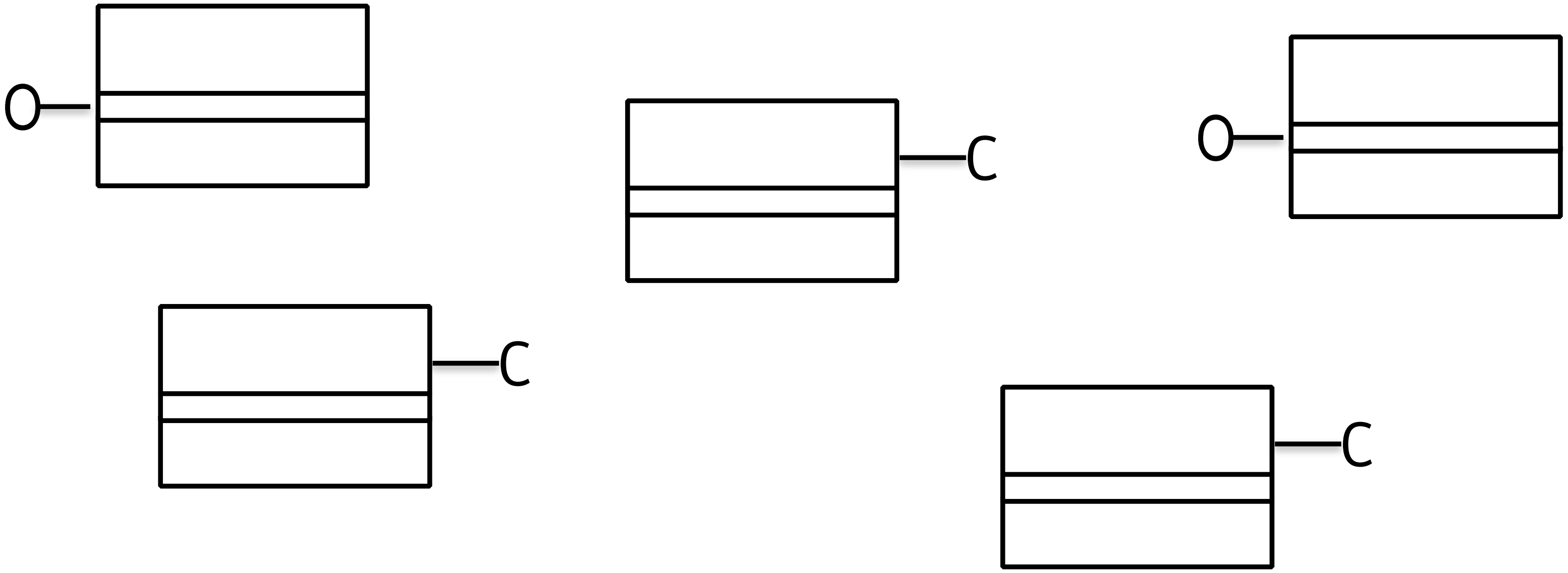
Steps

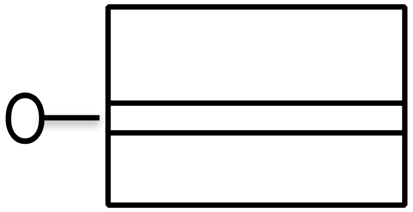

Stock
Broker


BrokerImp
Broker


BrokerImp2
Broker

© Shutterstock / 3d_kot

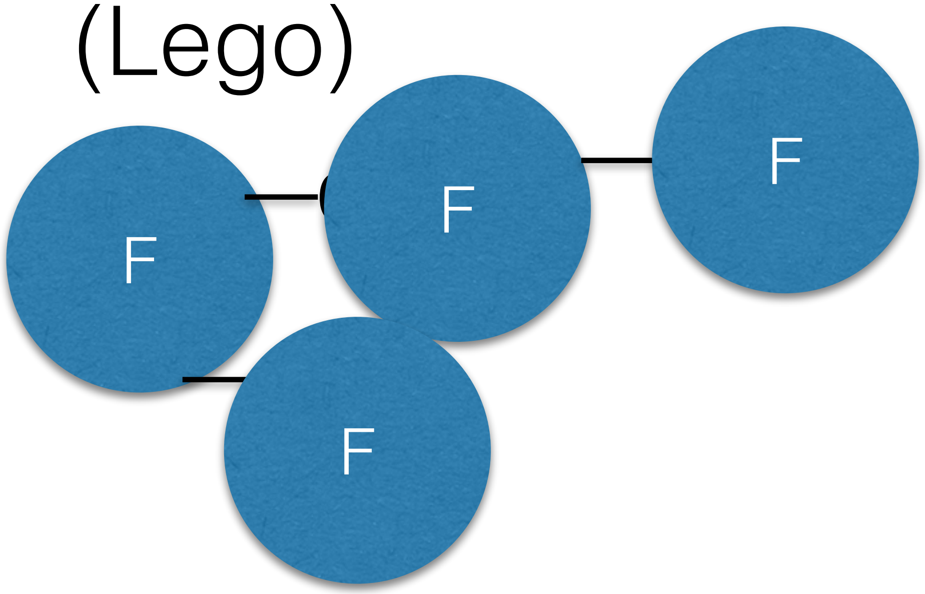


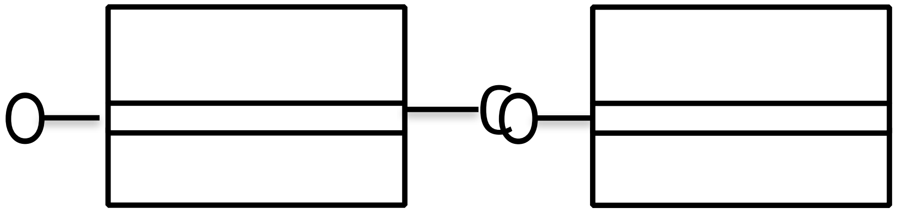


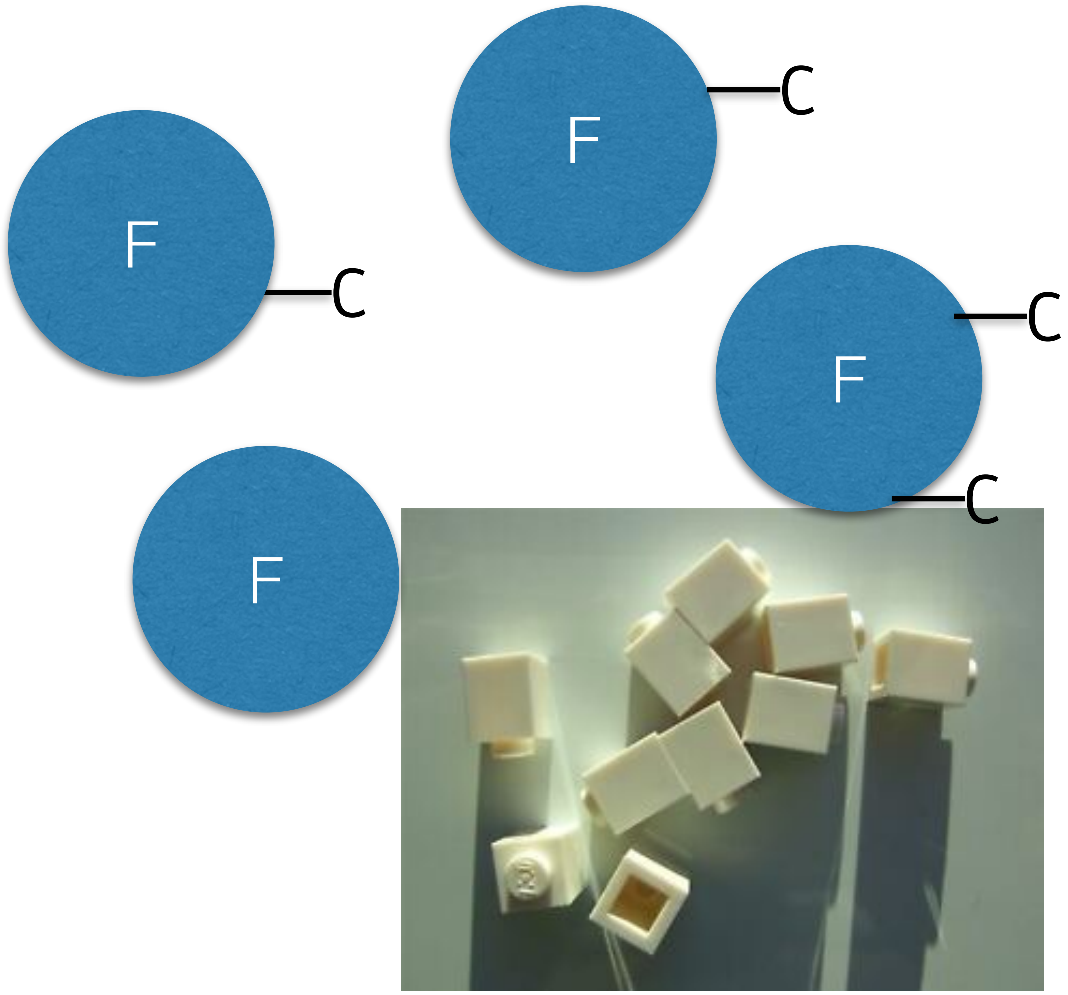
	Proc	**OO**	**Functional**
Performance	n/a	n/a	+ +
Security	n/a	n/a	n/a
Learning Curve	++ ●	- -	-
Development Effort	++ ●	- -	-
Unit test	- -	+ + ●	+ + + ●
Less Coupling	- -	+ +	+ +
Manage large code	- -	++ ●	+ ●
Concurrency	- -	- -	+ + ●

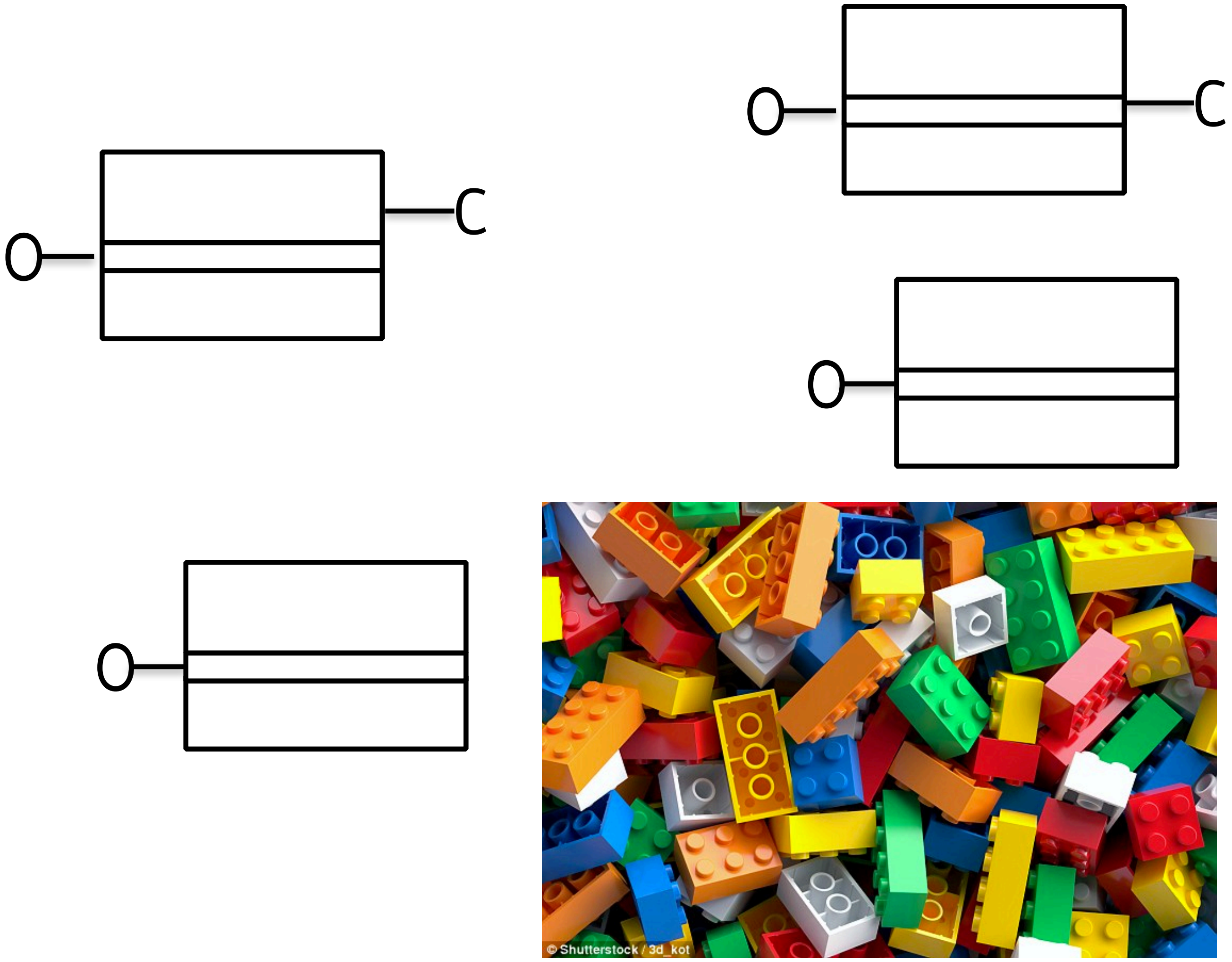
Functional Prog (Lego)

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( ))

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

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

do(Parrot parrot)
{
    parrot.fly();
}
	do(Bird bird)
	{
	    bird.fly();
	}
		do(bird)
		{
		    bird.fly();
		}
			do(Lamda fly)
			{
			    fly();
			}

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

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

do2(bird)
{
    bird.fly();
    bird.buildNest();
}

do3(bird)
{
    bird.fly();
    bird.buildNest();
    bird.layEggs();
}
```

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 f1)
		{
		   f1();
		}

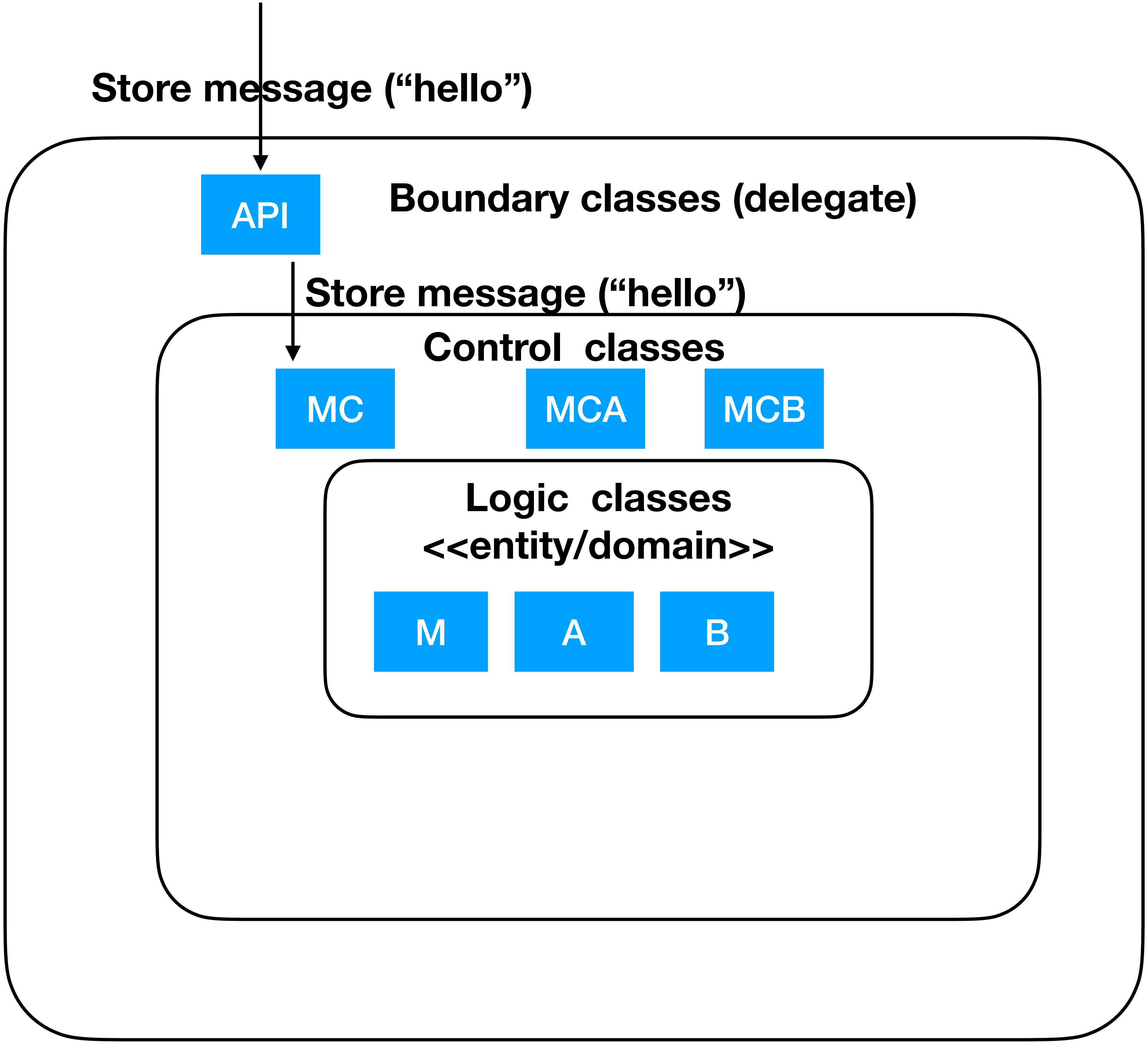
do(new Parrot( ))	do(new Parrot( ))	CA obj = new CA( ) do(()-> obj.fly())

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 f1)
			{
			f1();
			}
				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”)



Store message ("hello")

API

Boundary classes (delegate)

Store message ("hello")

Control classes

MC

MCA

MCB

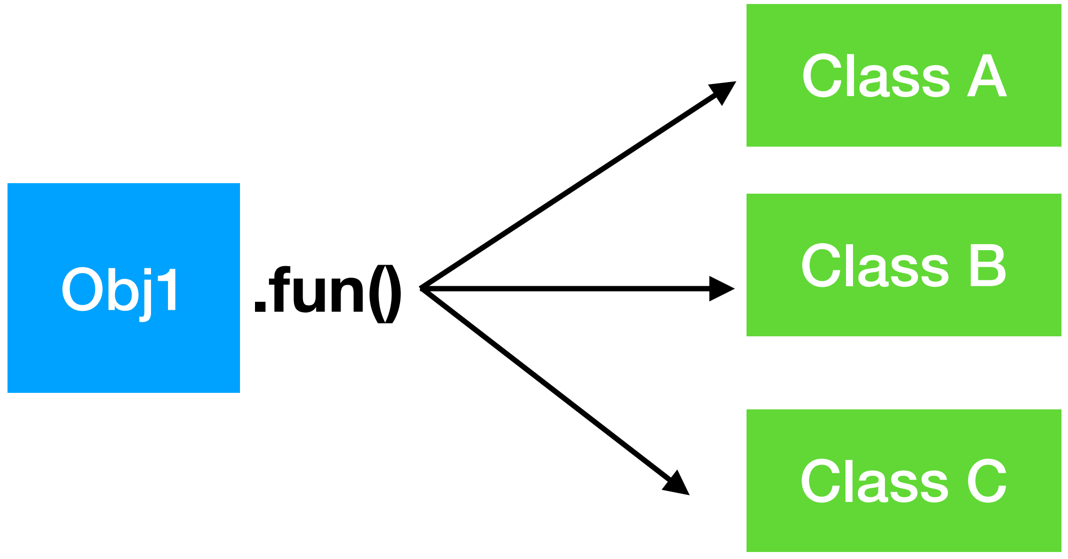
Logic classes <<entity/domain>>

M

A

B

Single dispatch



Obj1 .fun()

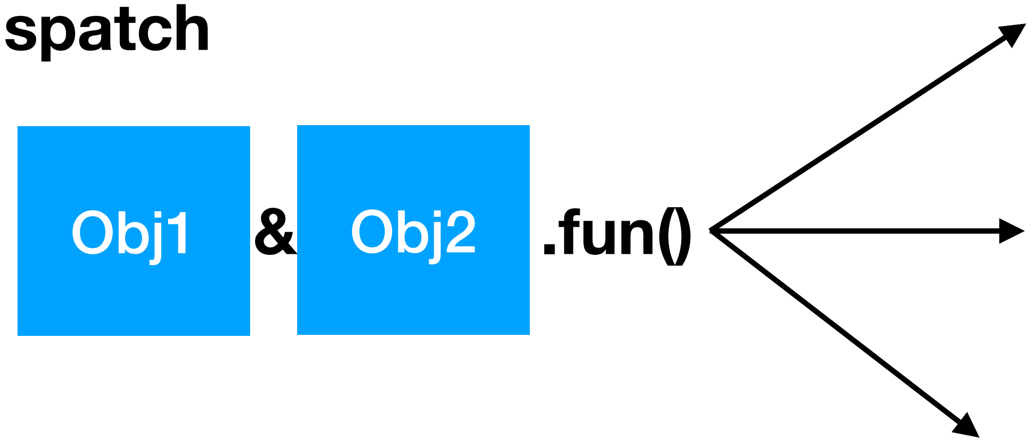
Class A

Class B

Class C

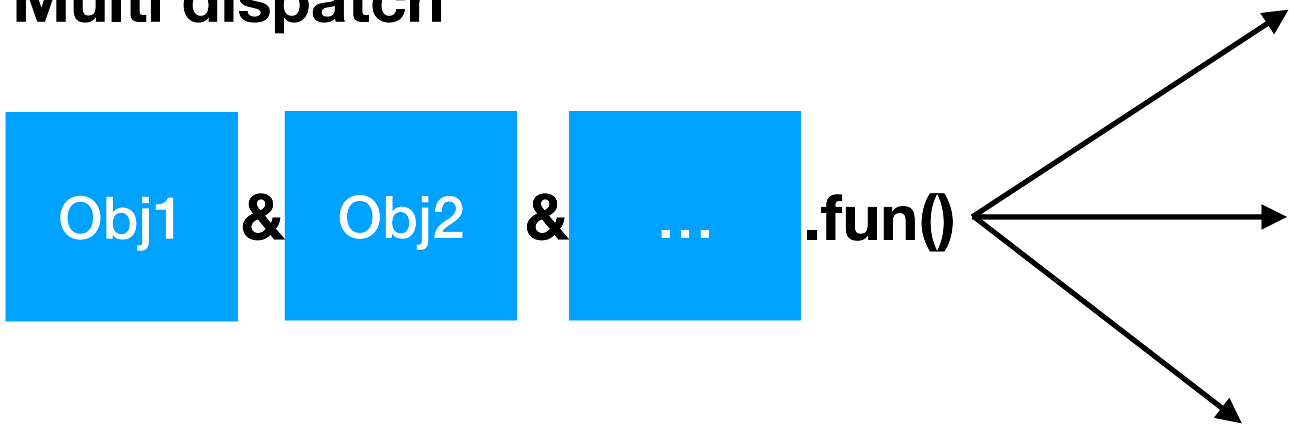
Java, c++, py, ..

Dual dispatch

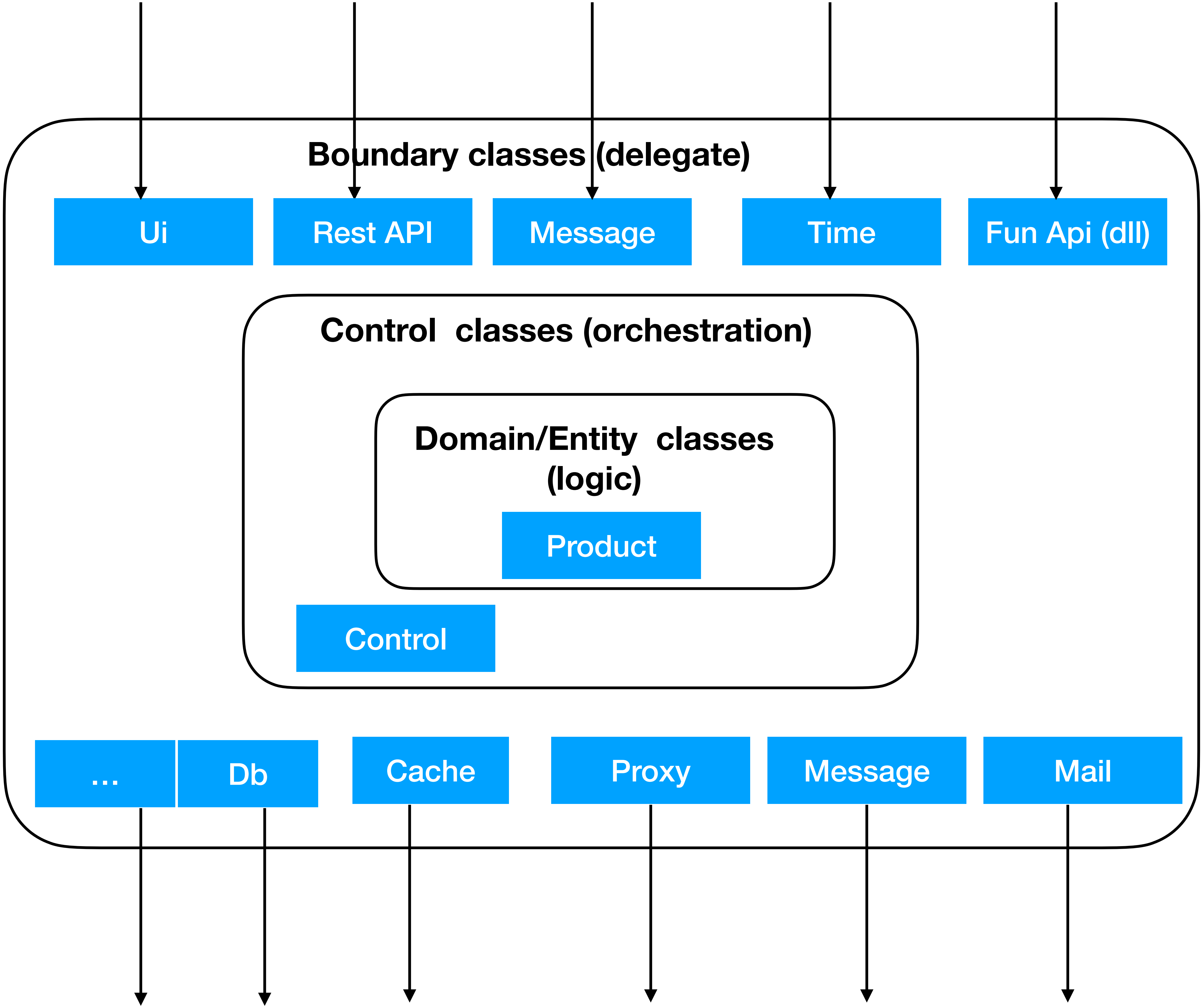


Obj1 & Obj2 .fun()

Multi dispatch



Obj1 & Obj2 & … .fun()



Boundary classes (delegate)

Ui

Rest API

Message

Time

Fun Api (dll)

Control classes (orchestration)

Domain/Entity classes (logic)

Product

Control

...

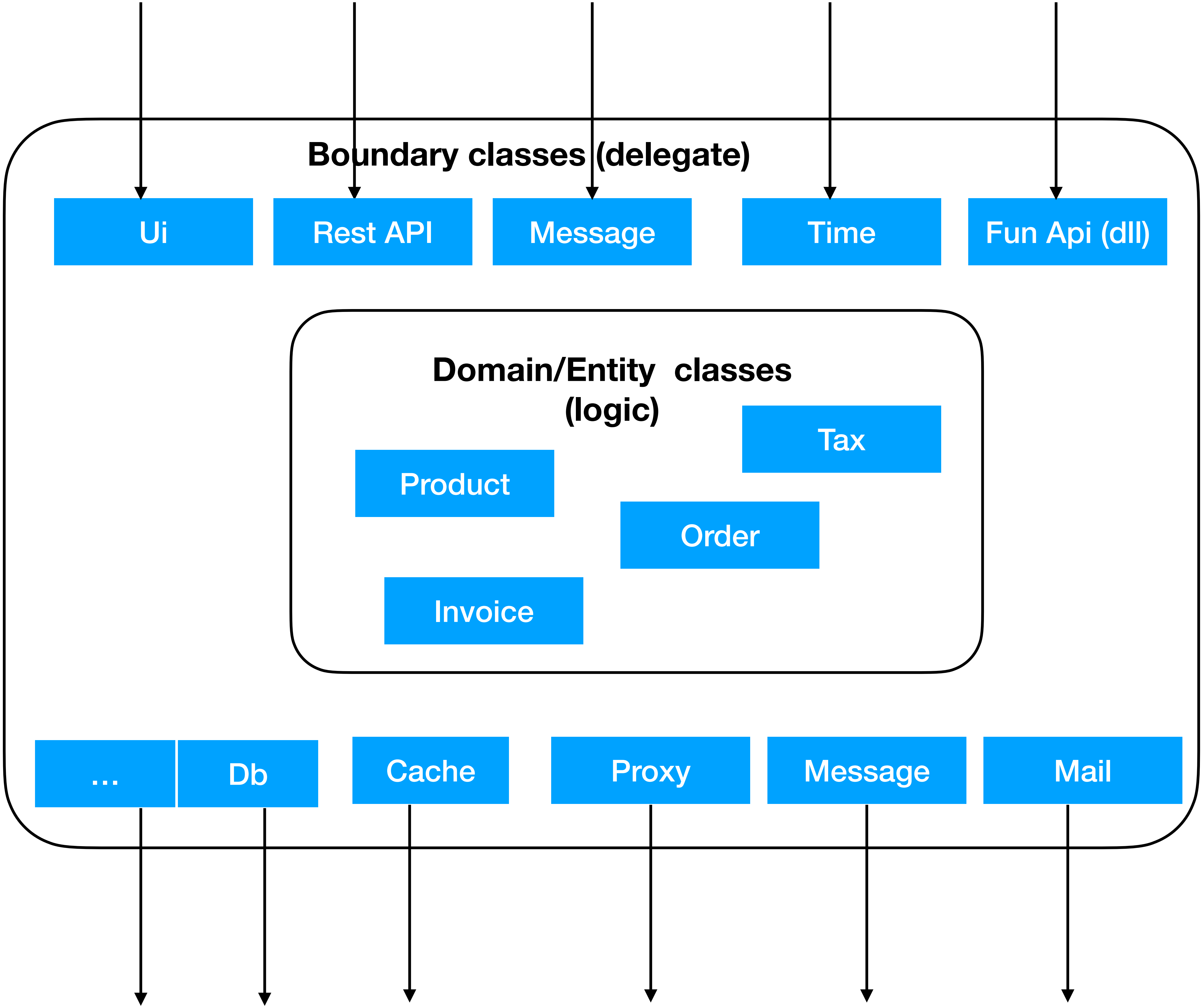
Db

Cache

Proxy

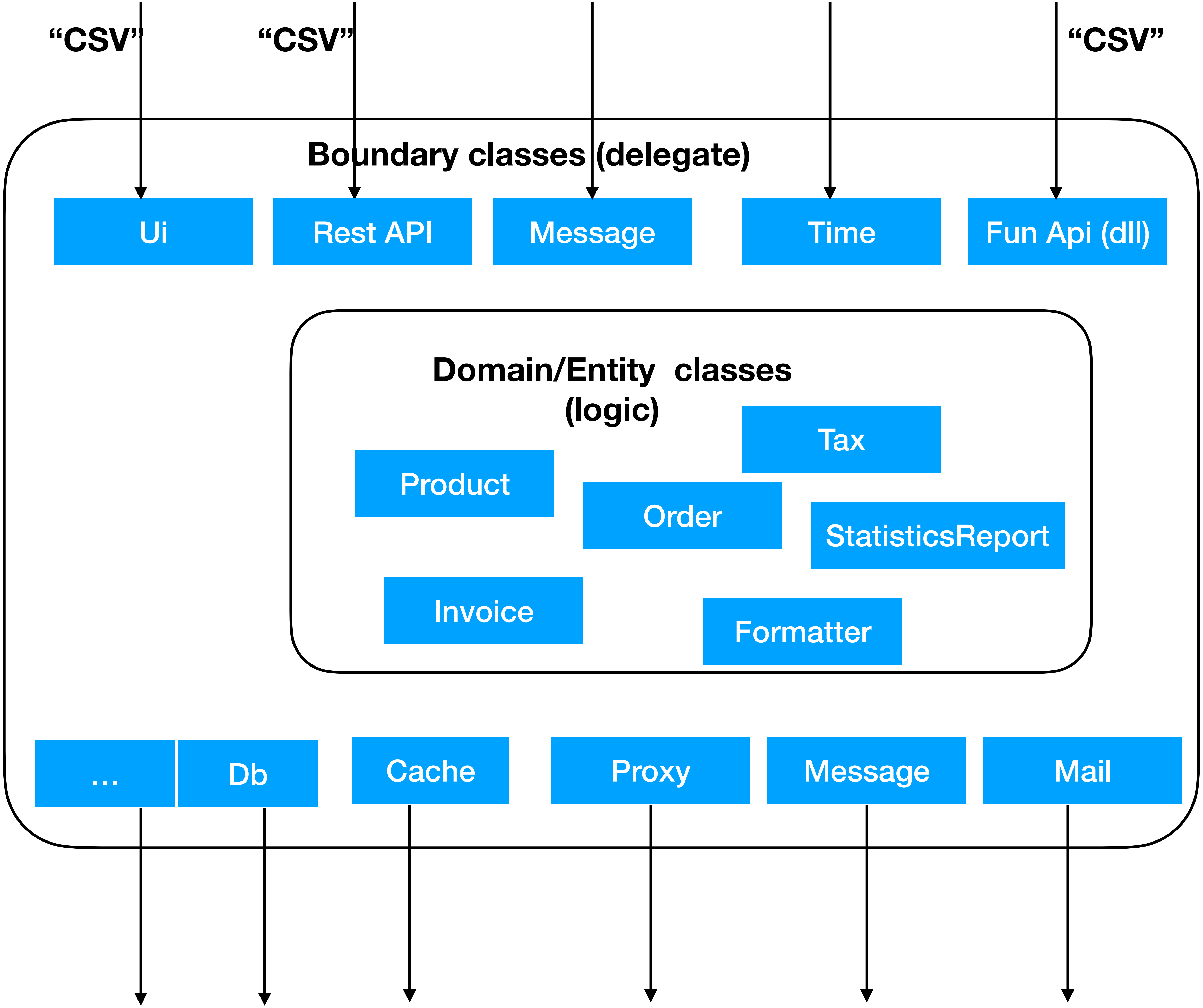
Message

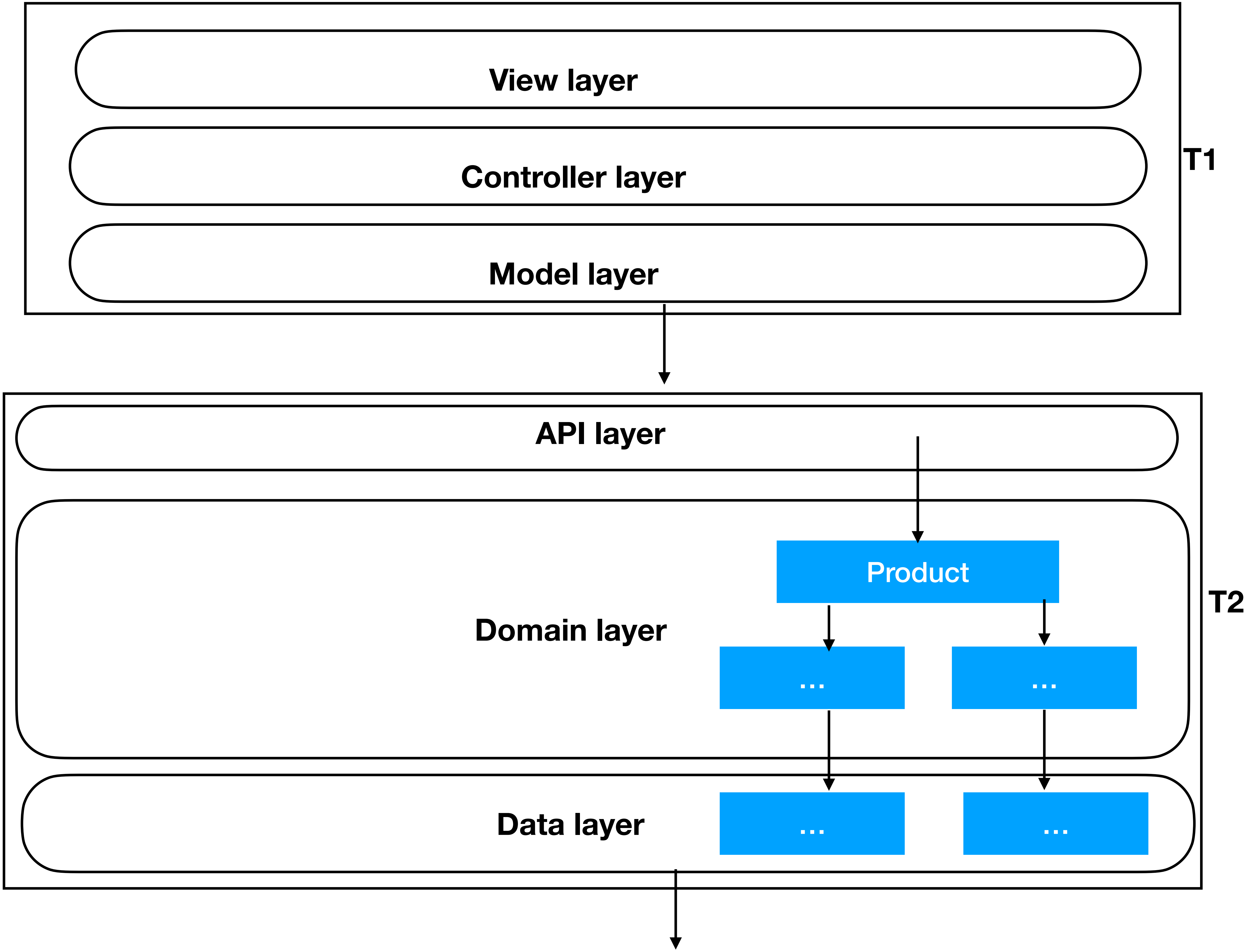
Mail

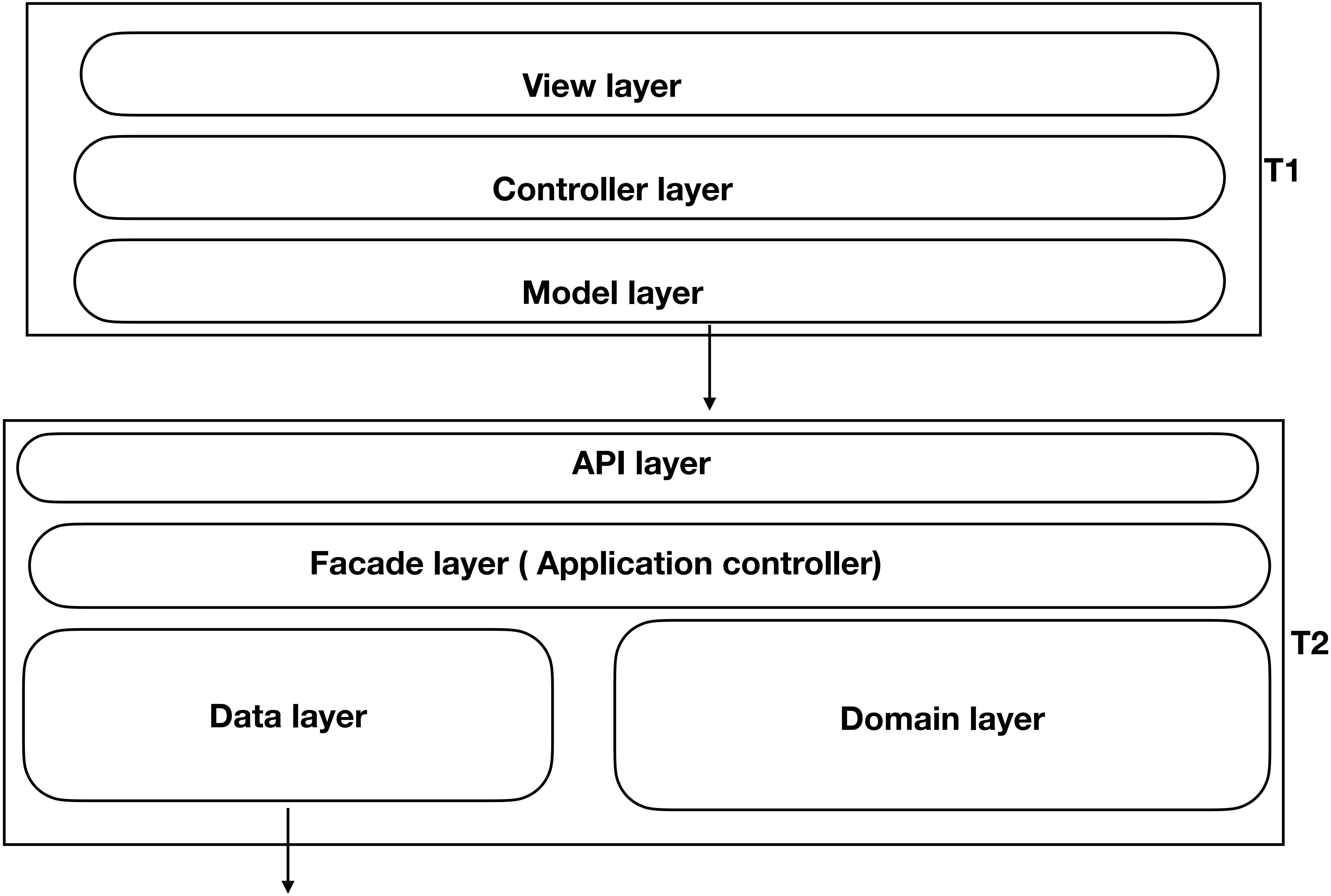


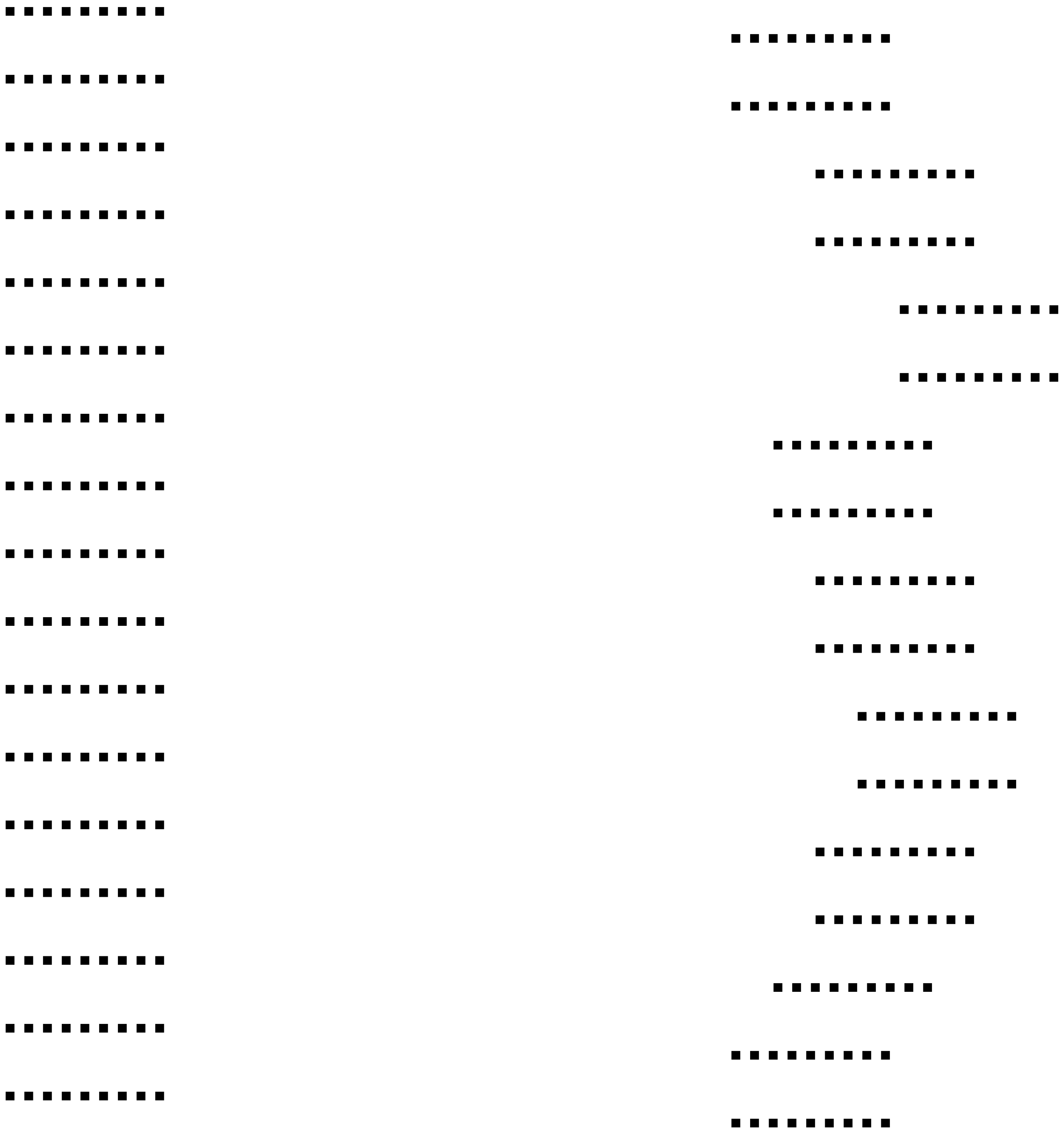
Boundary classes (delegate)

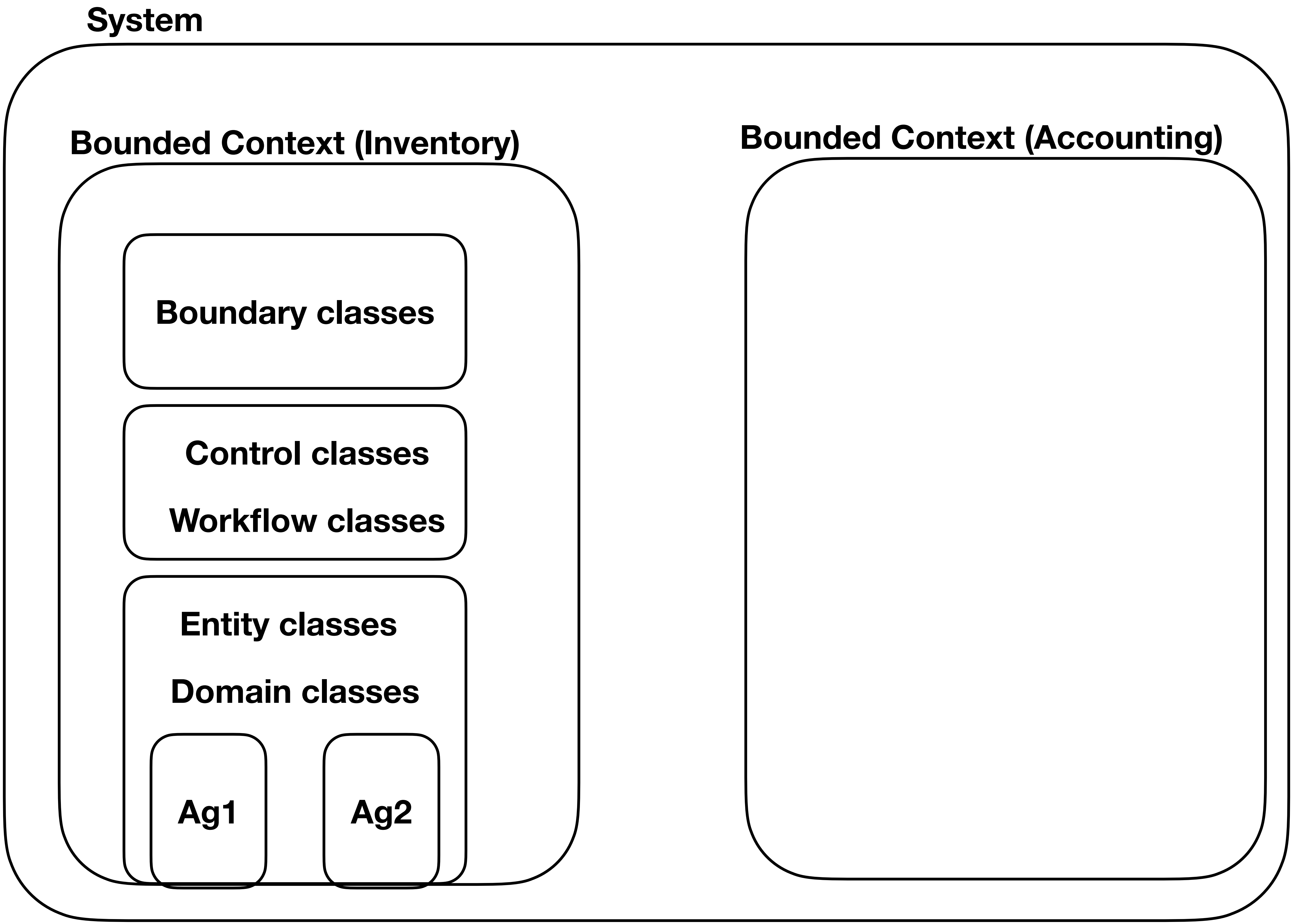
Domain/Entity classes (logic)


"CSV"
"CSV"
"CSV"
Boundary classes (delegate)
Ui
Rest API
Message
Time
Fun Api (dll)
Domain/Entity classes
(logic)
Tax
Product
Order
StatisticsReport
Invoice
Formatter
...
Db
Cache
Proxy
Message
Mail


View layer
Controller layer
Model layer
T1
API layer
Product
Domain layer
T2
...
...
Data layer
...
...

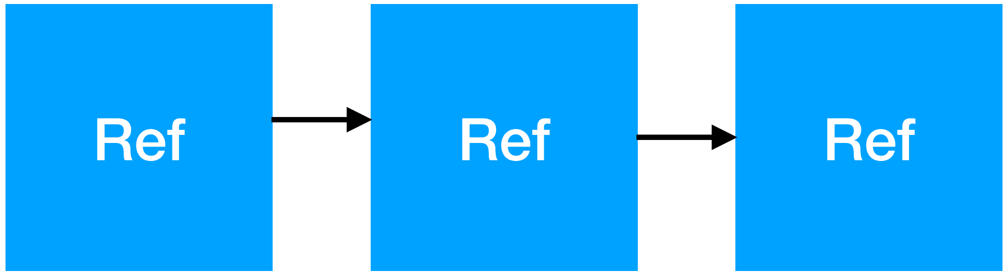

View layer
Controller layer
Model layer
T1
API layer
Facade layer (Application controller)
Data layer
Domain layer
T2

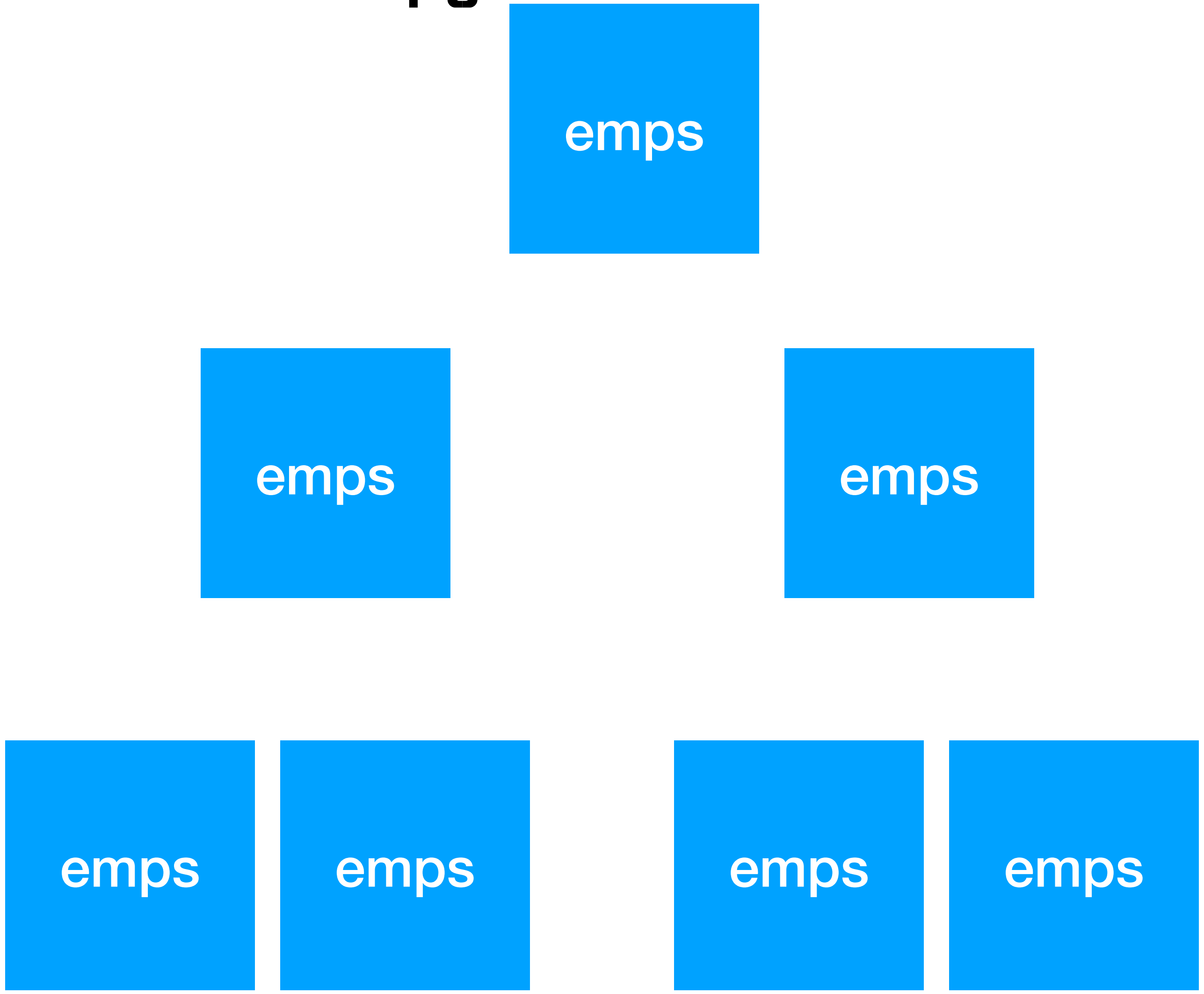



System
Bounded Context (Inventory)
Bounded Context (Accounting)
Boundary classes
Control classes
Workflow classes
Entity classes
Domain classes
Ag1
Ag2

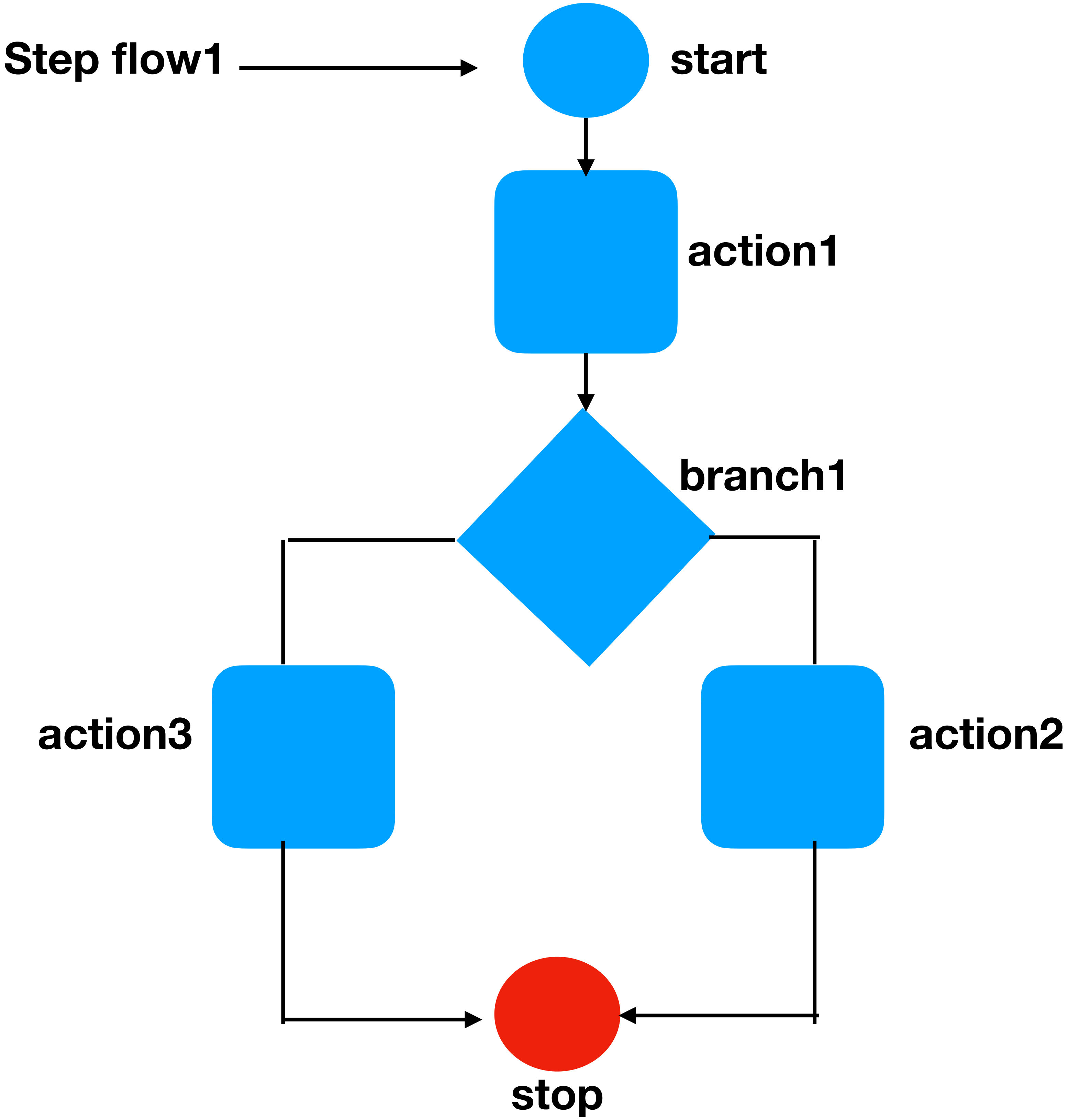
```
Class Emp{
   Emp ref;

}
```

```
Class Emp{
   List<Emp> emps;

}
Class FTE extends Emp{ }
Class PTE extends Emp{}
```

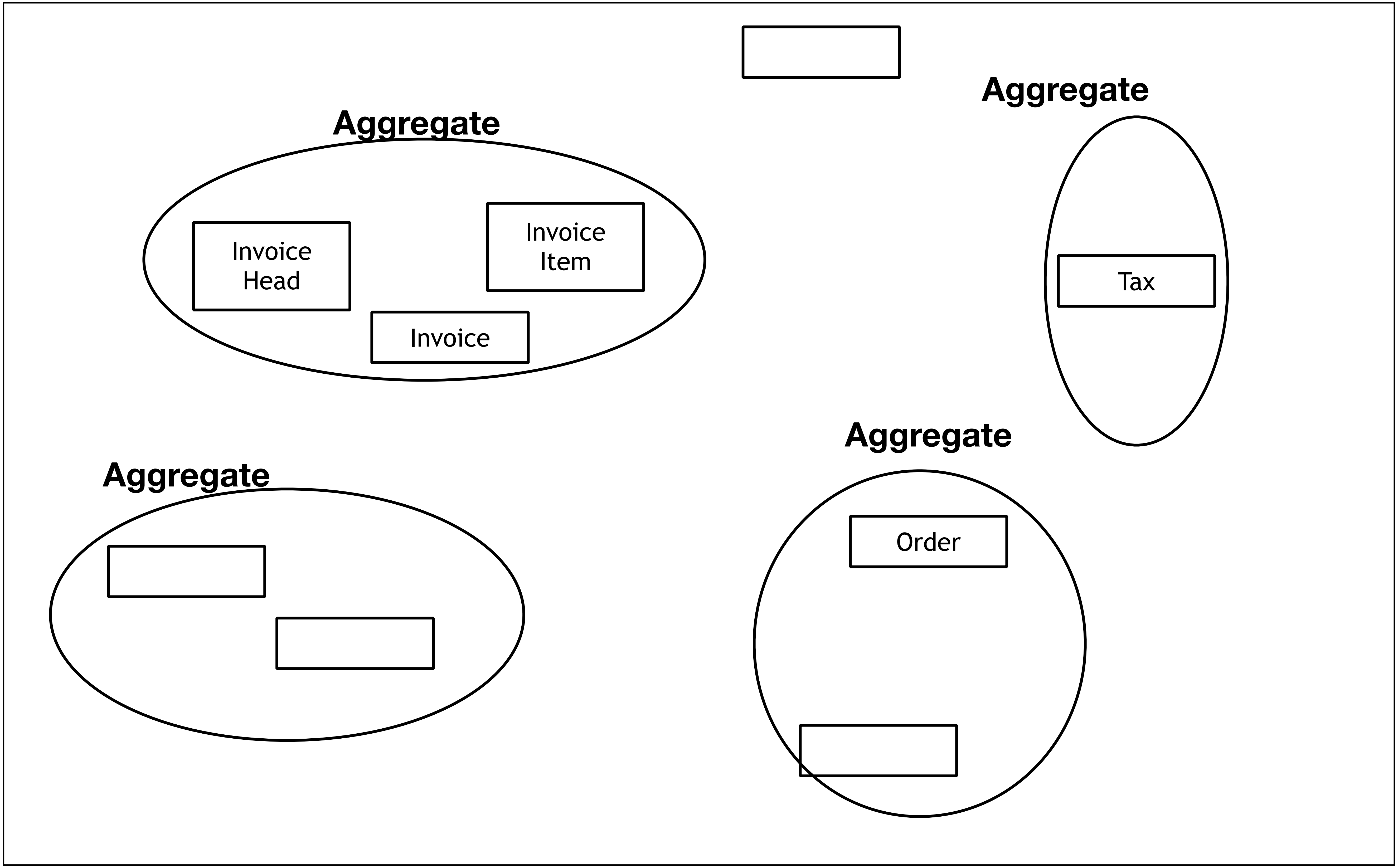


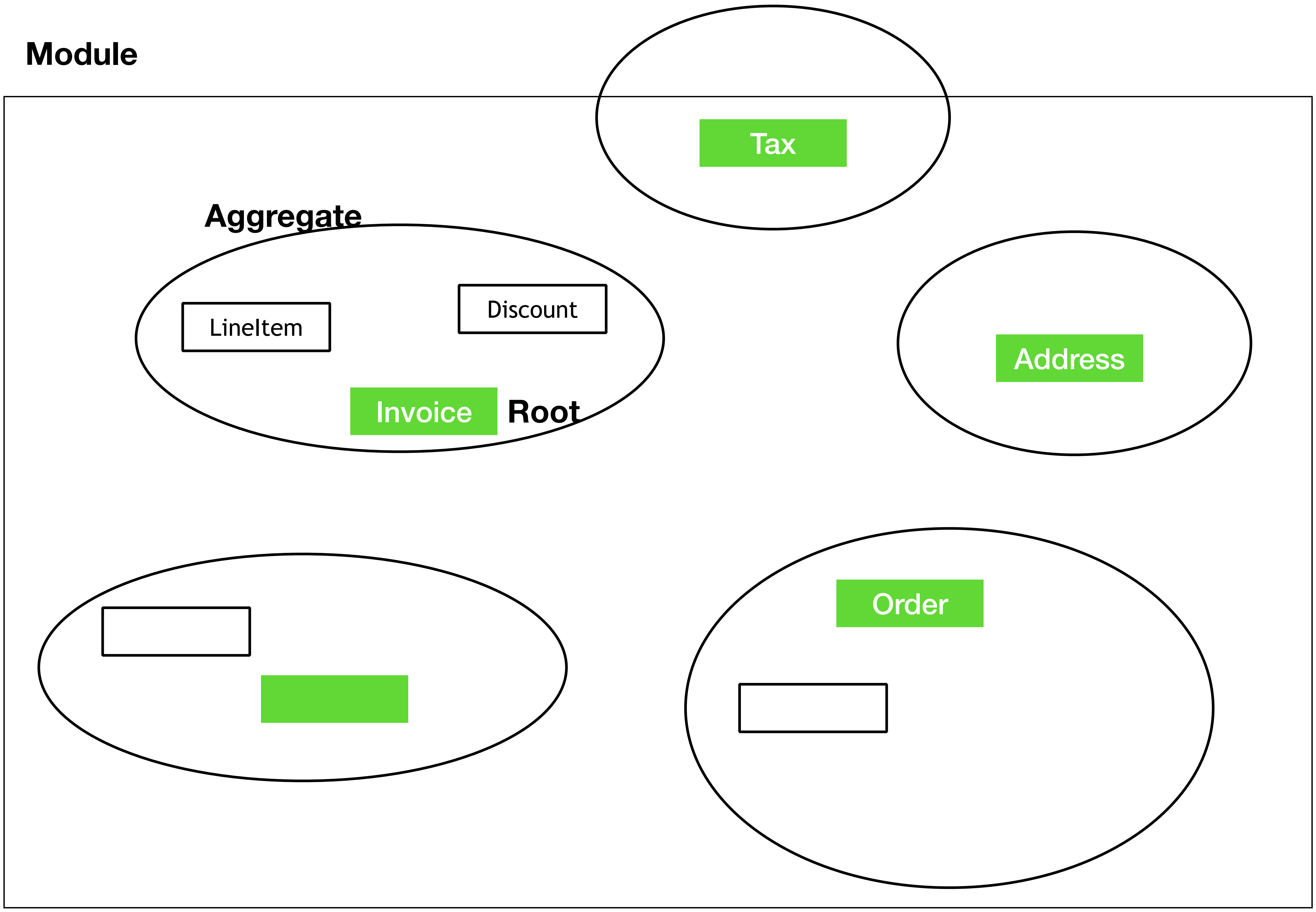


```
Class Emp{
   Emp first;
   Emp second;

}
```

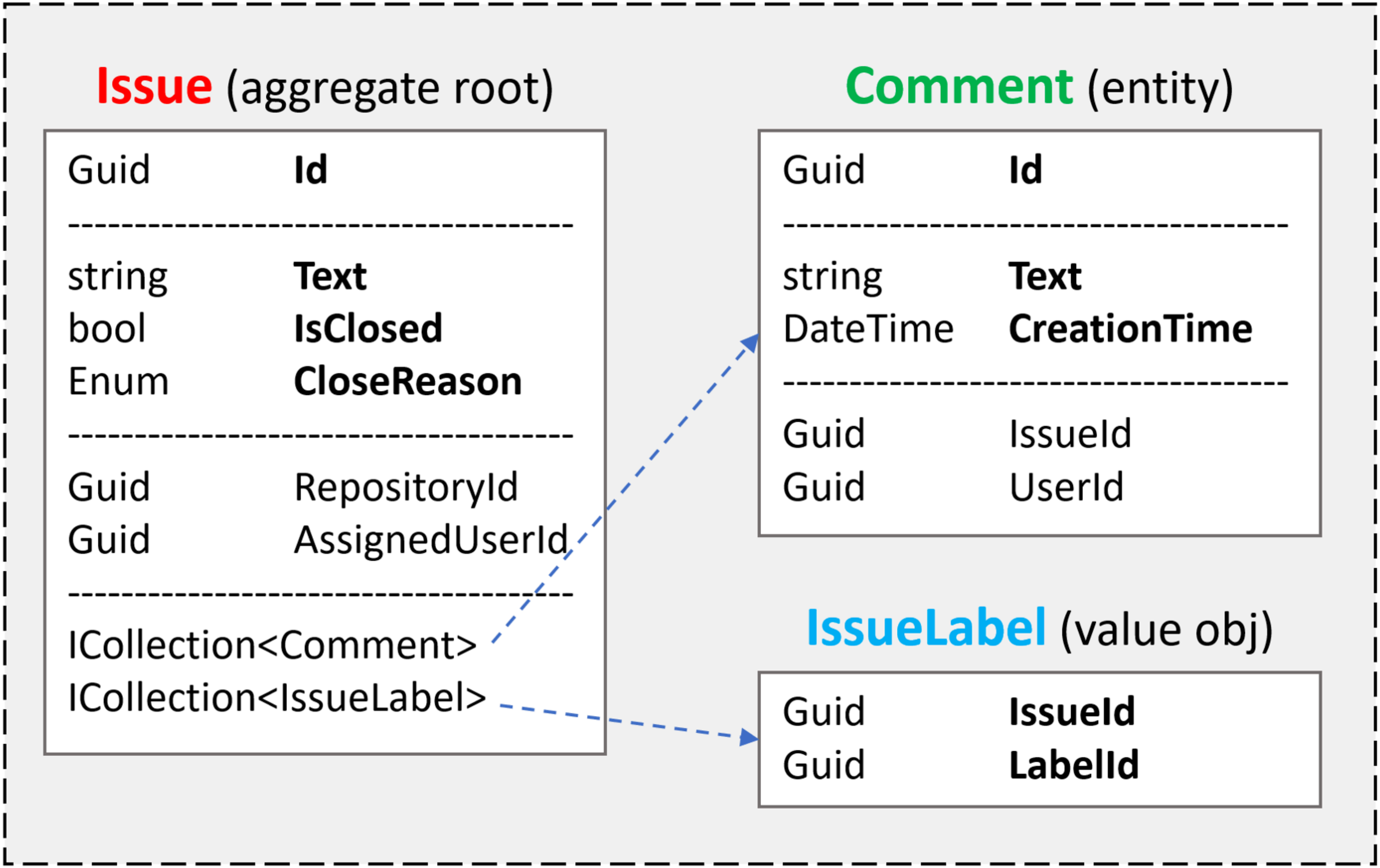

Step flow1
start
action1
branch1
action3
action2
stop

DDD


Aggregate
Aggregate
Invoice Head
Invoice Item
Invoice
Tax
Aggregate
Aggregate
Order


Module
Tax
Aggregate
LineItem
Discount
Invoice
Root
Address
Order

Issue Aggregate



Issue (aggregate root)

Guid **Id**

----

string **Text**

bool **IsClosed**

Enum **CloseReason**

Guid RepositoryId

Guid AssignedUserId

ICollection<Comment>

ICollection<IssueLabel>

Comment (entity)

Guid **Id**

----

string **Text**

DateTime **CreationTime**

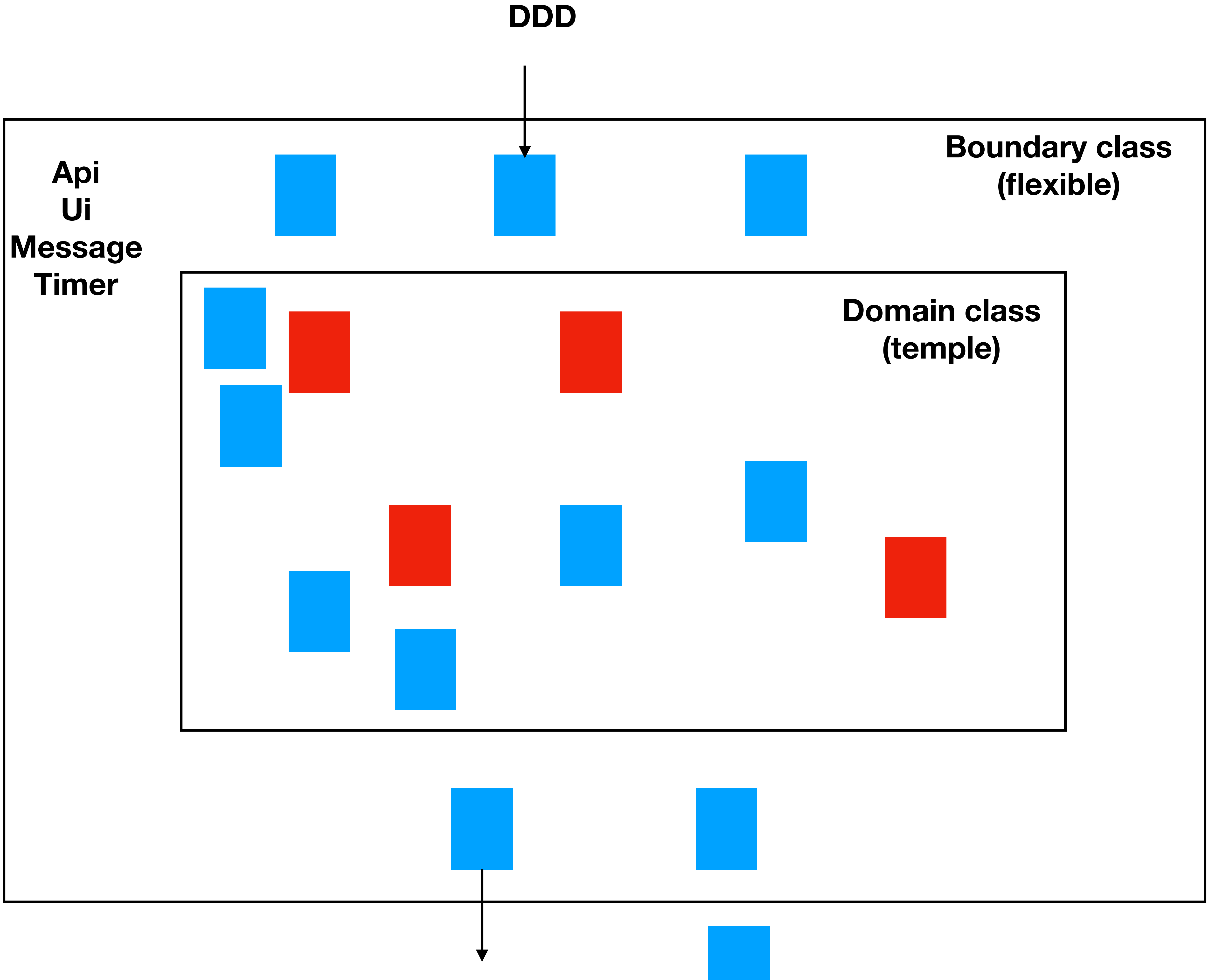
Guid IssueId

Guid UserId

IssueLabel (value obj)

Guid **IssueId**

Guid **LabelId**



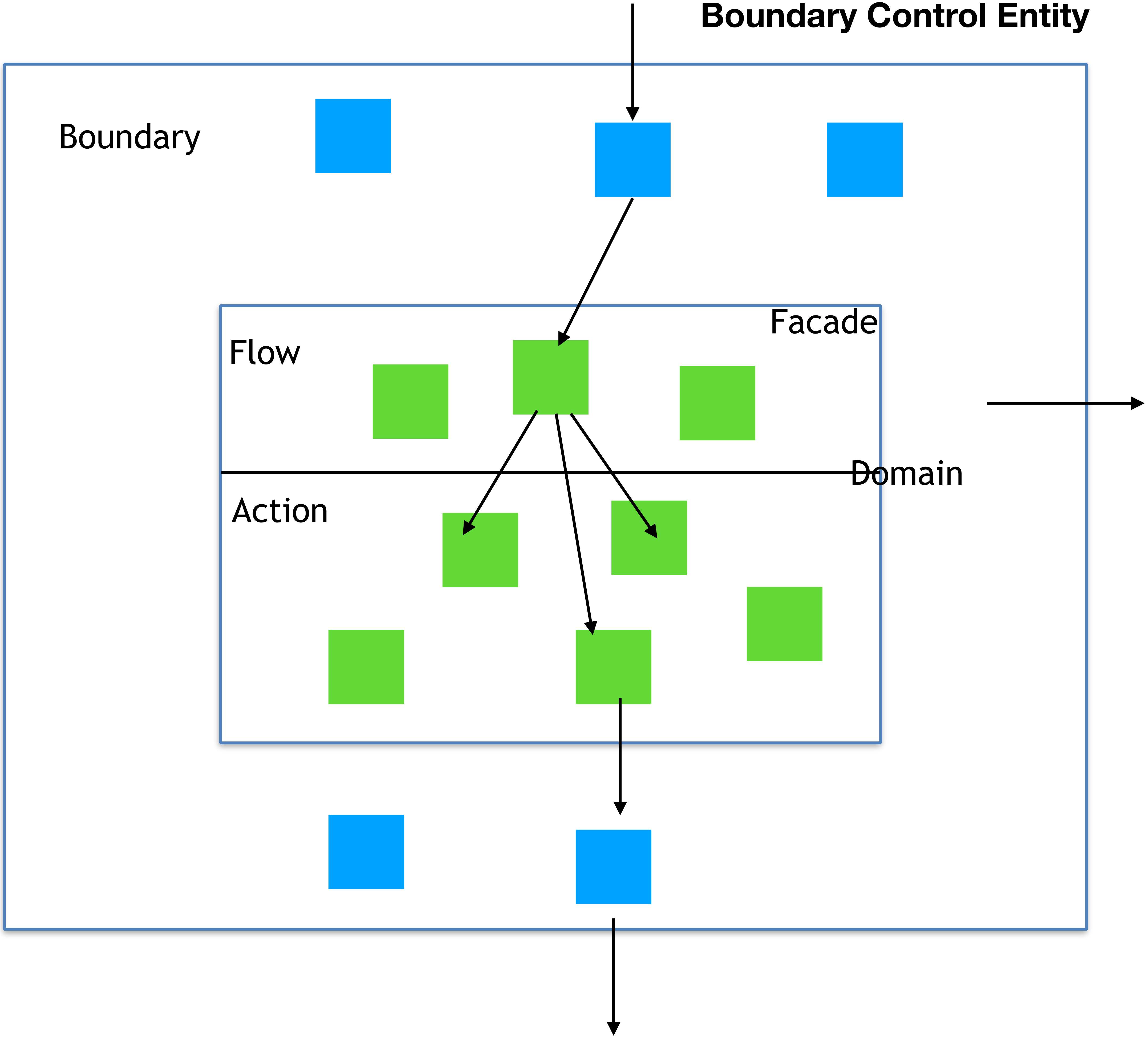
DDD

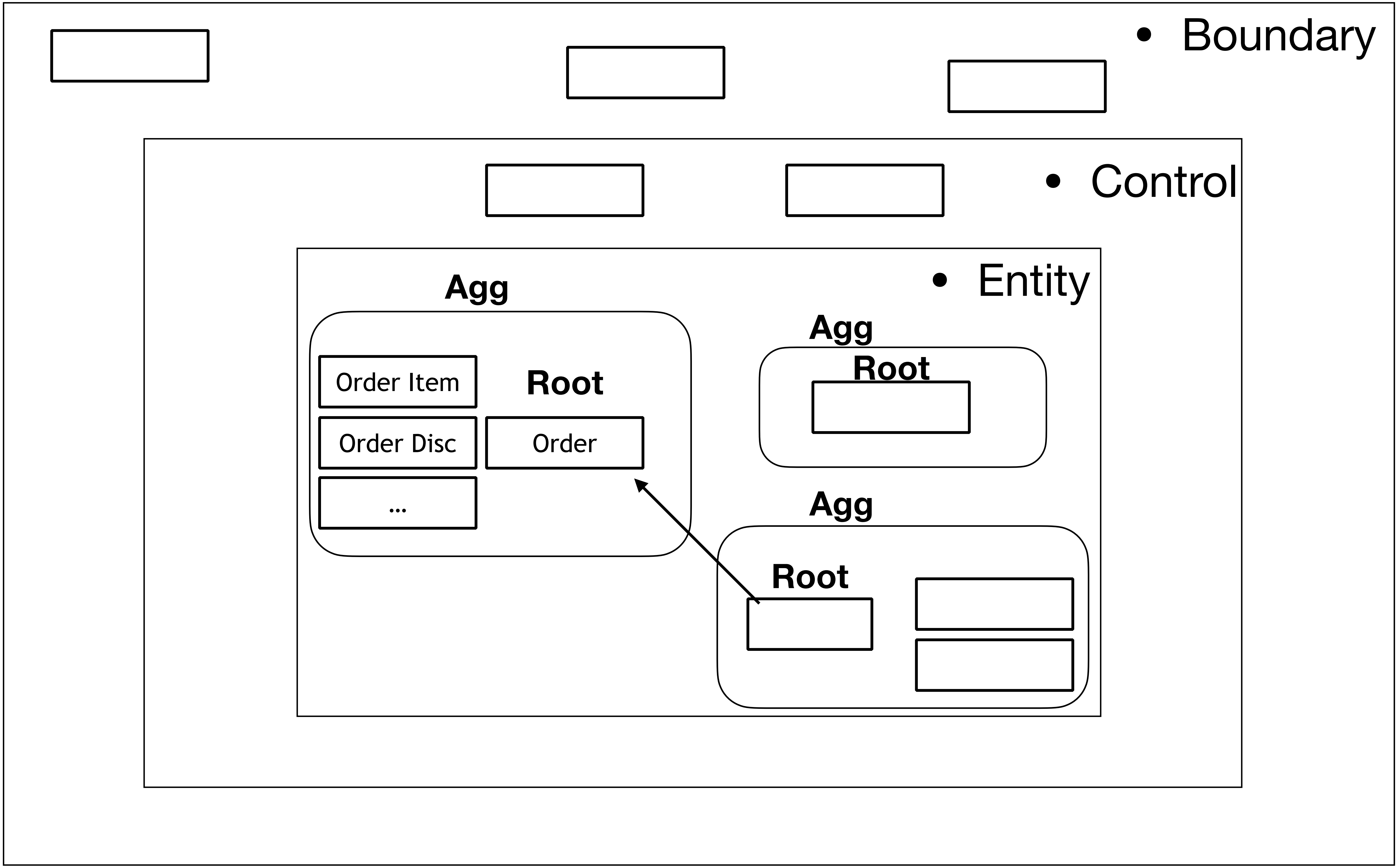
Api
Ui
Message
Timer

Boundary class
(flexible)

Domain class
(temple)

Boundary Control Entity





- Boundary
- Control
- Entity

Agg

Order Item

Root

Order Disc

Order

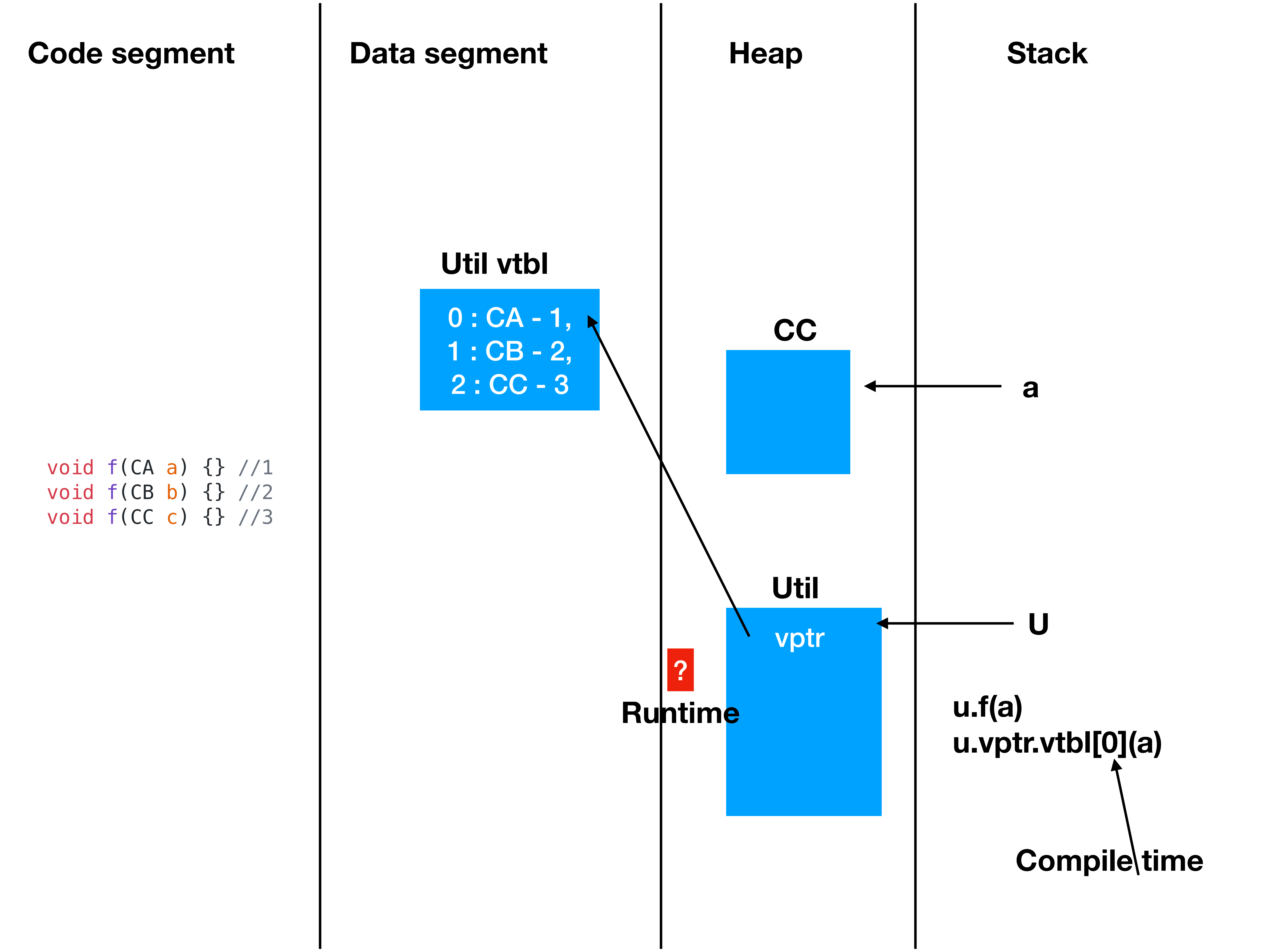
...

Agg

Root

Agg

Root



Code segment

Data segment

Heap

Stack

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

Util vtbl

0 : CA - 1,
1 : CB - 2,
2 : CC - 3

CC

a

Util

vptr

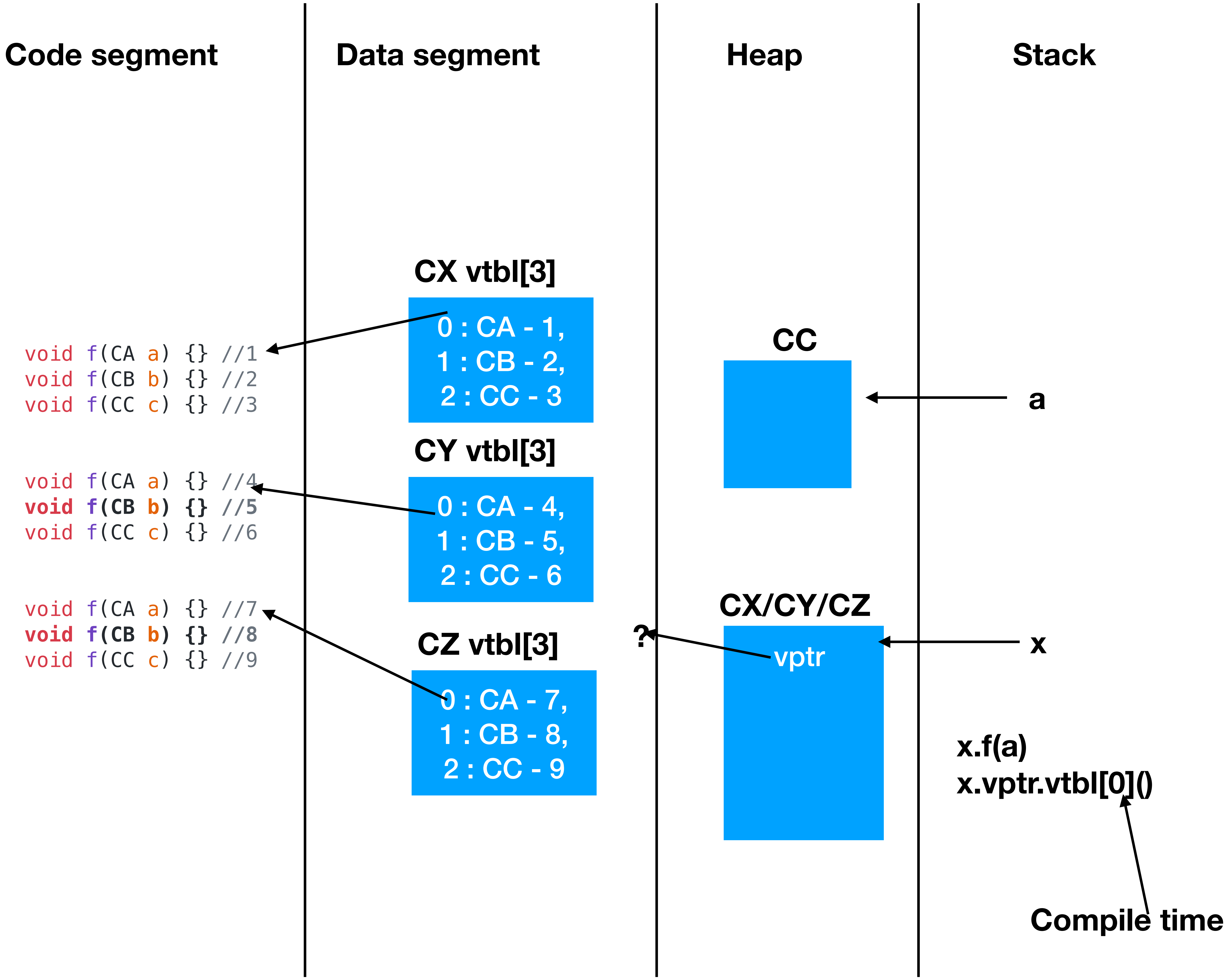
U

?

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]

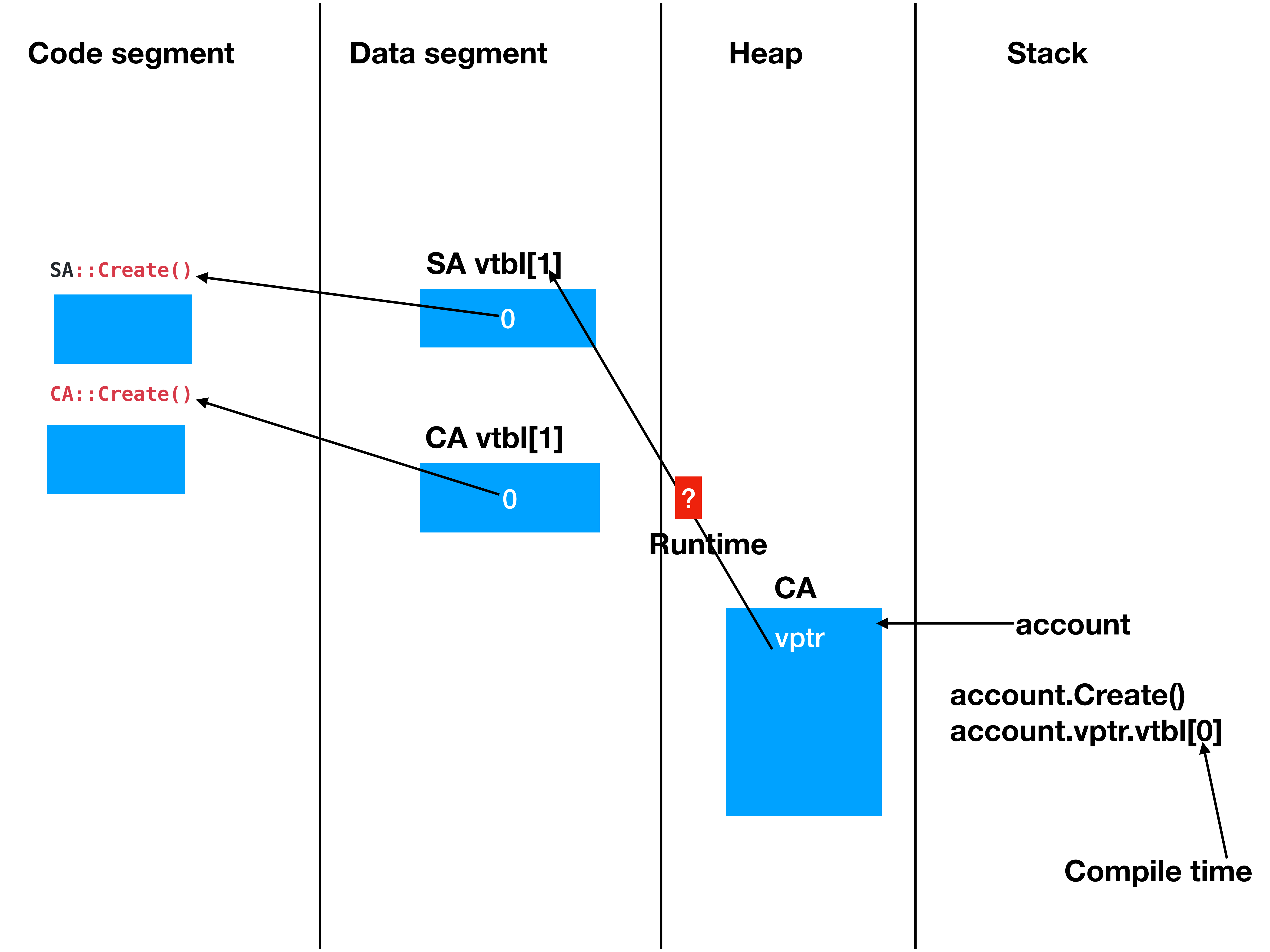
0 : CA - 4,
1 : CB - 5,
2 : CC - 6

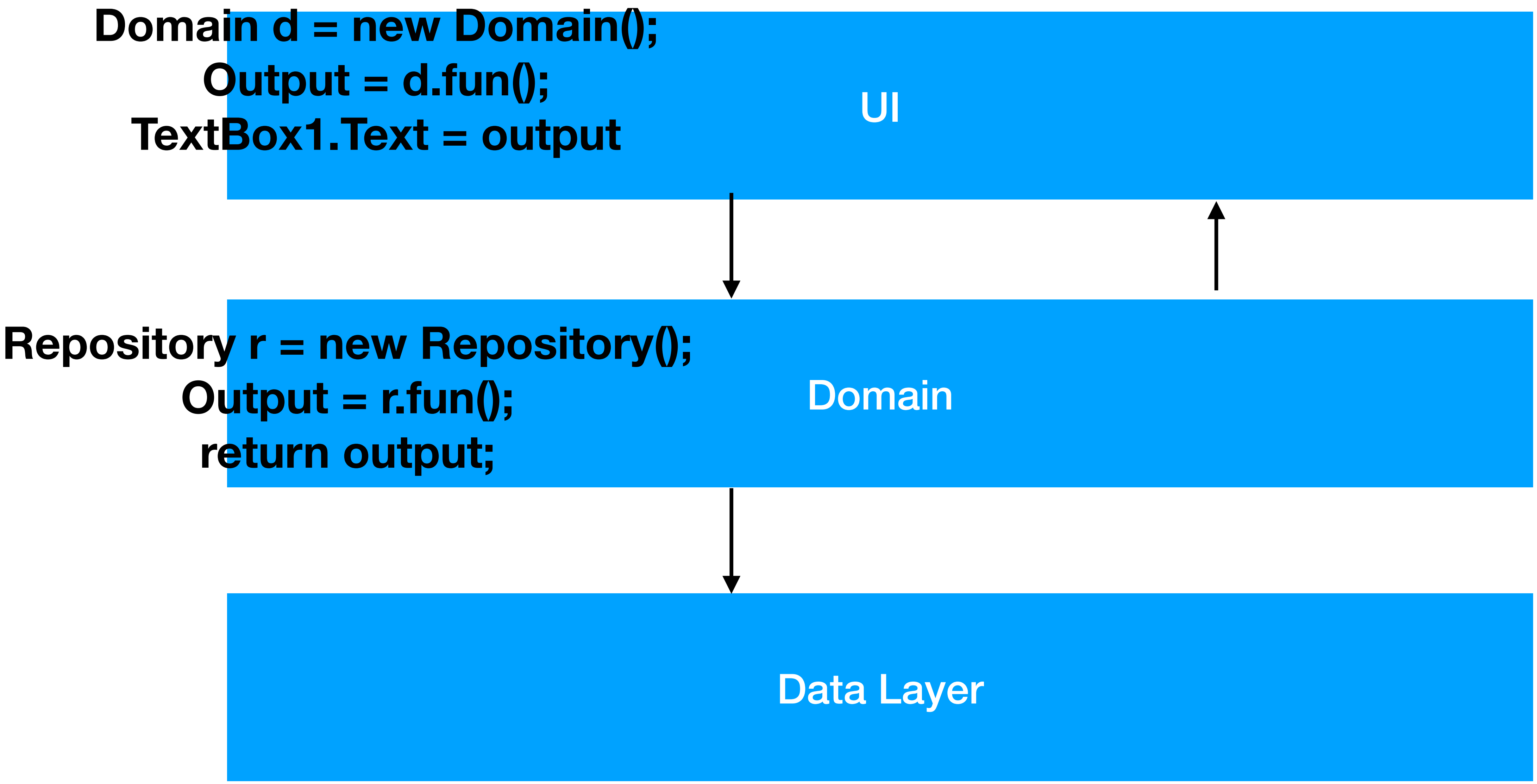
CZ vtbl[3]

0 : CA - 7,
1 : CB - 8,
2 : CC - 9

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

Compile time


Code segment
Data segment
Heap
Stack
SA::Create()
CA::Create()
SA vtbl[1]
0
CA vtbl[1]
0
?
Runtime
CA
vptr
account
account.Create()
account.vptr.vtbl[0]
Compile time



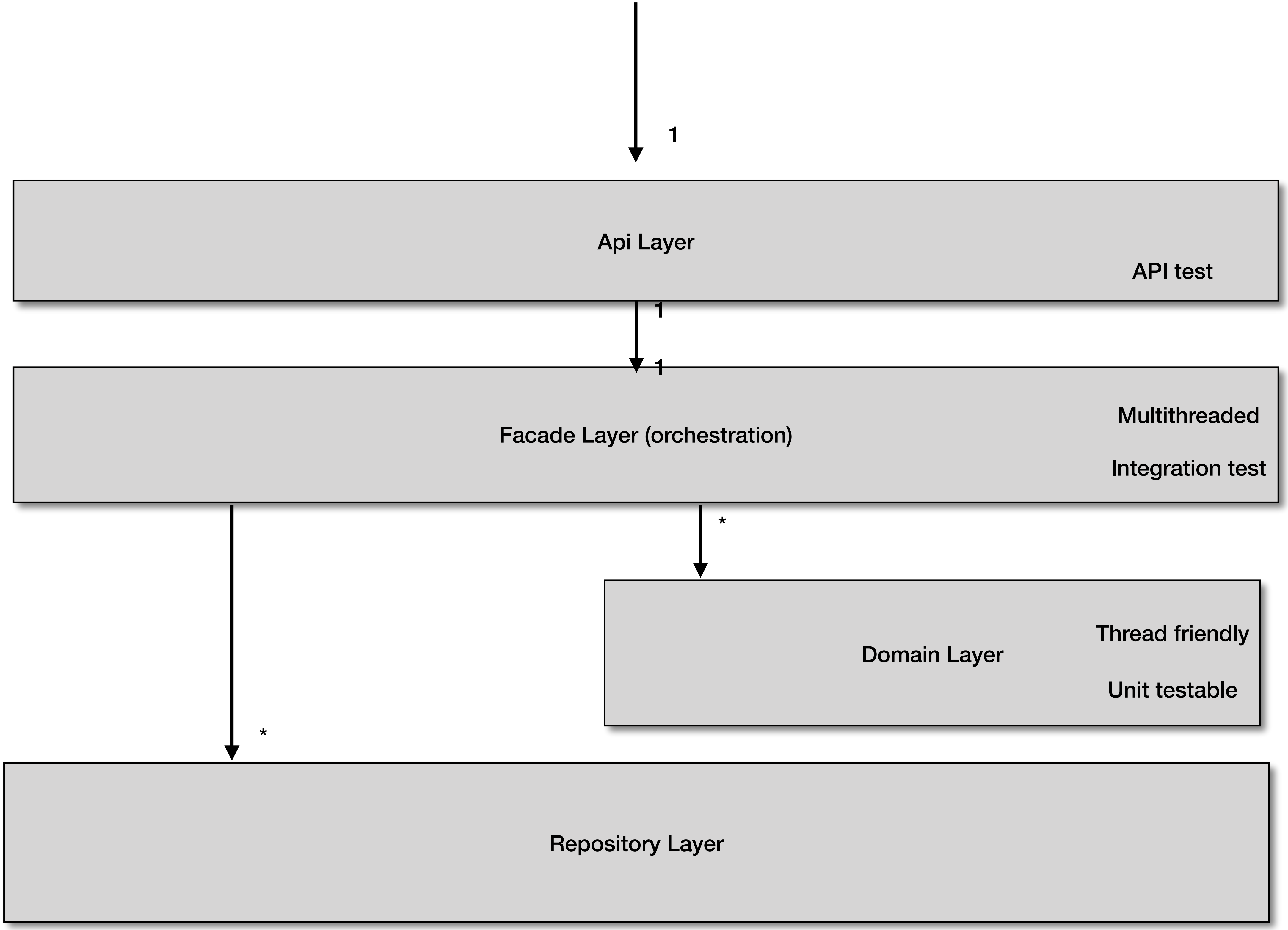
```
Domain d = new Domain();
Output = d.fun();
TextBox1.Text = output
```

UI

```
Repository r = new Repository();
Output = r.fun();
return output;
```

Domain

Data Layer



Api Layer

API test

Facade Layer (orchestration)

Multithreaded

Integration test

Domain Layer

Thread friendly

Unit testable

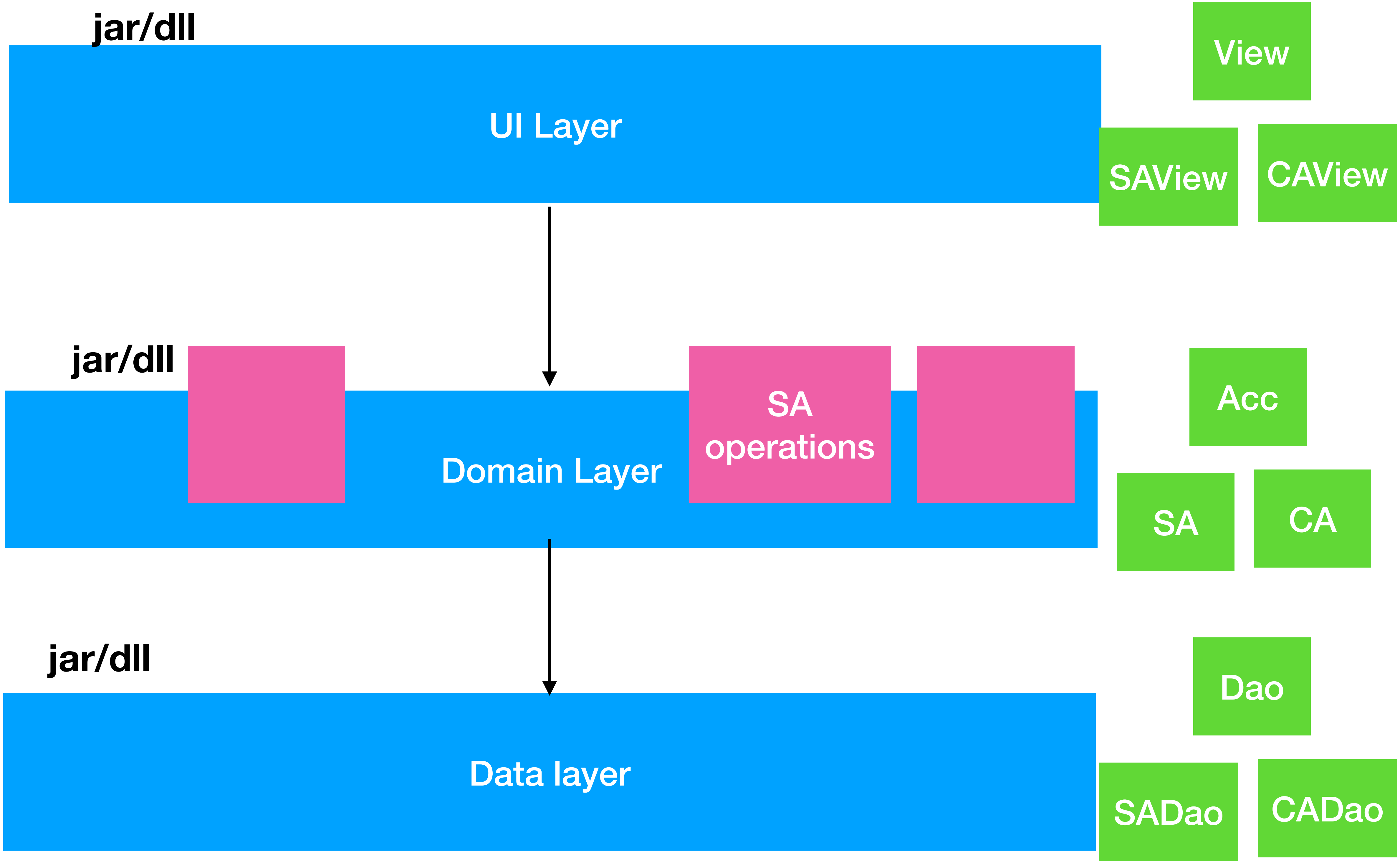
Repository Layer

Bad

- Sleep
- Suspend / Resume
- Changing Priority
- Abort
- Thread Local Storage

Good

- Synchronization constructs -> lock free constructs
- Cancelation design
- Thread pool
- async/ await



jar/dll

UI Layer

View

SAView

CAView

jar/dll

Domain Layer

SA operations

Acc

SA

CA

jar/dll

Data layer

Dao

SADao

CADao