» 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
- Multi thread

- Expectations
- Years of Exp
- Technology stack

https://forms.gle/nKn4mG1bVVX1U16x8

Good

- SRP (***)
- KISS
- LSP
- OCP (?)
- POLA

Bad

- Flag
- Type check
- Downcast
- Overloading family
- Magic string/int
- High cyclomatic complexity
-

- Open for adding new code without modifying existing code
- Open for add closed for modification
-

SOLID

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

Good

- SRP (***)
- Low coupling (**)
- Abstraction/ Interface/ Family / Implements
- LSP
- ISP
- Unit testability (*)
- OCP
- Program to an Interface

- DDD
  - Aggregates (unit)
- DRY
- KISS
- DIP

Bad

- Type checks
- Downcasting
- High Cyclomatic complexity
- Overloading family
- Flag
- Magic strings/numbers
- Extends / Inheritance
- Functional interface

- God Class
- Static methods
- * to * coupling
- bool/null/int for error
- Null / Nothing
-

SRP

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

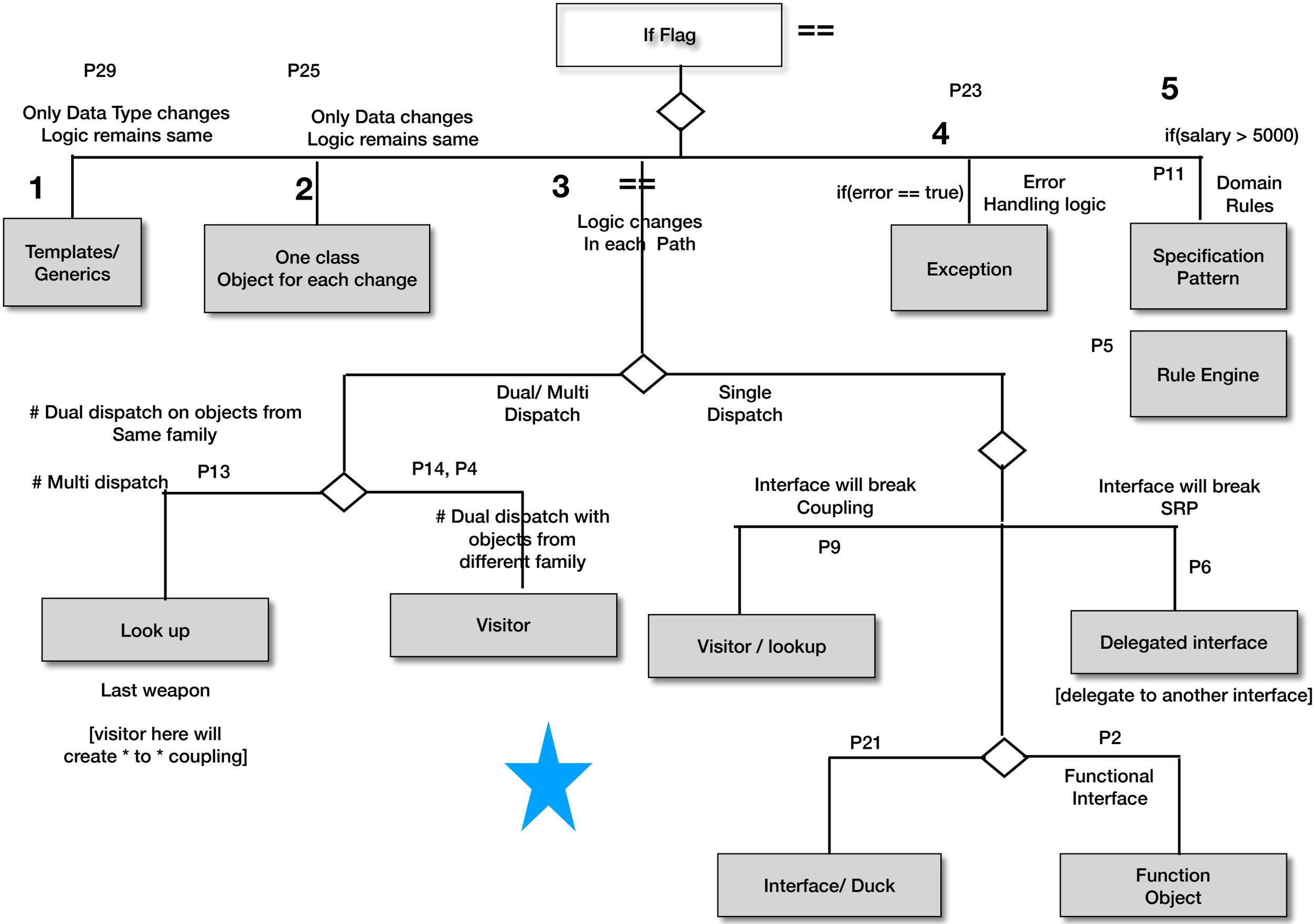
SOC

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

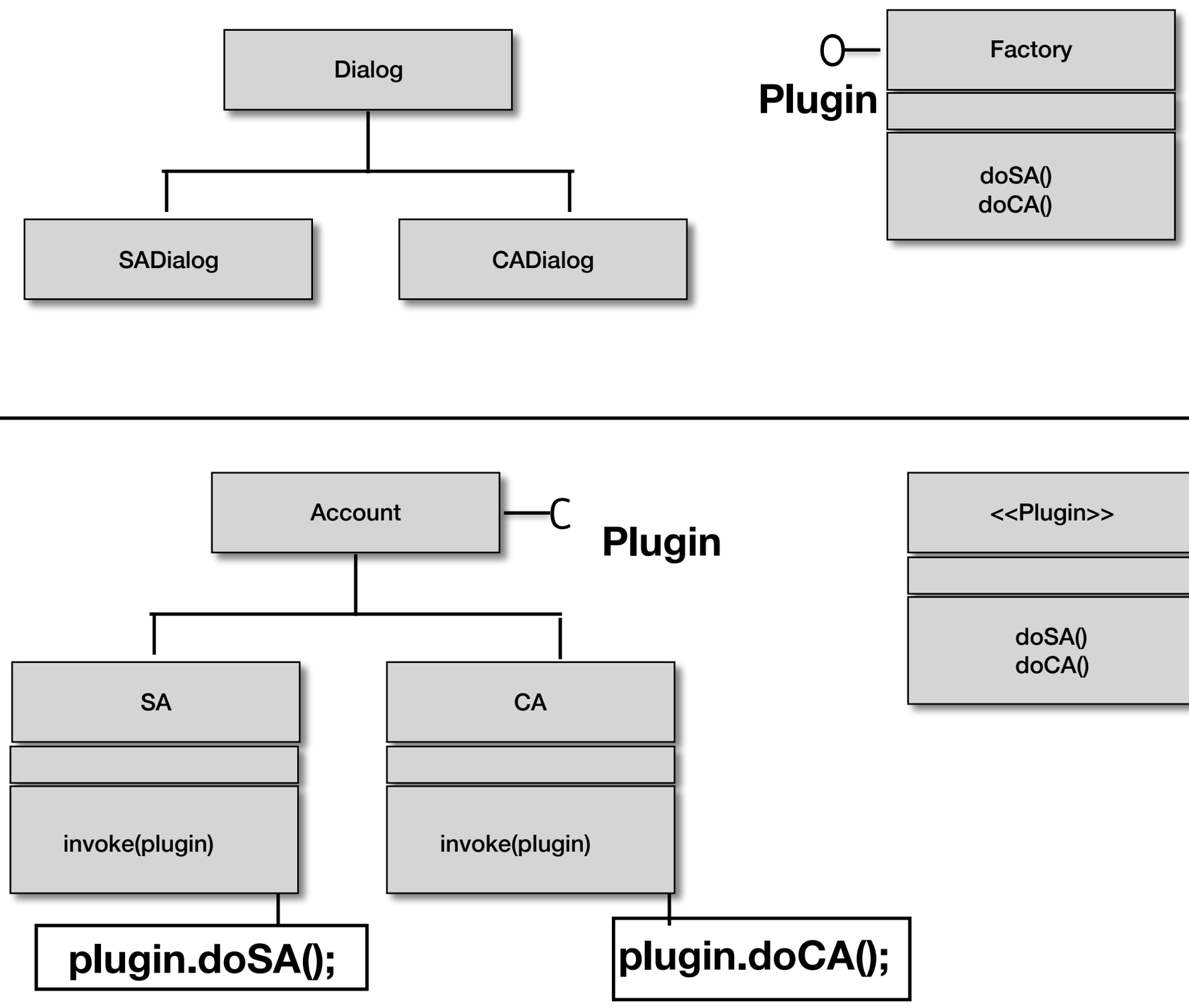
Size

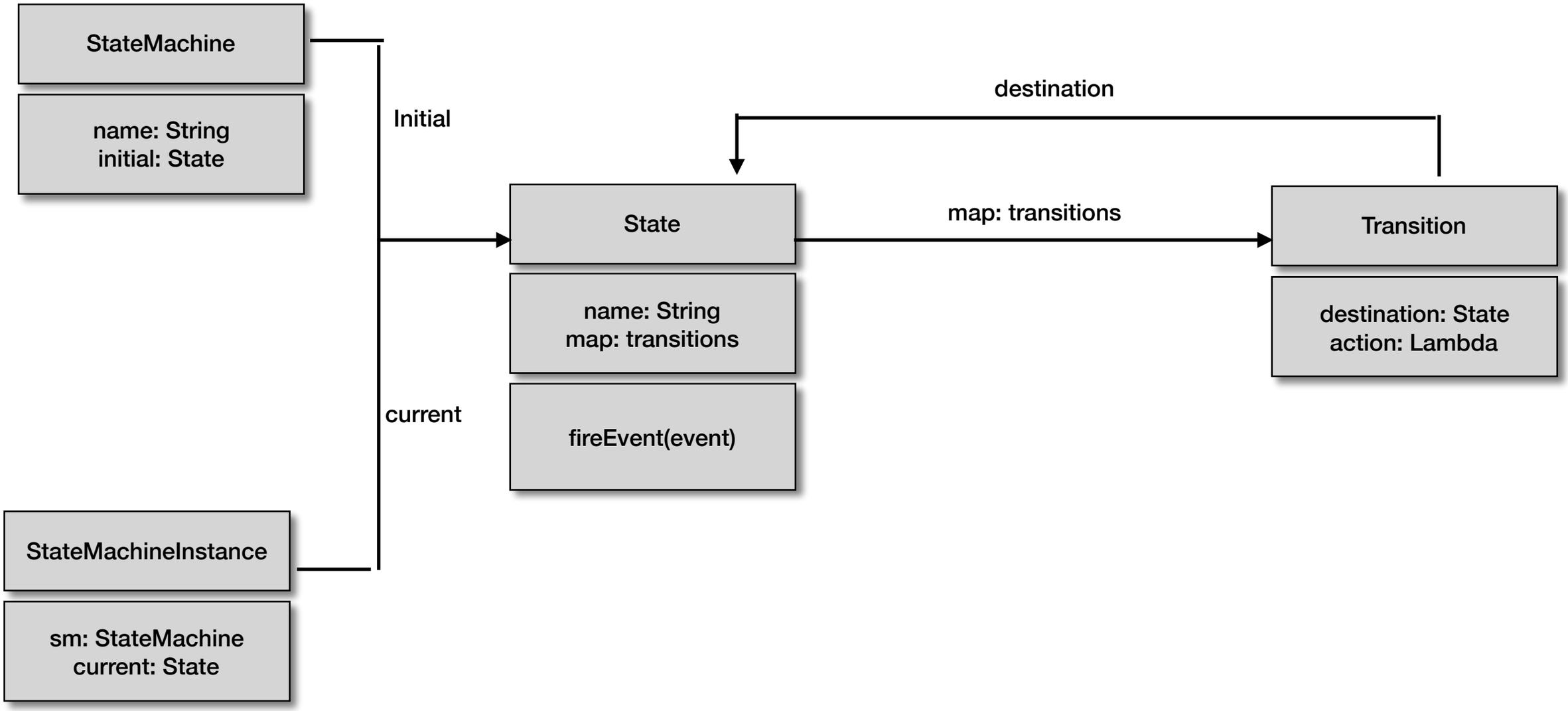
- Module
  - Max class : 25
- Class / Interface
  - Max public methods : ~12
  - Avg : ~4
- Fun
  - Max : fit screen
  - Avg : 10 lines

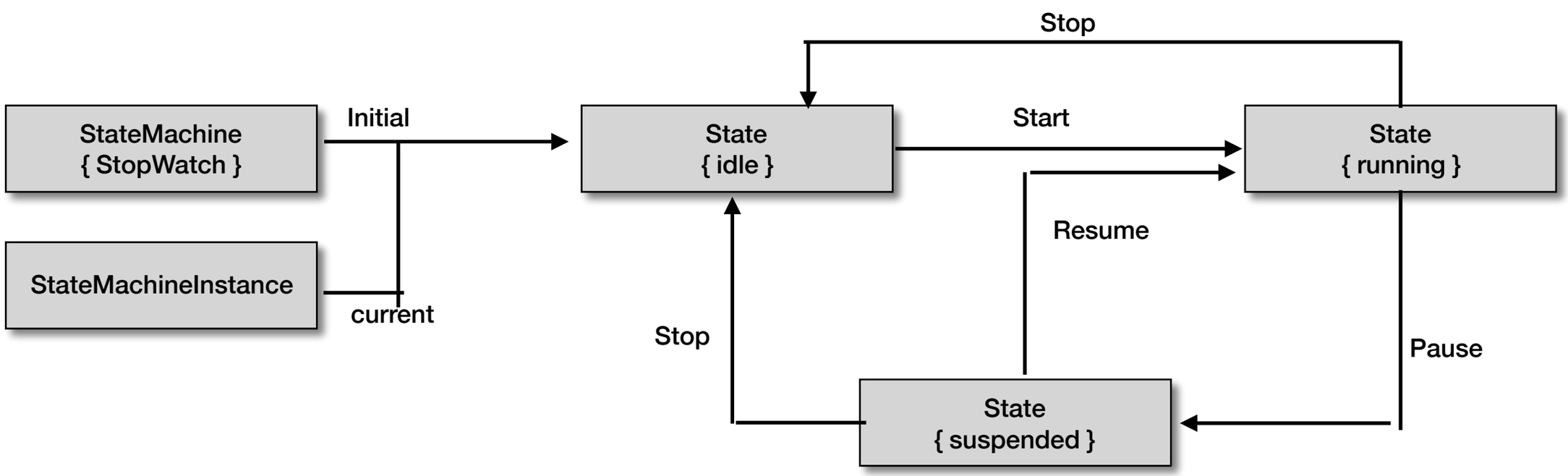
500 lines of code	*(1) - dense 10 fun 50 lines each*	*(2) - sparse 50 fun 10 lines each*
Perf	~	~
Easy to Name		*
Unit test		*
Readability		*
Agility to change		*
Reusability		*


If Flag
==
P29
Only Data Type changes
Logic remains same
1
Templates/
Generics
P25
Only Data changes
Logic remains same
2
One class
Object for each change
3
==
Logic changes
In each Path
P23
4
if(error == true)
Error
Handling logic
Exception
5
if(salary > 5000)
P11
Domain
Rules
Specification
Pattern
P5
Rule Engine
Dual/ Multi
Dispatch
Single
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
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

account.invoke(factory);

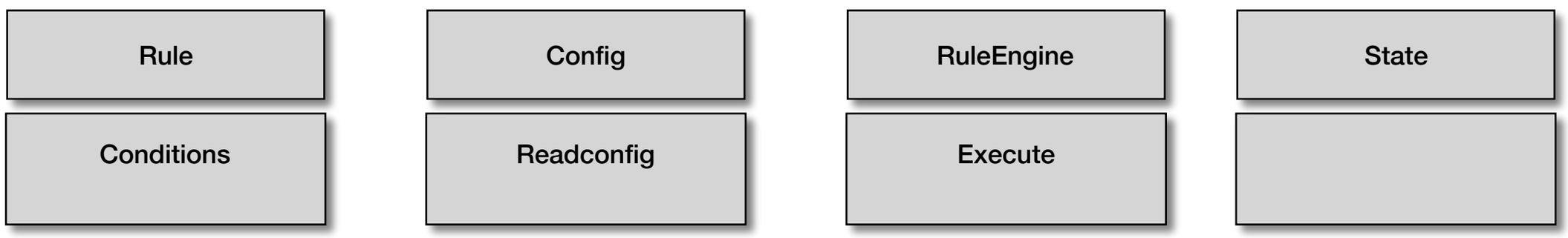



StateMachine
name: String
initial: State
Initial
destination
State
name: String
map: transitions
fireEvent(event)
map: transitions
Transition
destination: State
action: Lambda
current
StateMachineInstance
sm: StateMachine
current: State


Stop
StateMachine
{ StopWatch }
Initial
State
{ idle }
Start
State
{ running }
Resume
StateMachineInstance
current
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) => logic
- If(salary > 5000 & age < 21) => logic
- if(salary > 5000 & (age > 35 || location == “NY”) => logic



Rule

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

```
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

```
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) => 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);
```

- isSalaryHigh()
-

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

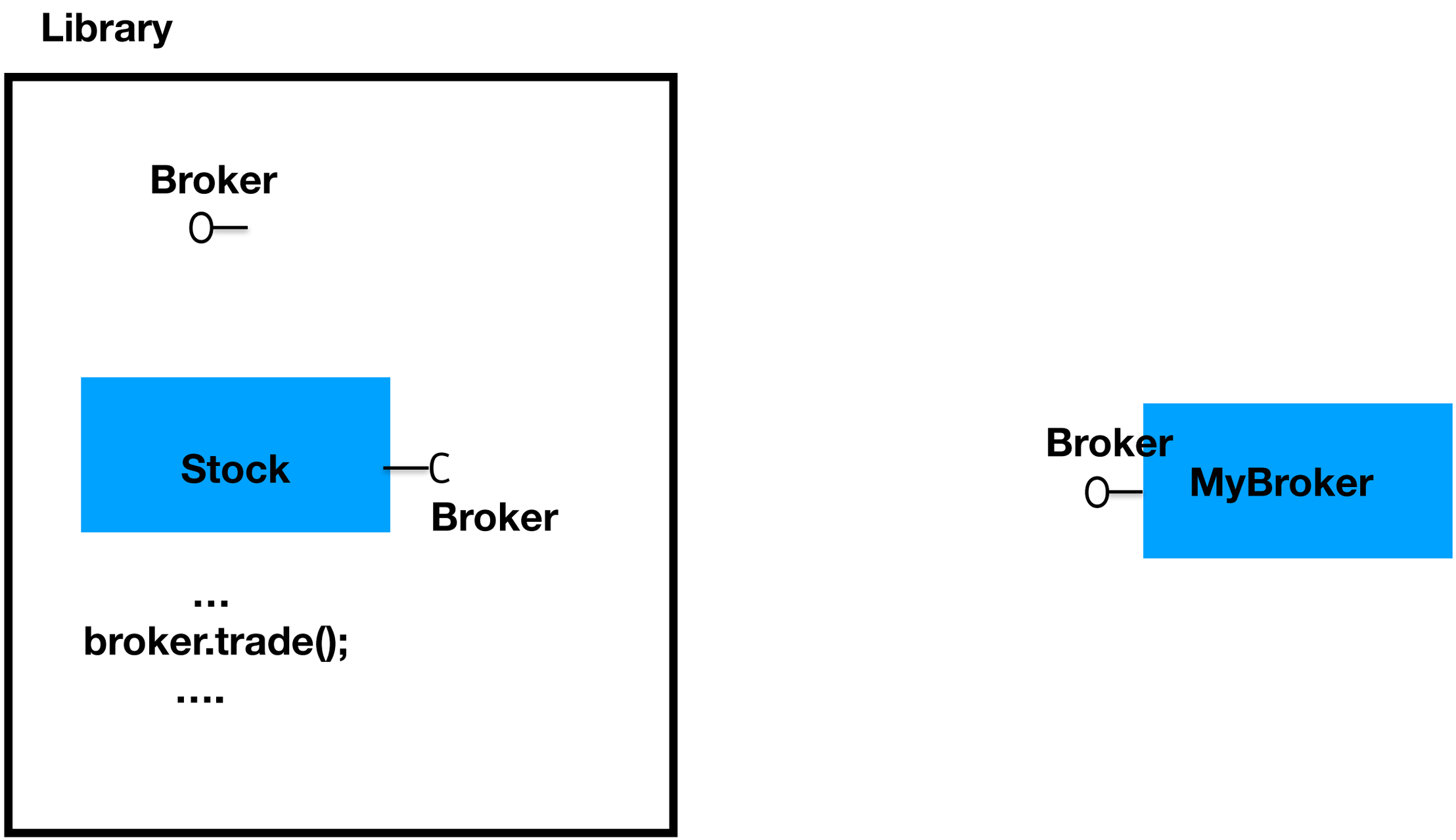
```
class Emp{
   Emp manager;
}
```

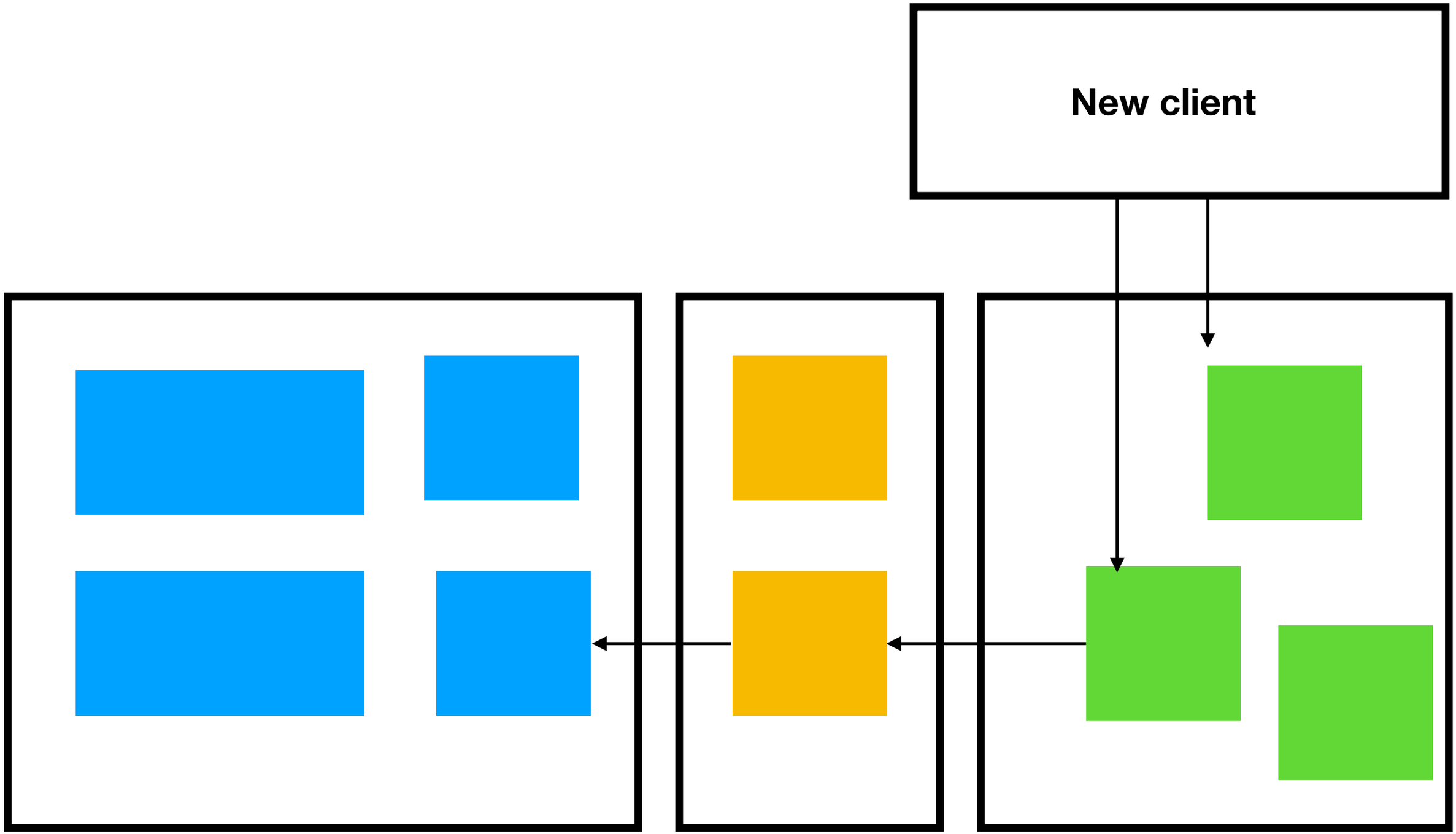
Graph /tree

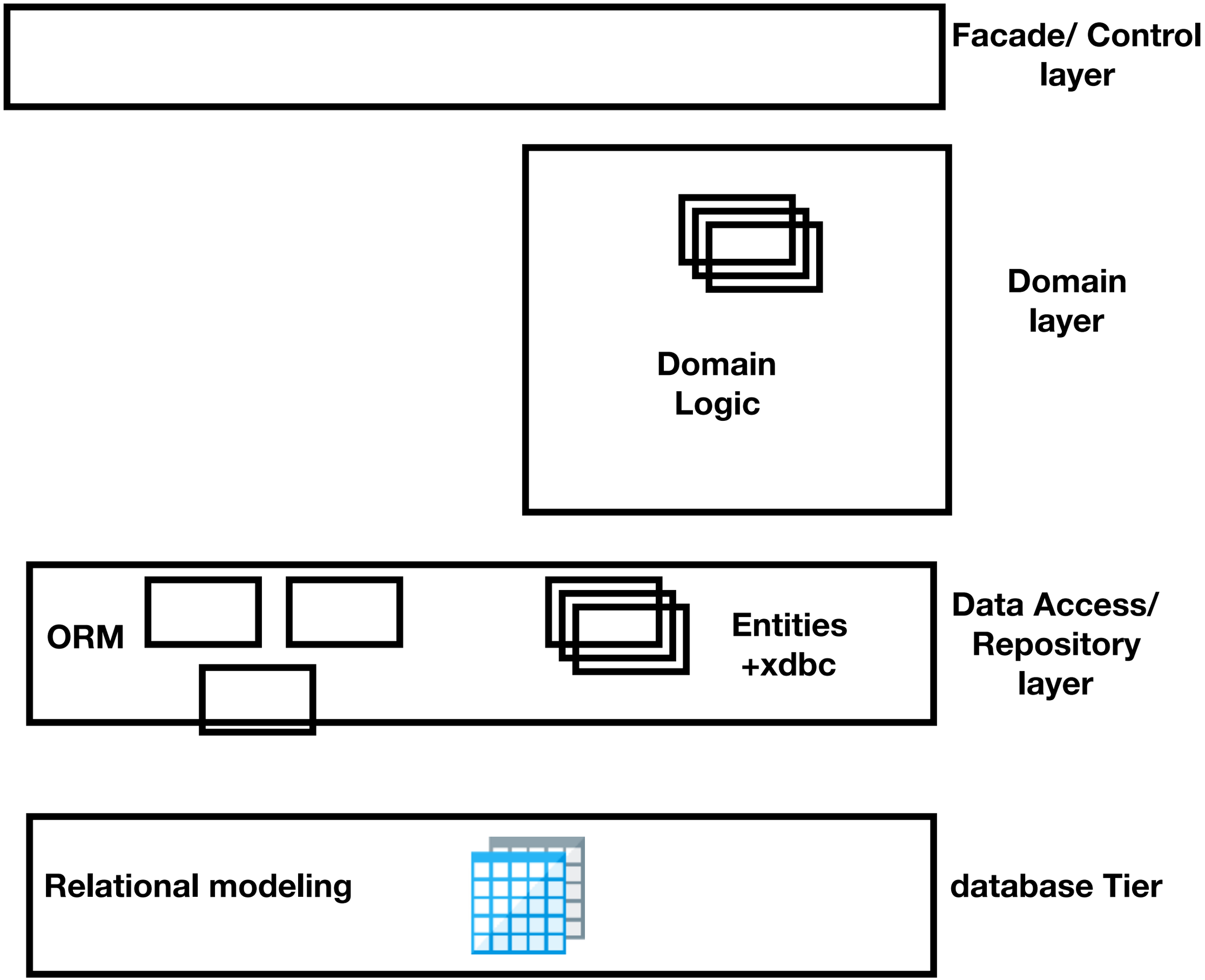
Linked list

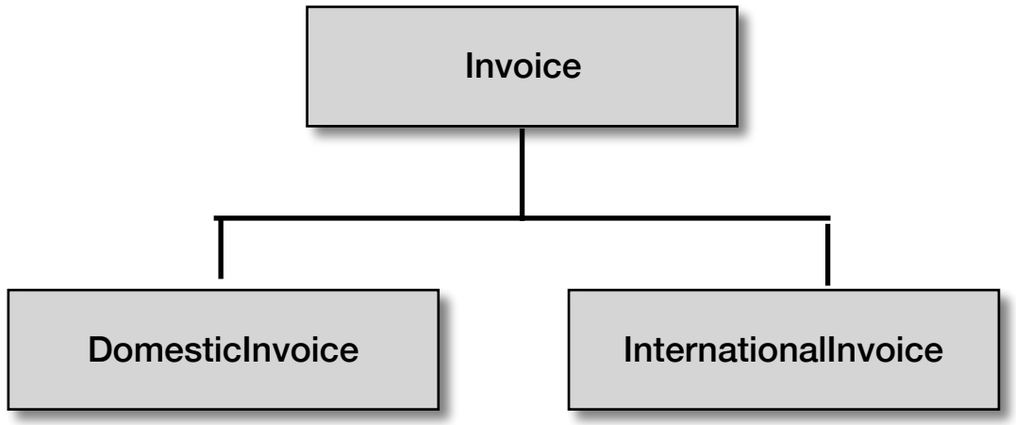
```
class Emp{
   Emp a;
   Emp b;
}
```

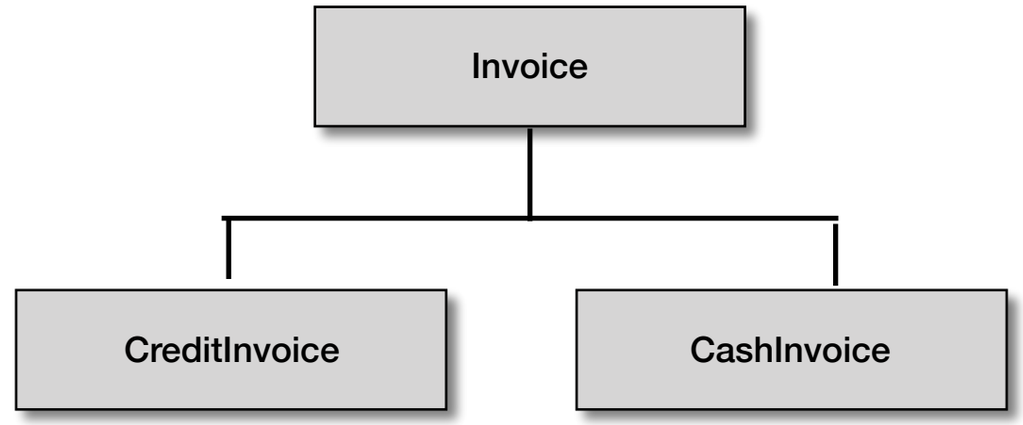
	1	2	3
Type of Coupling	**Method call**	**Instantiation**	**Deallocation**
Examples of coupling	Emp obj … obj.fun();	new Emp()	Emp obj … delete obj;
Approach for Low coupling	Abstraction # Interface typing * # Duck typing # Lamda	# DI * # factory	# smart pointers # virtual destructor
Xtreme Approach	#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

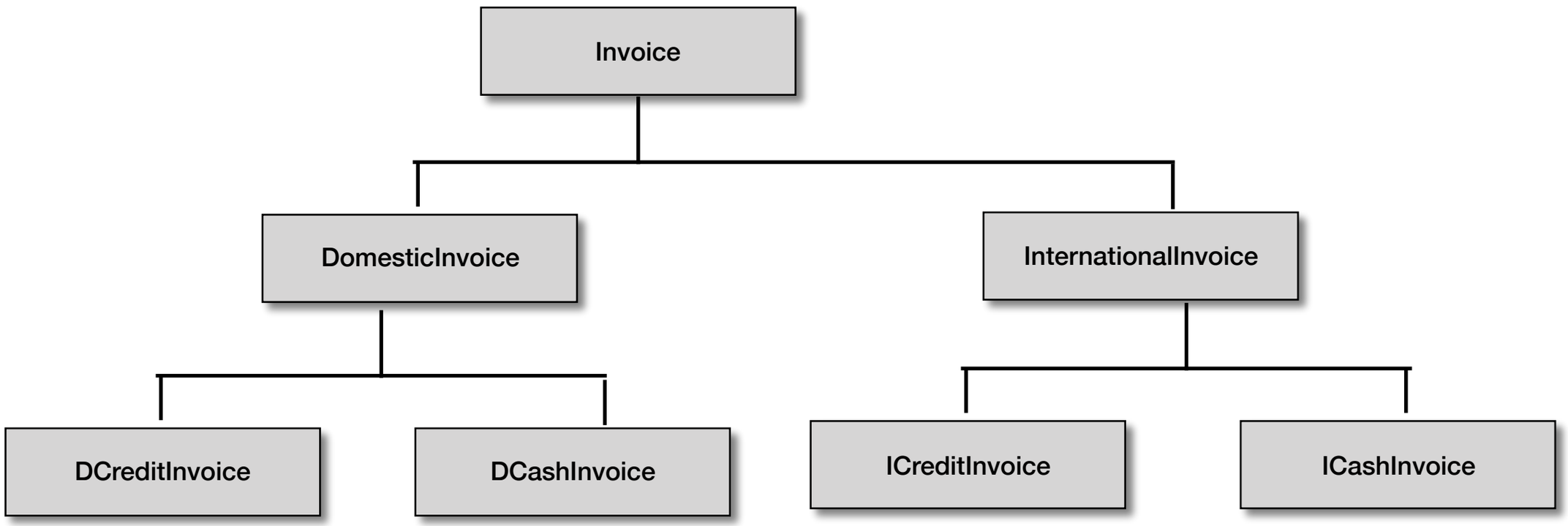

Library
Broker
Stock
Broker
...
broker.trade();
....
Broker
MyBroker

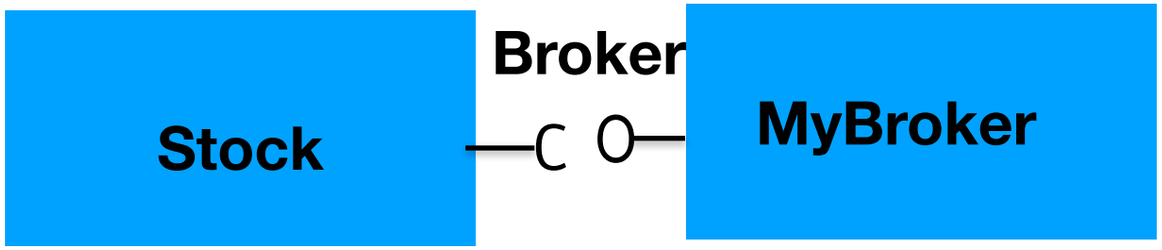

New client

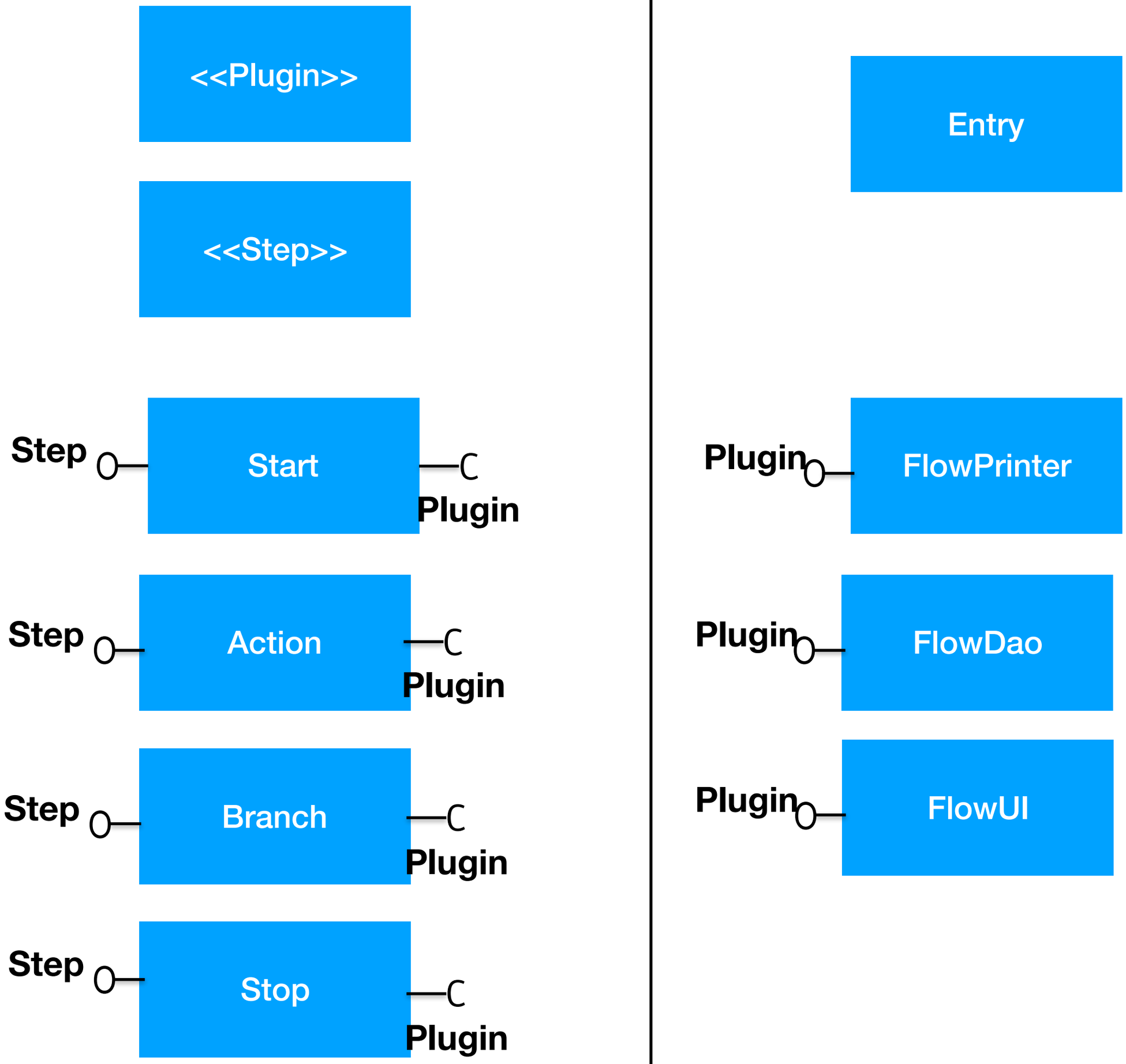

Facade/ Control layer
Domain Logic
Domain layer
ORM
Entities +xdbc
Data Access/ Repository layer
Relational modeling
database Tier

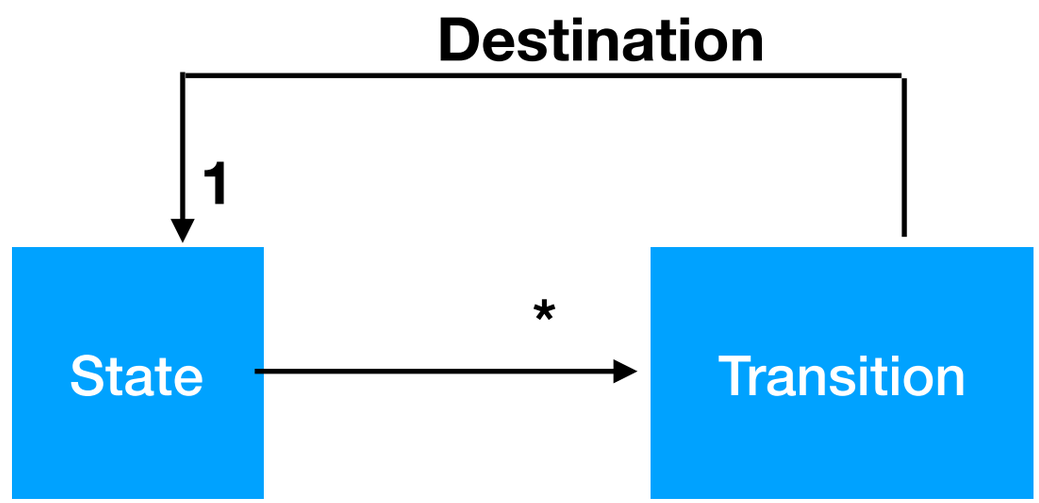

Invoice
DomesticInvoice
InternationalInvoice

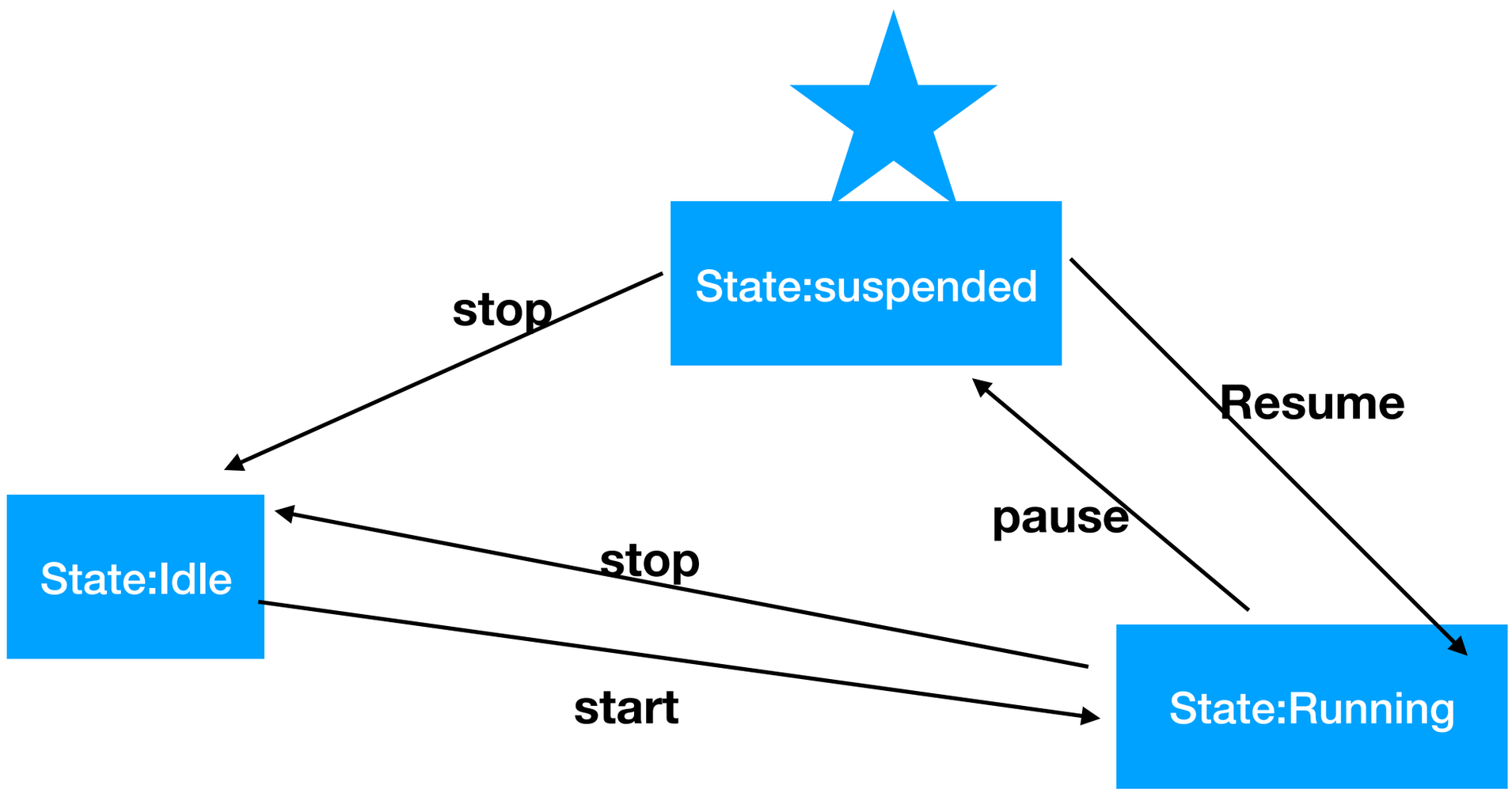

Invoice
CreditInvoice
CashInvoice


Invoice
DomesticInvoice
InternationalInvoice
DCreditInvoice
DCashInvoice
ICreditInvoice
ICashInvoice

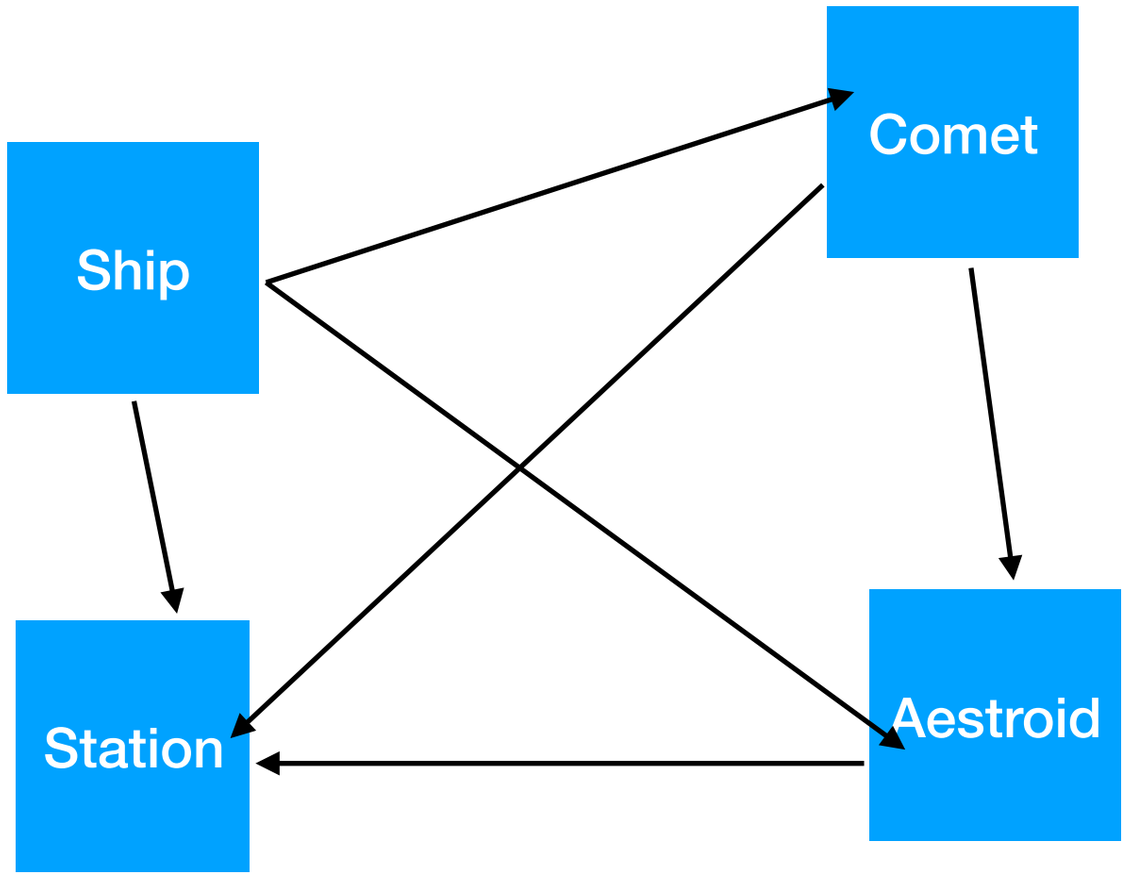

Stock
Broker
MyBroker

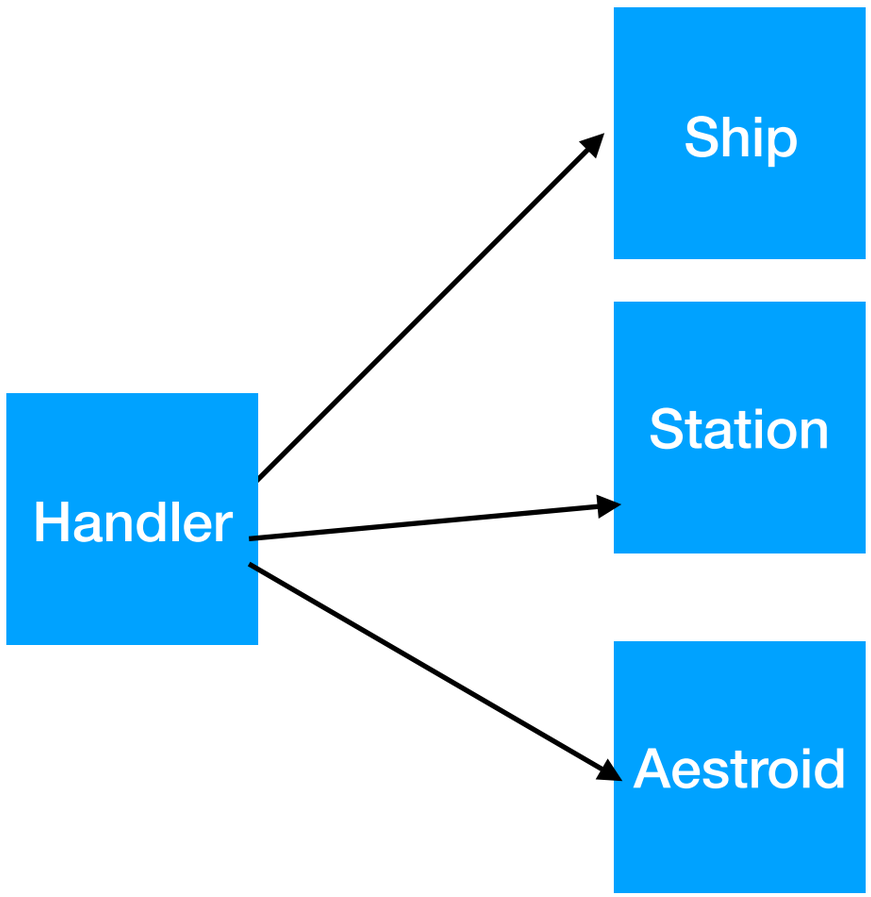

<<Plugin>>
<<Step>>
Entry
Step
Start
Plugin
Plugin
FlowPrinter
Step
Action
Plugin
Plugin
FlowDao
Step
Branch
Plugin
Plugin
FlowUI
Step
Stop
Plugin


Destination
1
State
*
Transition


State:suspended
stop
Resume
pause
State:Idle
stop
start
State:Running

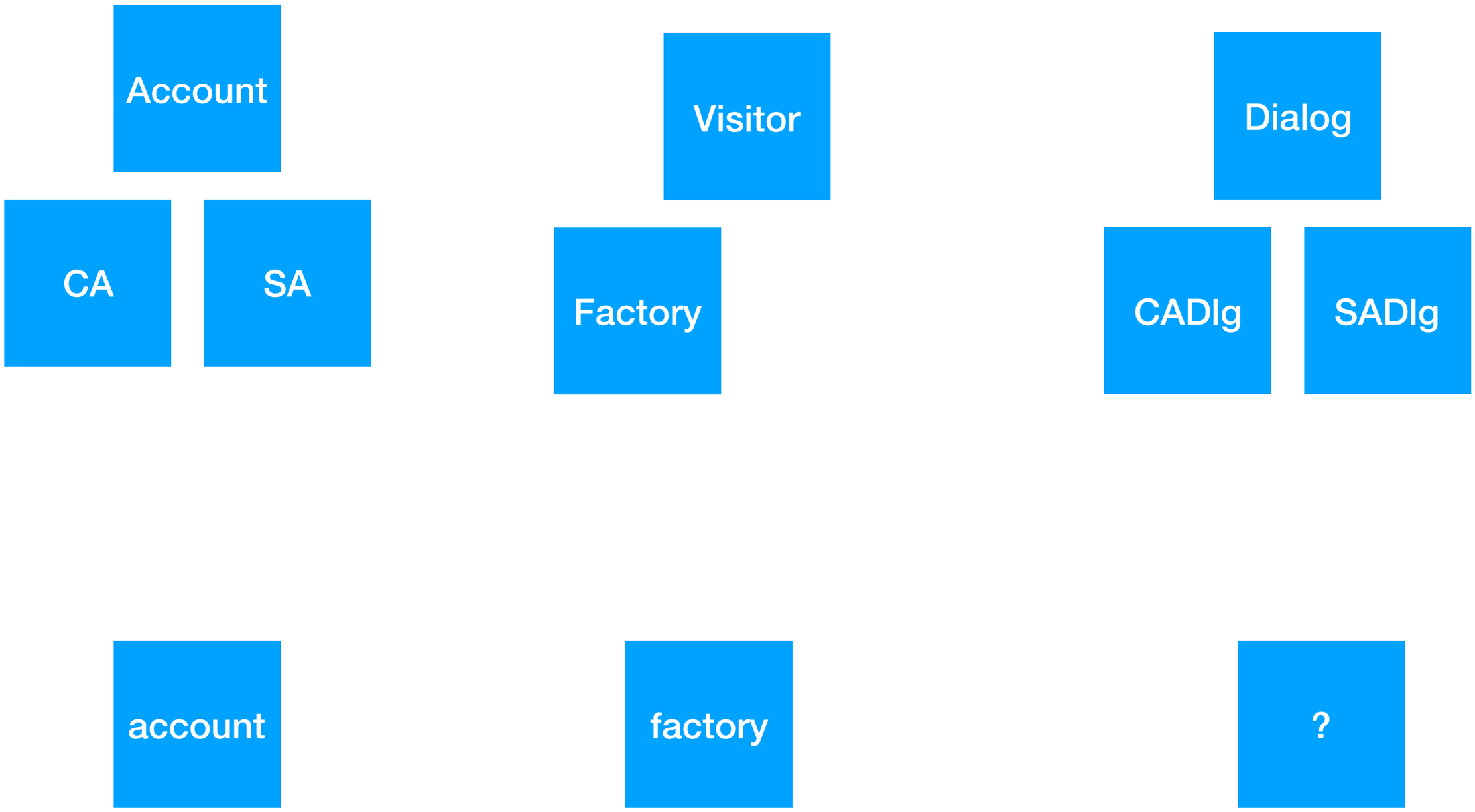
	5A (static)	3 (dynamic)
OCP		***
KISS	***	

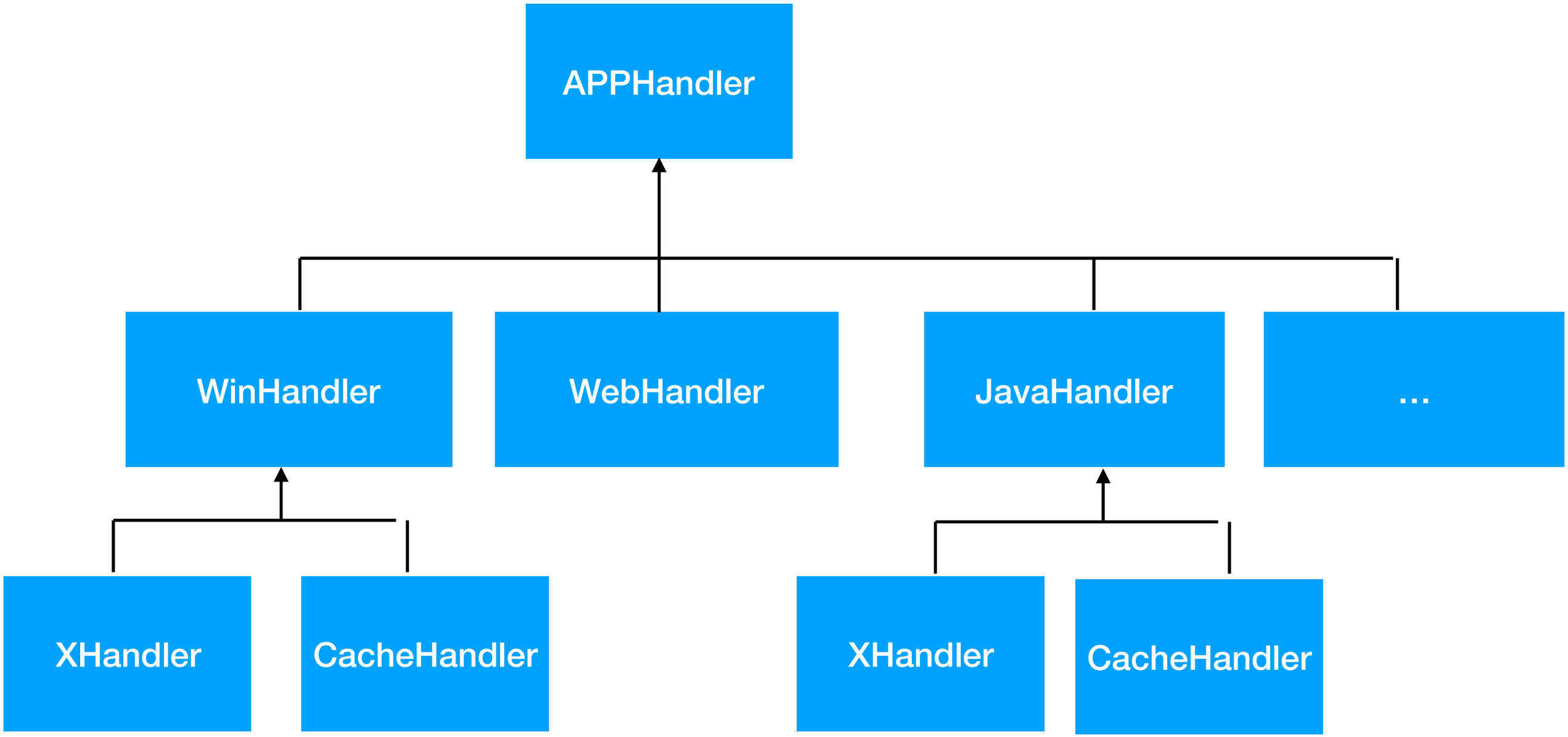

Comet
Ship
Station
Aestroid

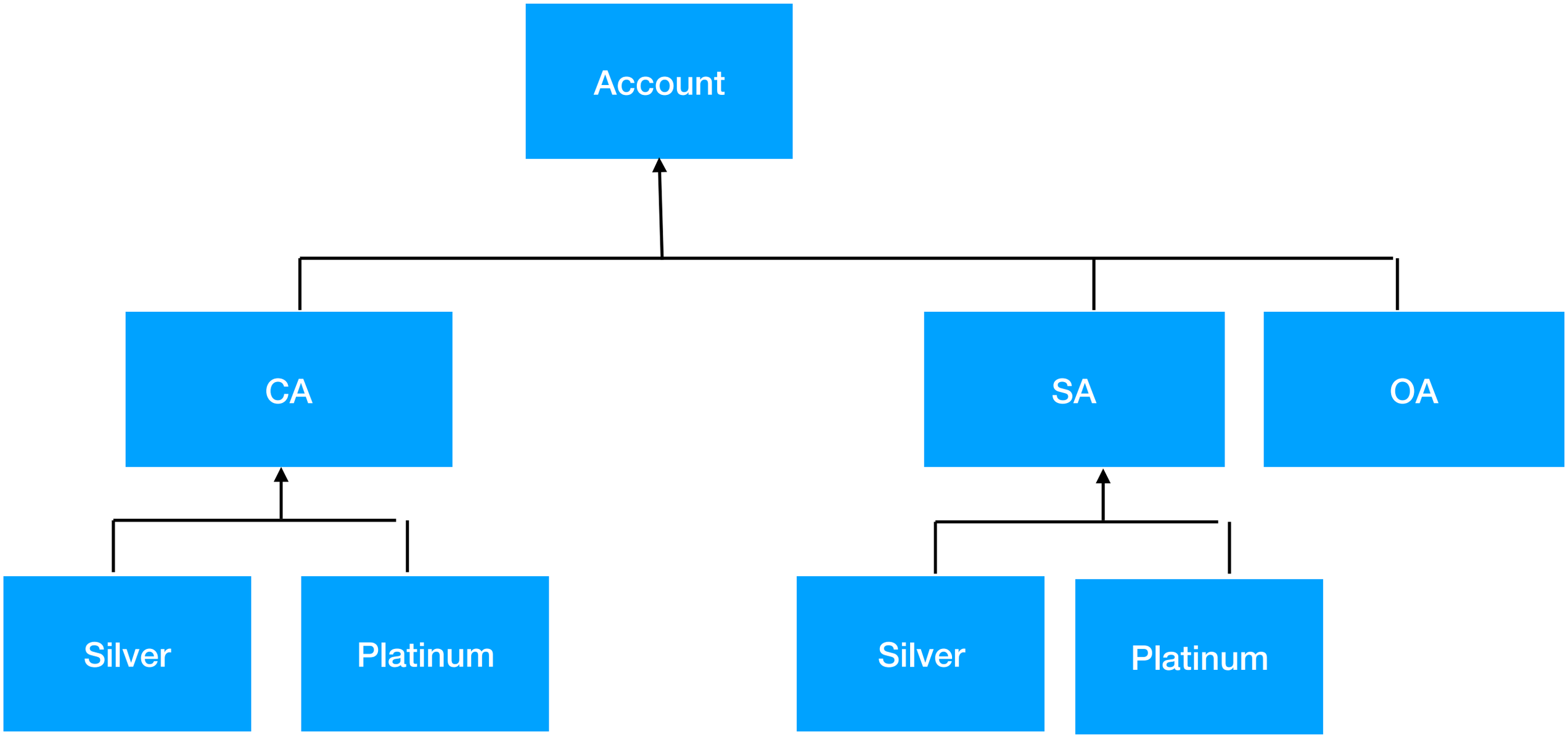

Ship
Handler
Station
Aestroid

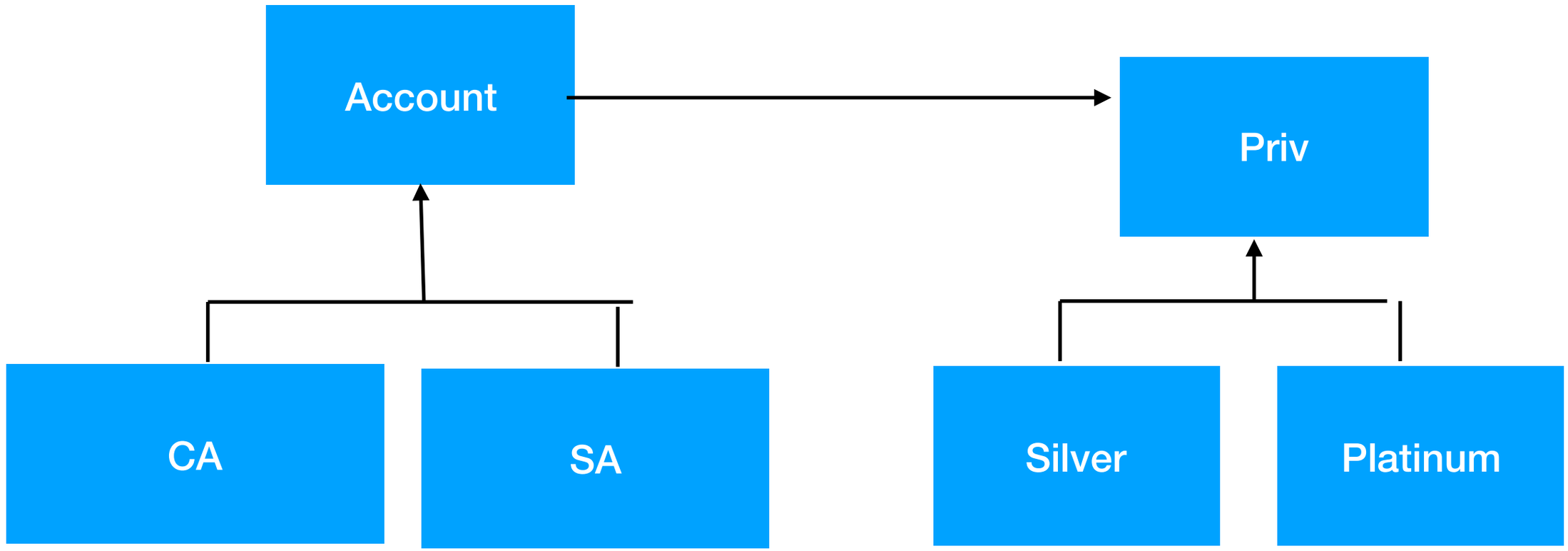
Factory

- 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


Account
CA
SA
OA
Silver
Platinum
Silver
Platinum


Account
Priv
CA
SA
Silver
Platinum

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

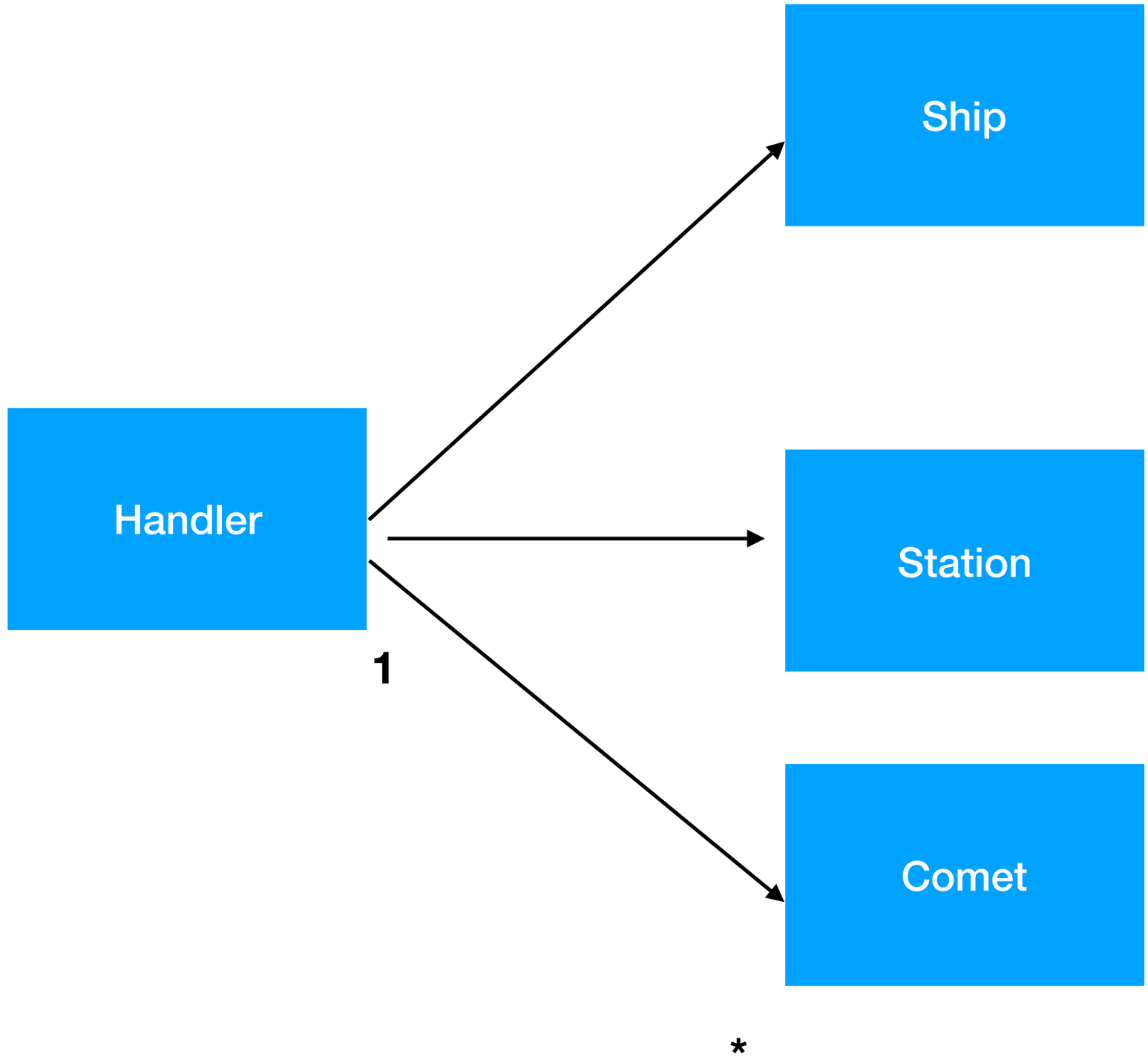
```
class SADialog : Dialog {}
```

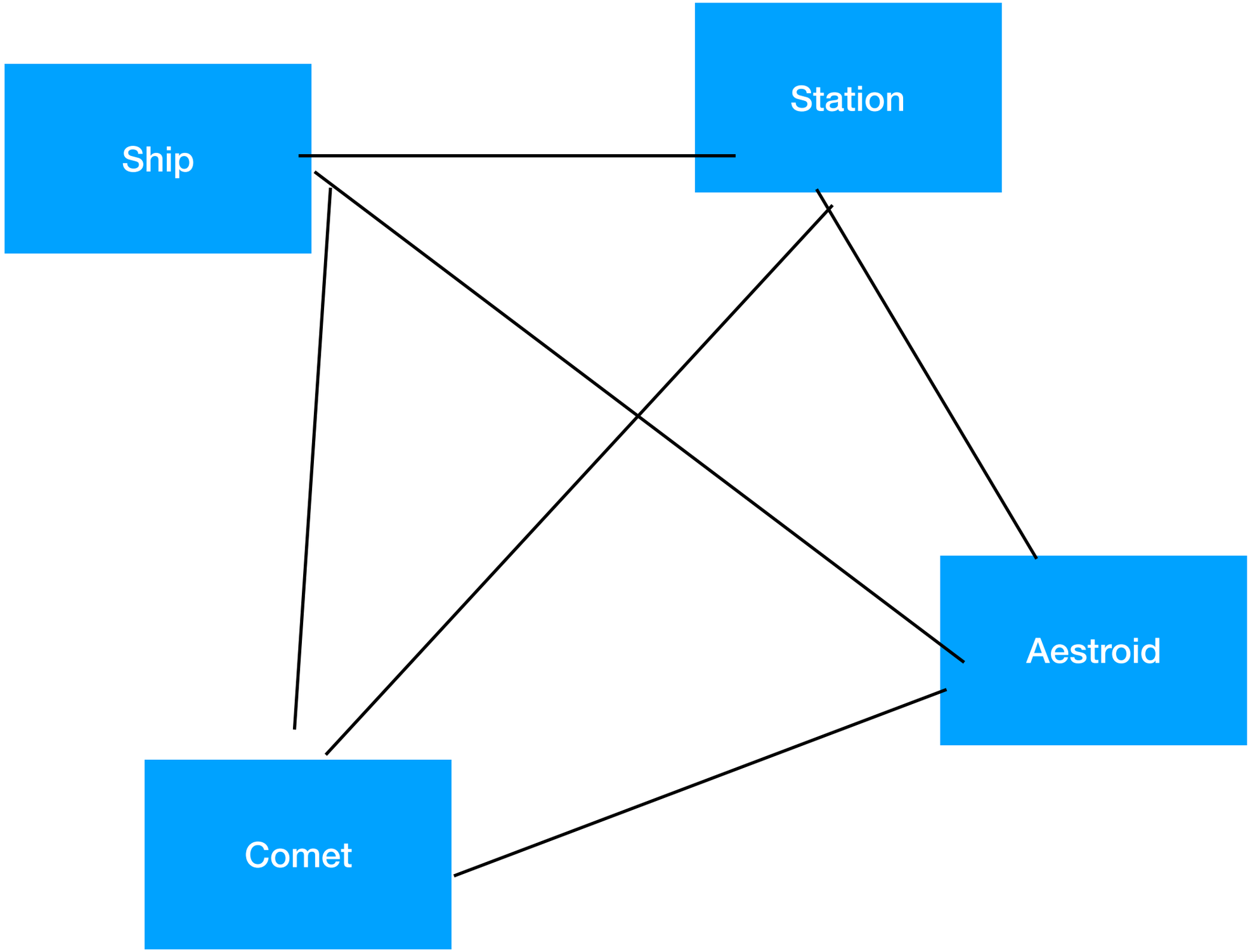
```
class CADialog : Dialog {}
```

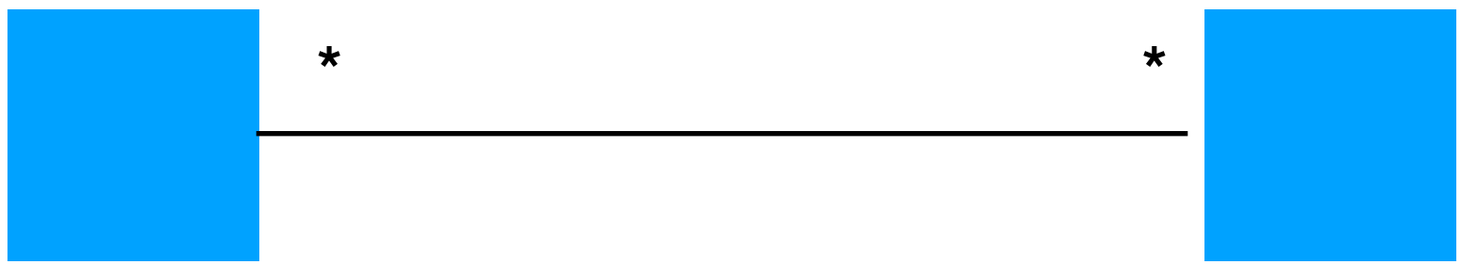
```
public class DialogFactory{
        public Dialog CreateUI(Account account){
                Dialog dlg=null;

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

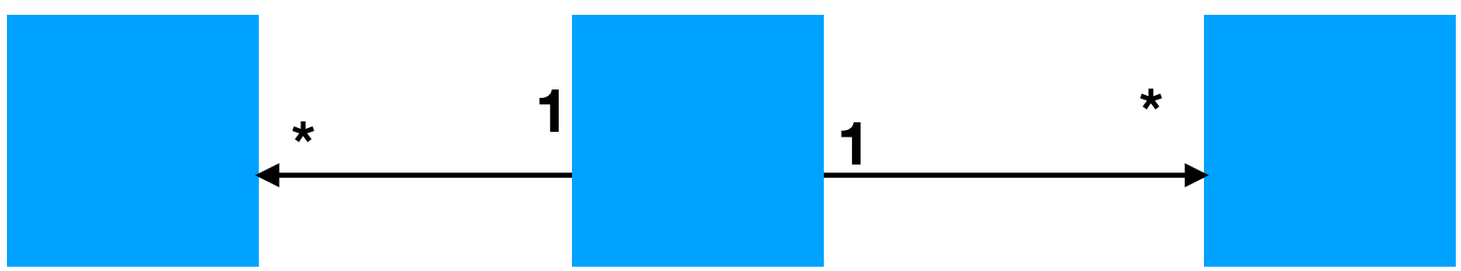
Domain Layer

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


Ship
Handler
Station
1
Comet
*


Ship
Station
Aestroid
Comet


*
*


*
1
1
*


Dlg
Acc
CADlg
SADlg
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
-

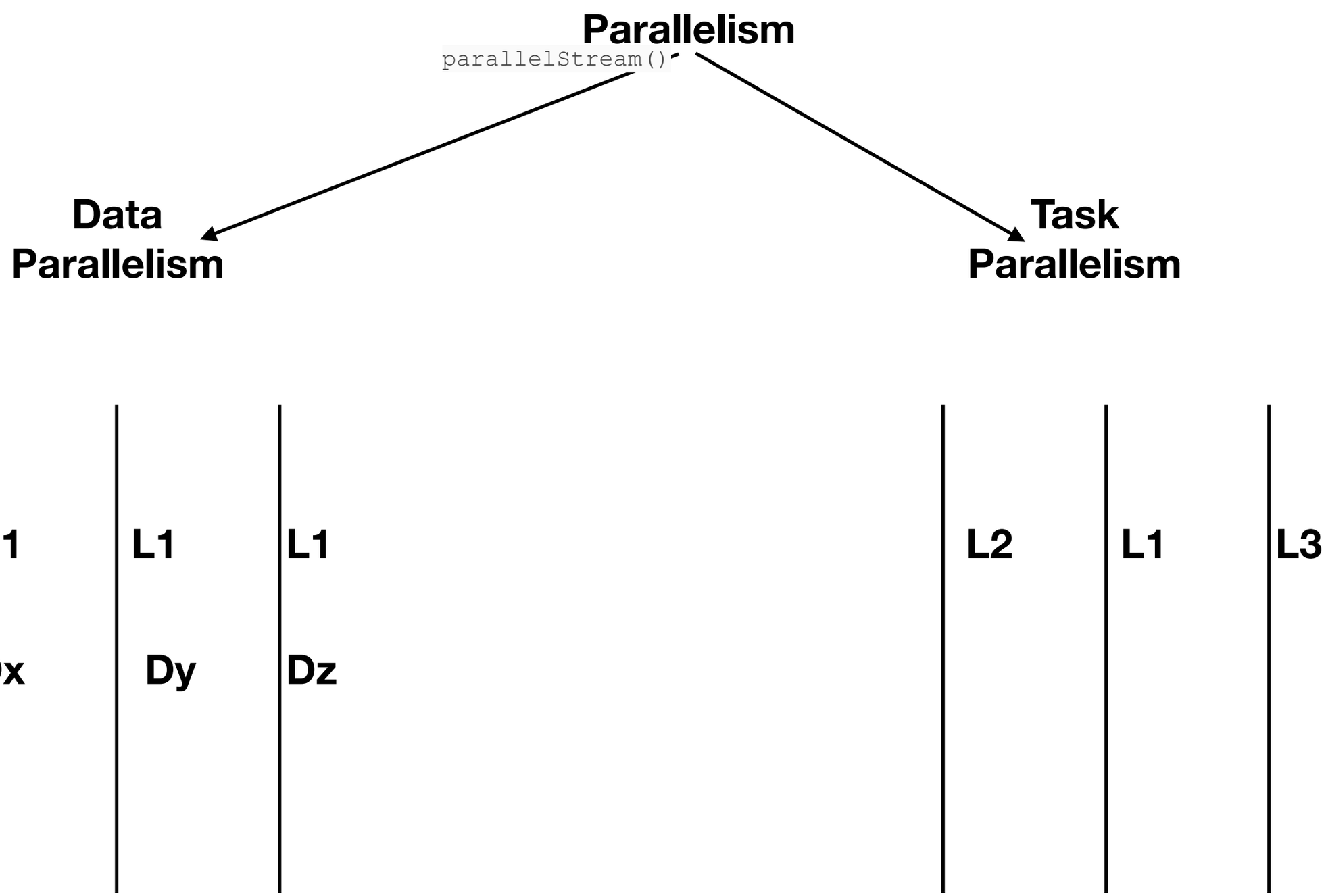
- Prefer composition over inheritance
- Boundary control entity (*)
- YAGNI
- KISS
- Program to an Interface
- DDD
  - Aggregates

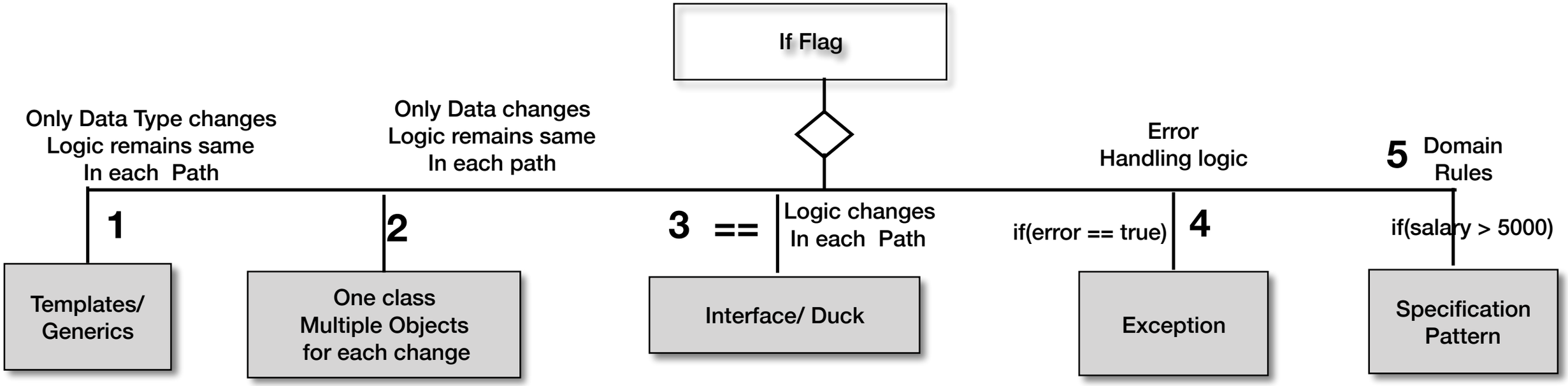
Bad

- 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)




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;

}
```

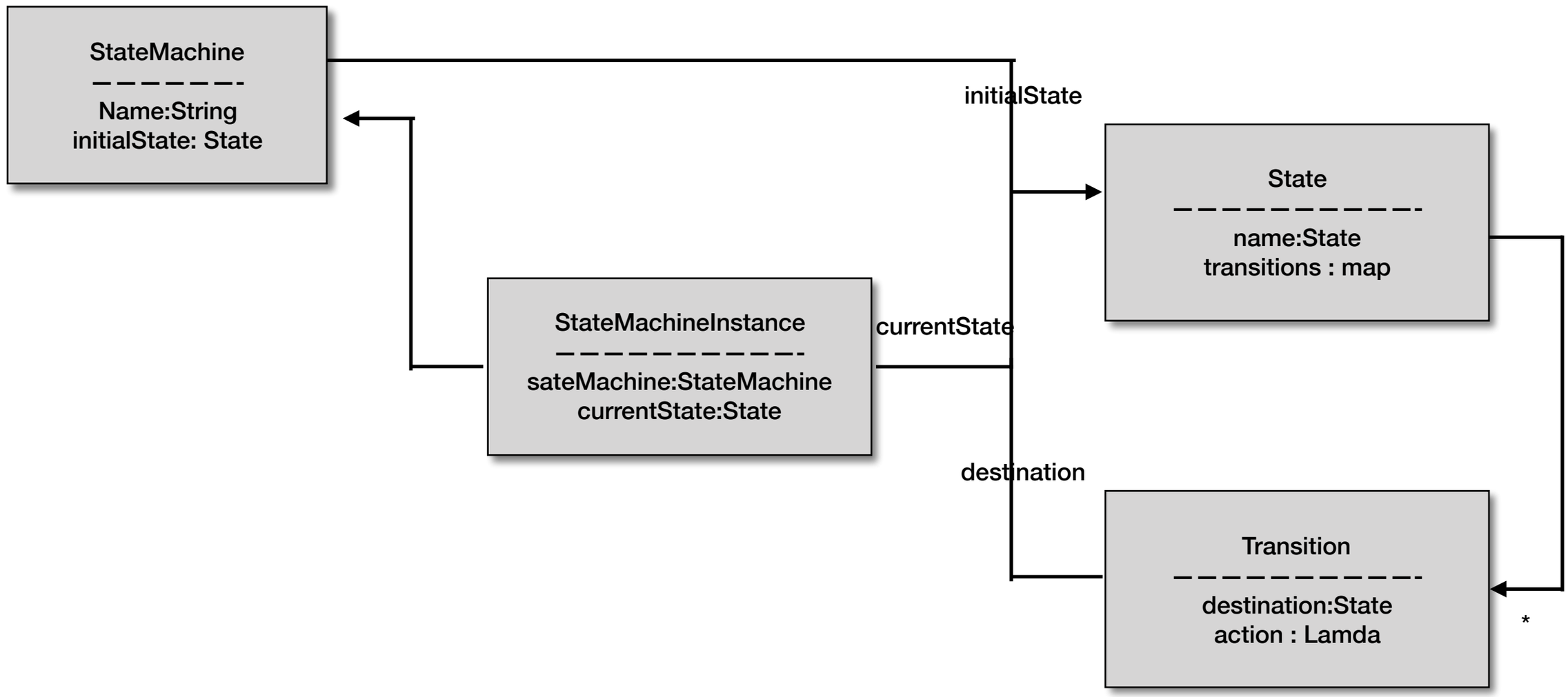
```
Class Emp{
List<Emp> emps;

}
```

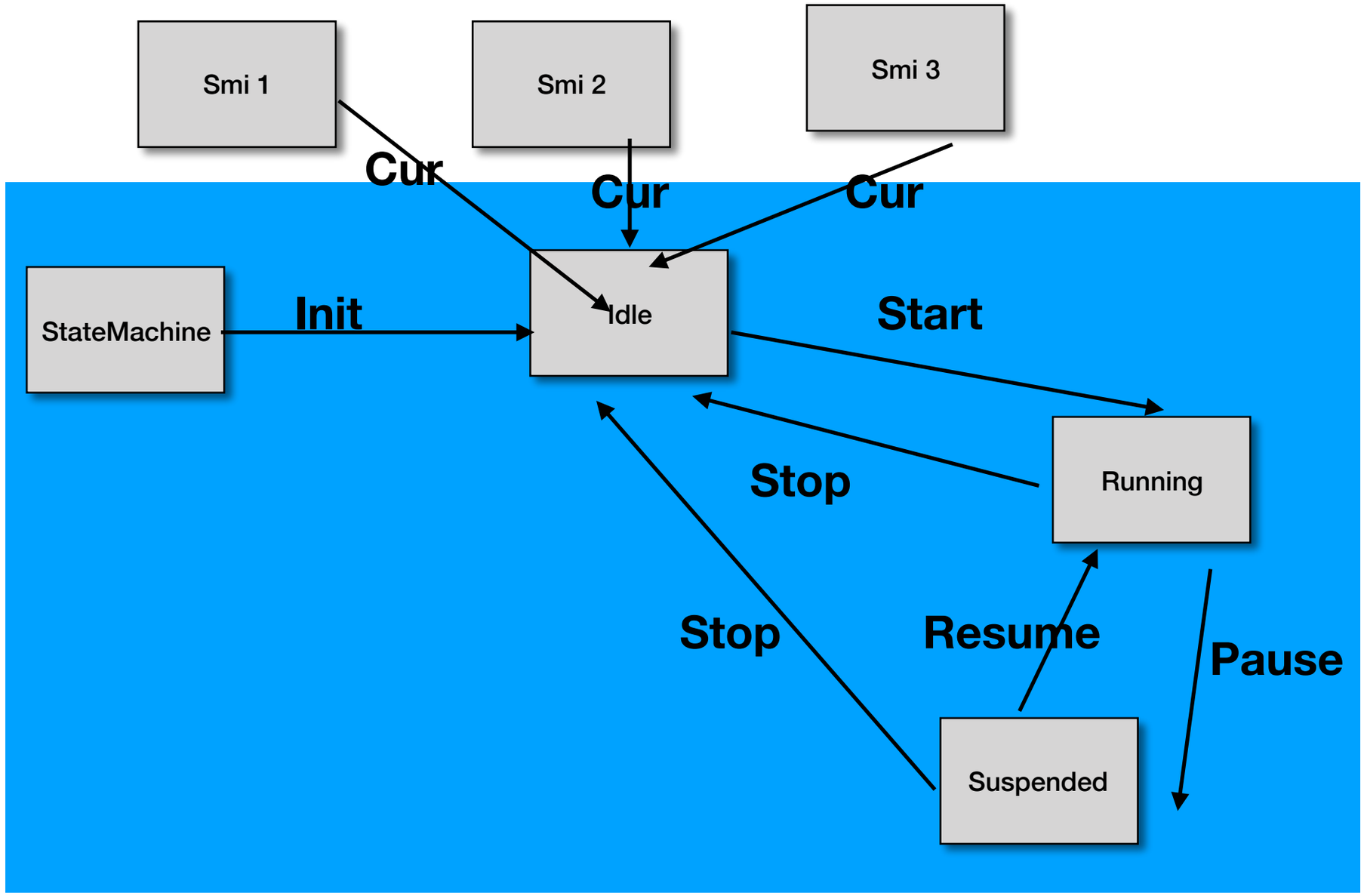
```
Class Emp{
Emp e1;
Emp e2;

}
```

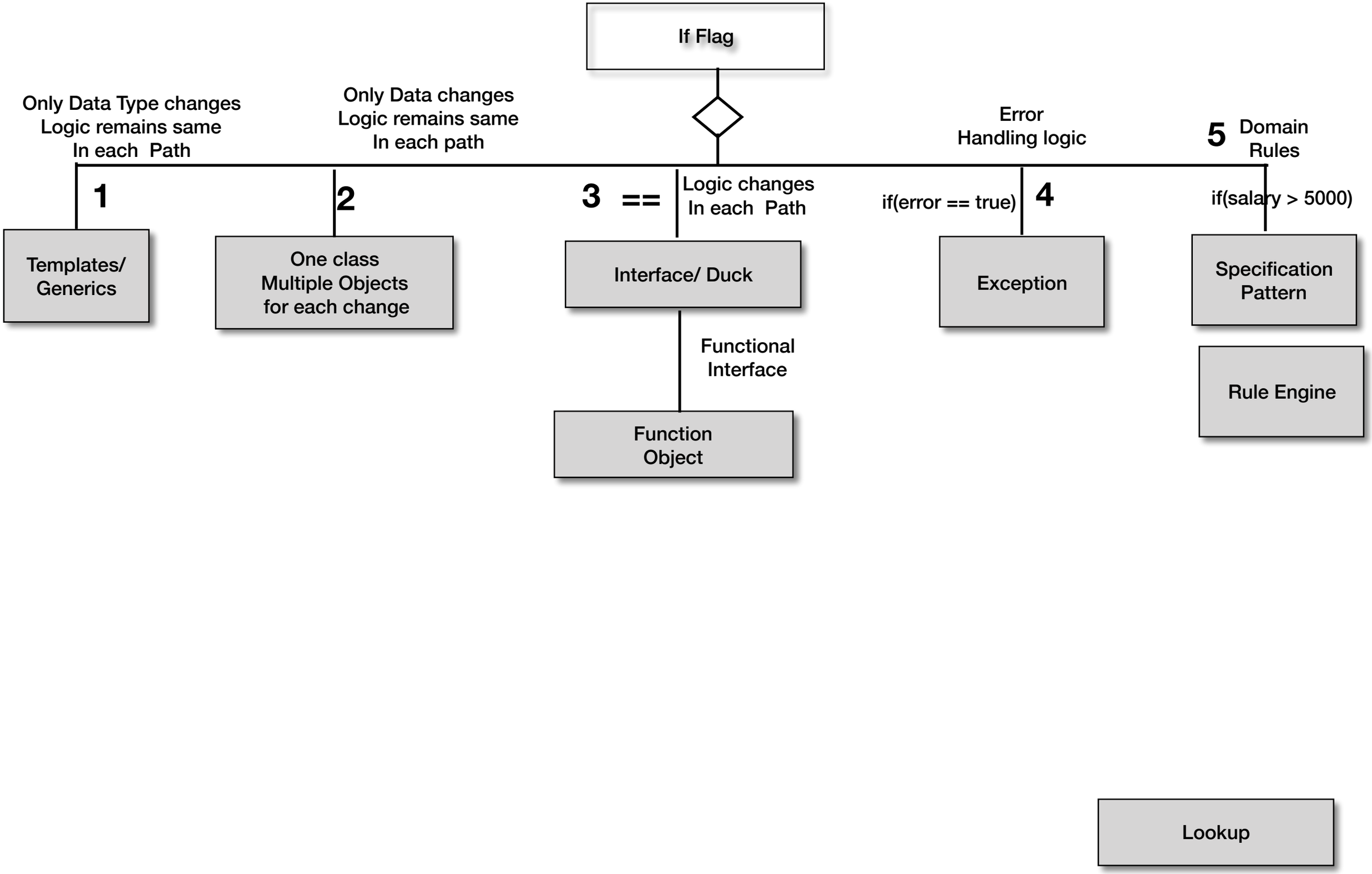
```
class State{


}
```

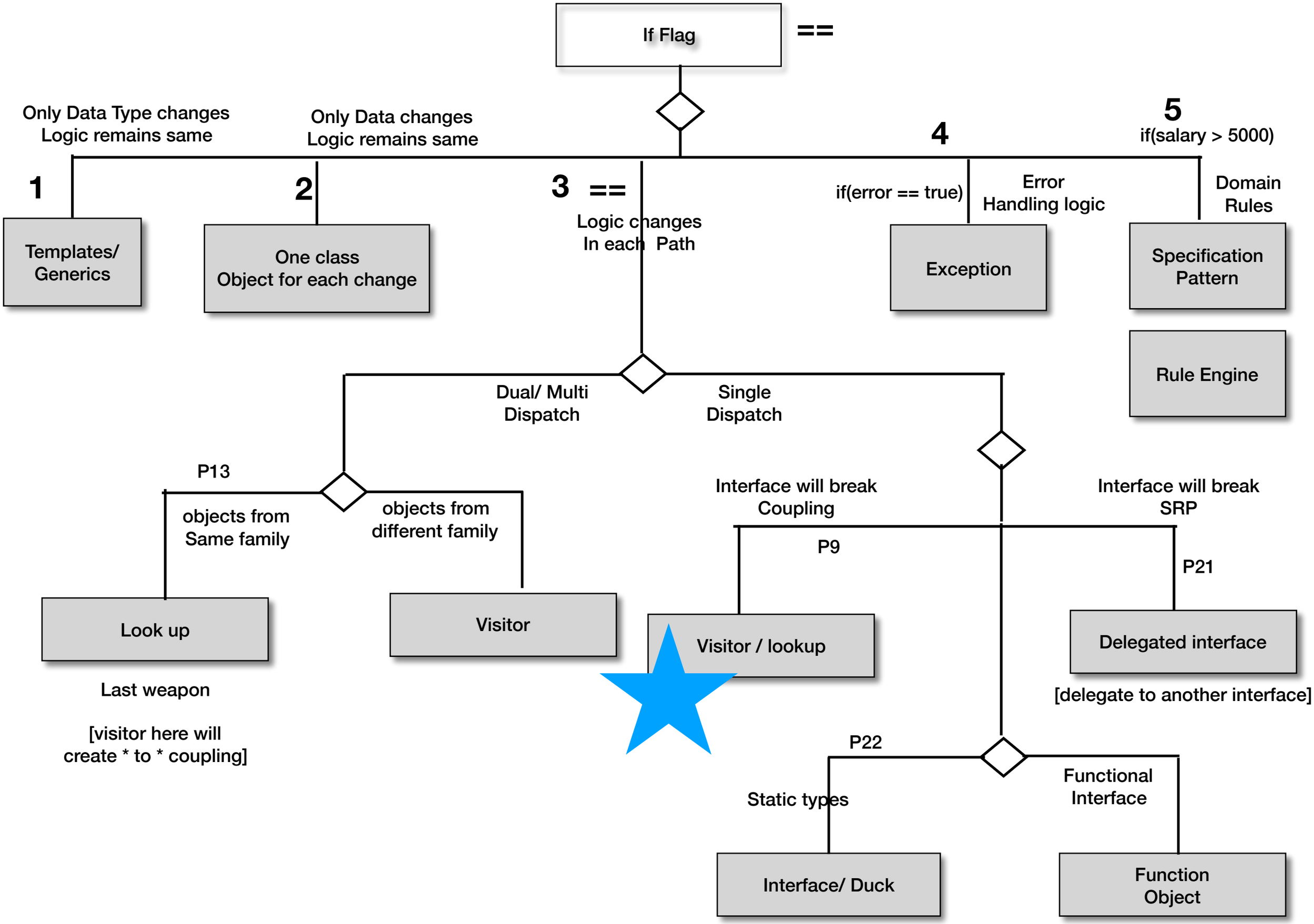
```
State idle = new State();
State running = new State();
State suspended = new State();
```

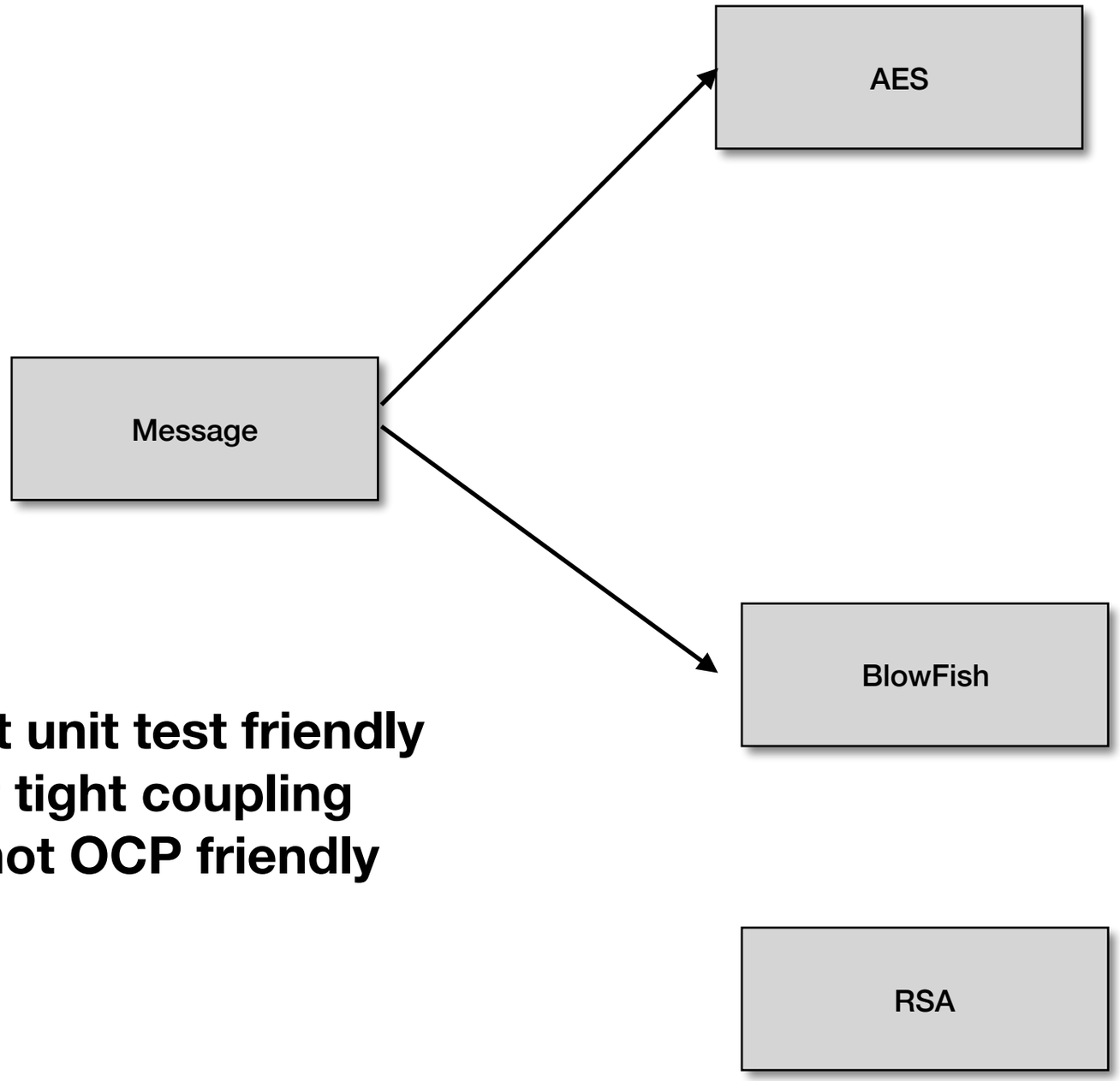
```
Class Idle implements State{}
Class Running implements State{}
Class Suspended implements State{}
```


StateMachine
Name:String
initialState: State
initialState
State
name:State
transitions : map
StateMachineInstance
sateMachine:StateMachine
currentState:State
currentState
destination
Transition
destination:State
action : Lamda
*

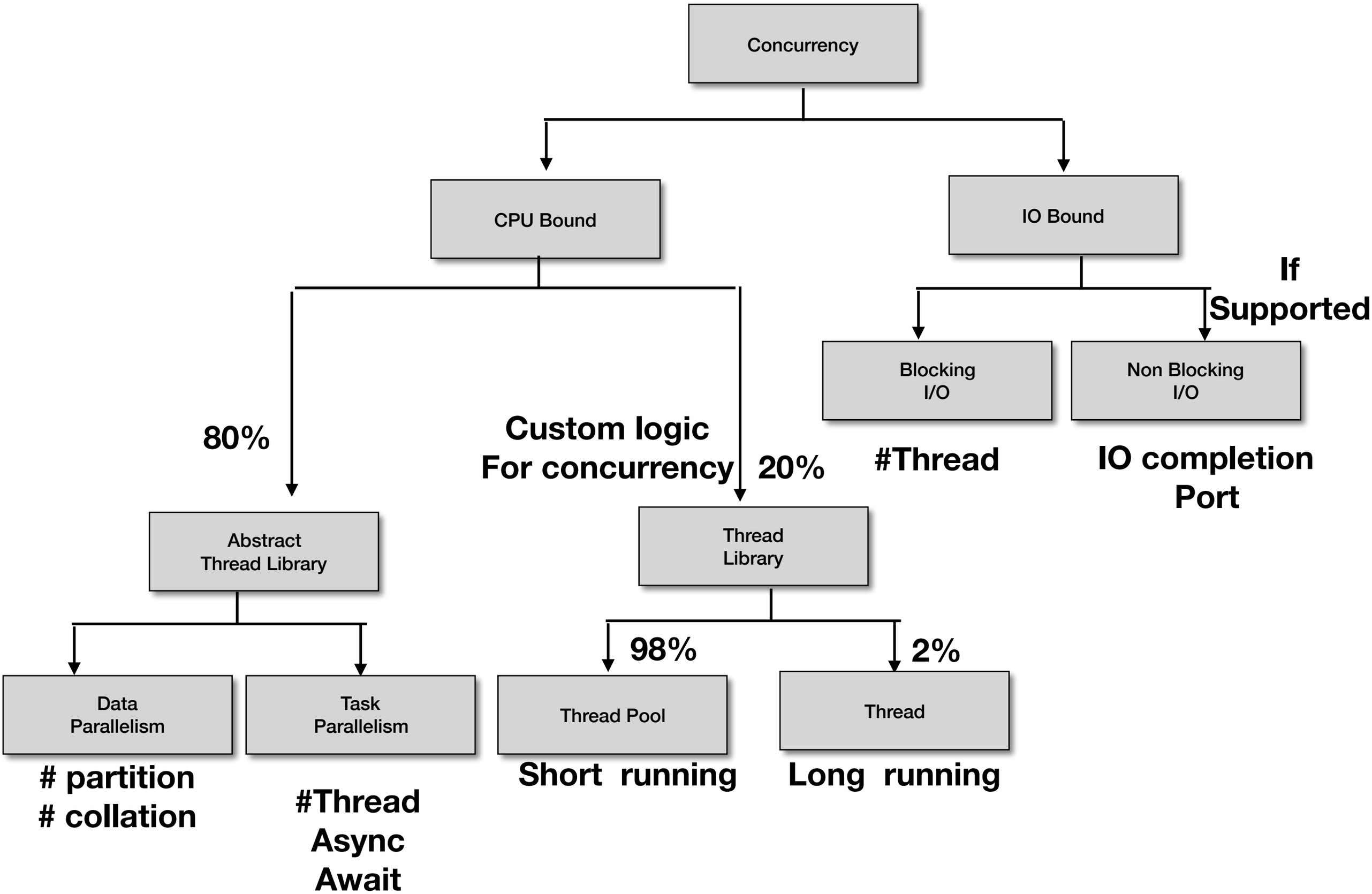

Smi 1
Smi 2
Smi 3
Cur
Cur
Cur
StateMachine
Init
Idle
Start
Running
Stop
Stop
Resume
Pause
Suspended


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
Functional
Interface
Function
Object
Error
Handling logic
if(error == true)
4
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
Single
Dispatch
P13
objects from
Same family
objects from
different family
Look up
Last weapon
[visitor here will
create * to * coupling]
Visitor
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

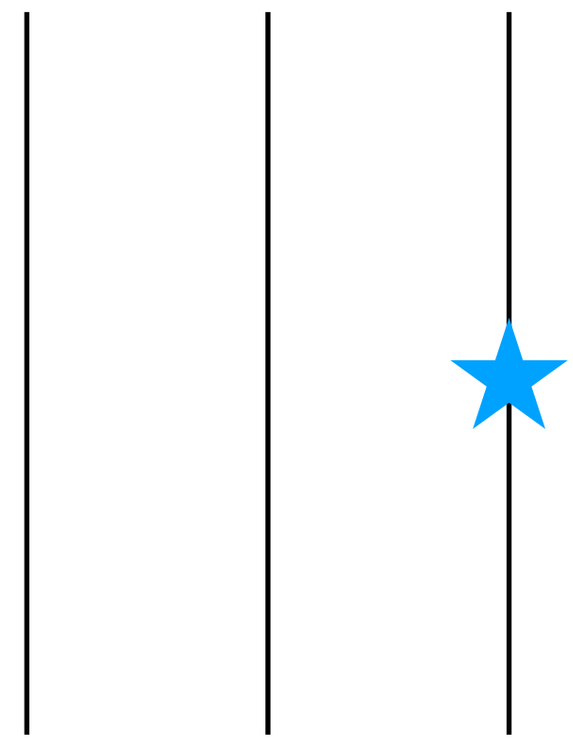


 # not unit test friendly
 # tight coupling
 # not OCP friendly


Concurrency
CPU Bound
IO Bound
If Supported
Blocking I/O
Non Blocking I/O
#Thread
IO completion Port
80%
Custom logic For concurrency
20%
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)

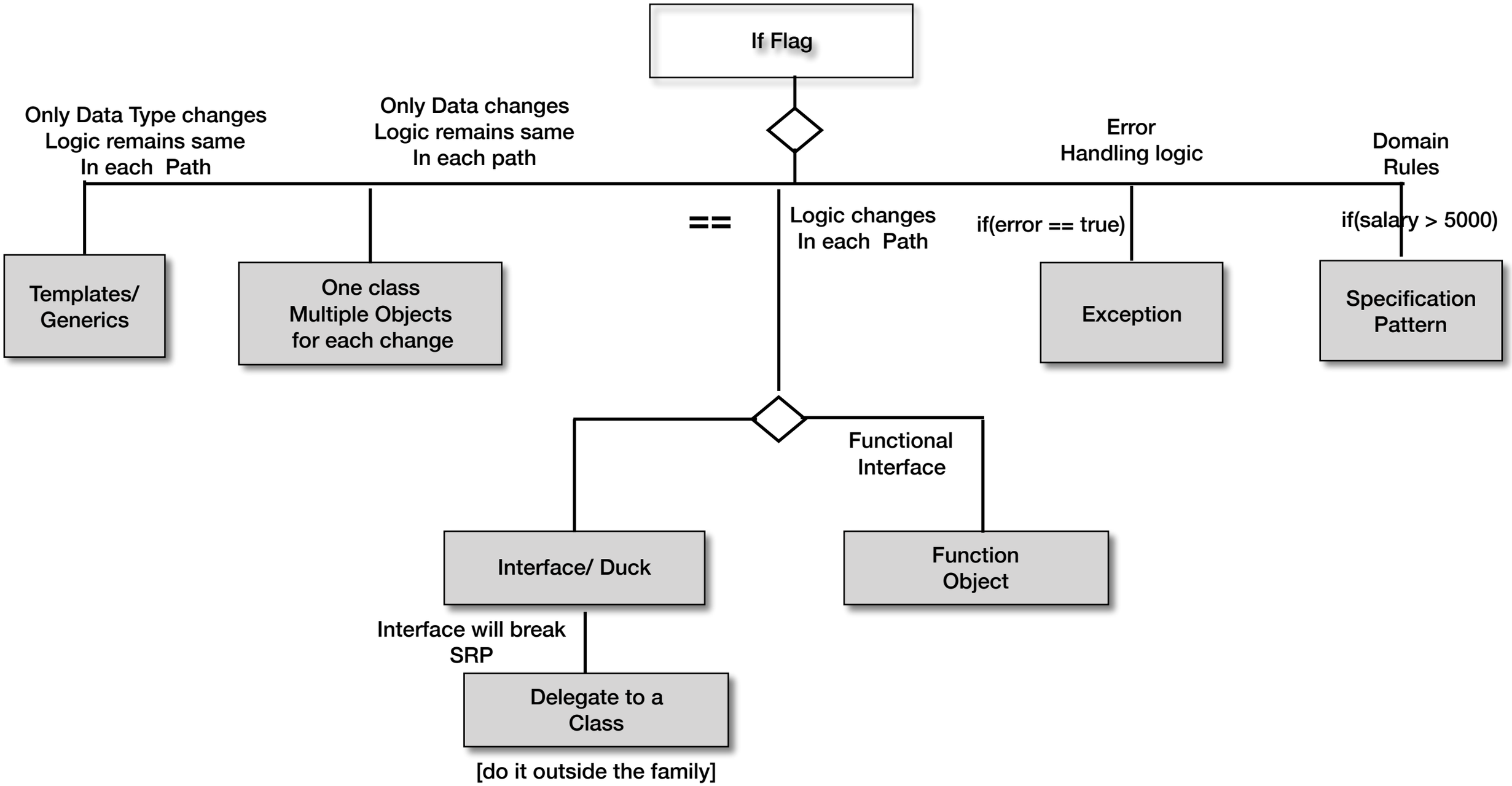


SOC

- 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
Functional
Interface
Interface/ Duck
Interface will break
SRP
Delegate to a
Class
[do it outside the family]
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

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

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

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

Upcast

vs

Downcast

```
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

Flag => Interface

easy to code

low cyclomatic complexity
readablity
unit test
OCP

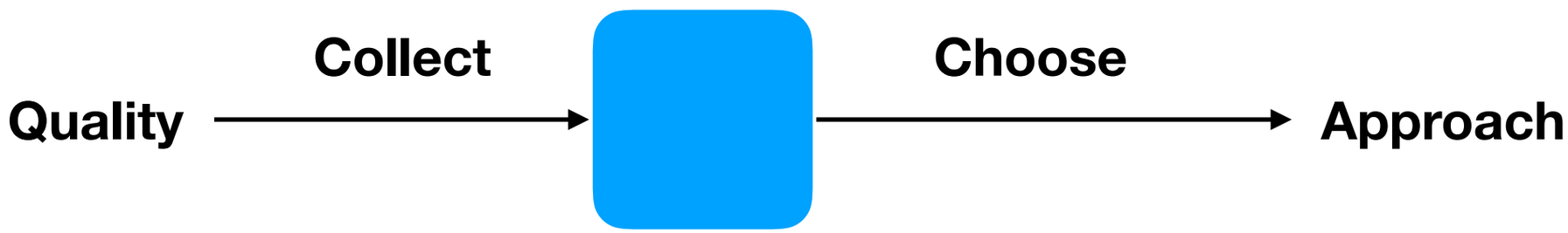
1

Flag => Polymorphism/ Abstraction/ Interface

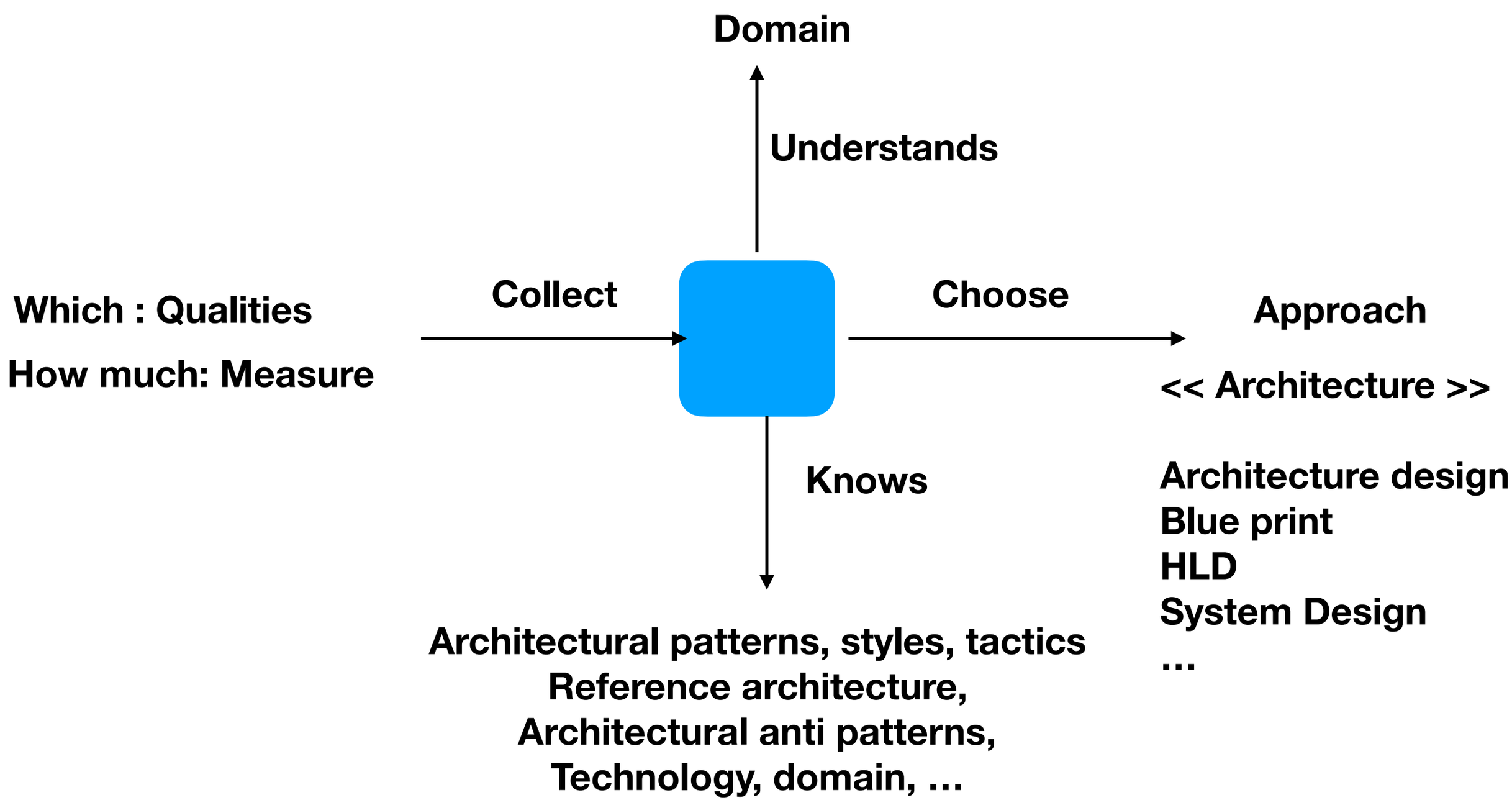
Coupling => Polymorphism/
Abstraction/
Interface

Type check => Polymorphism/ Abstraction/ Interface

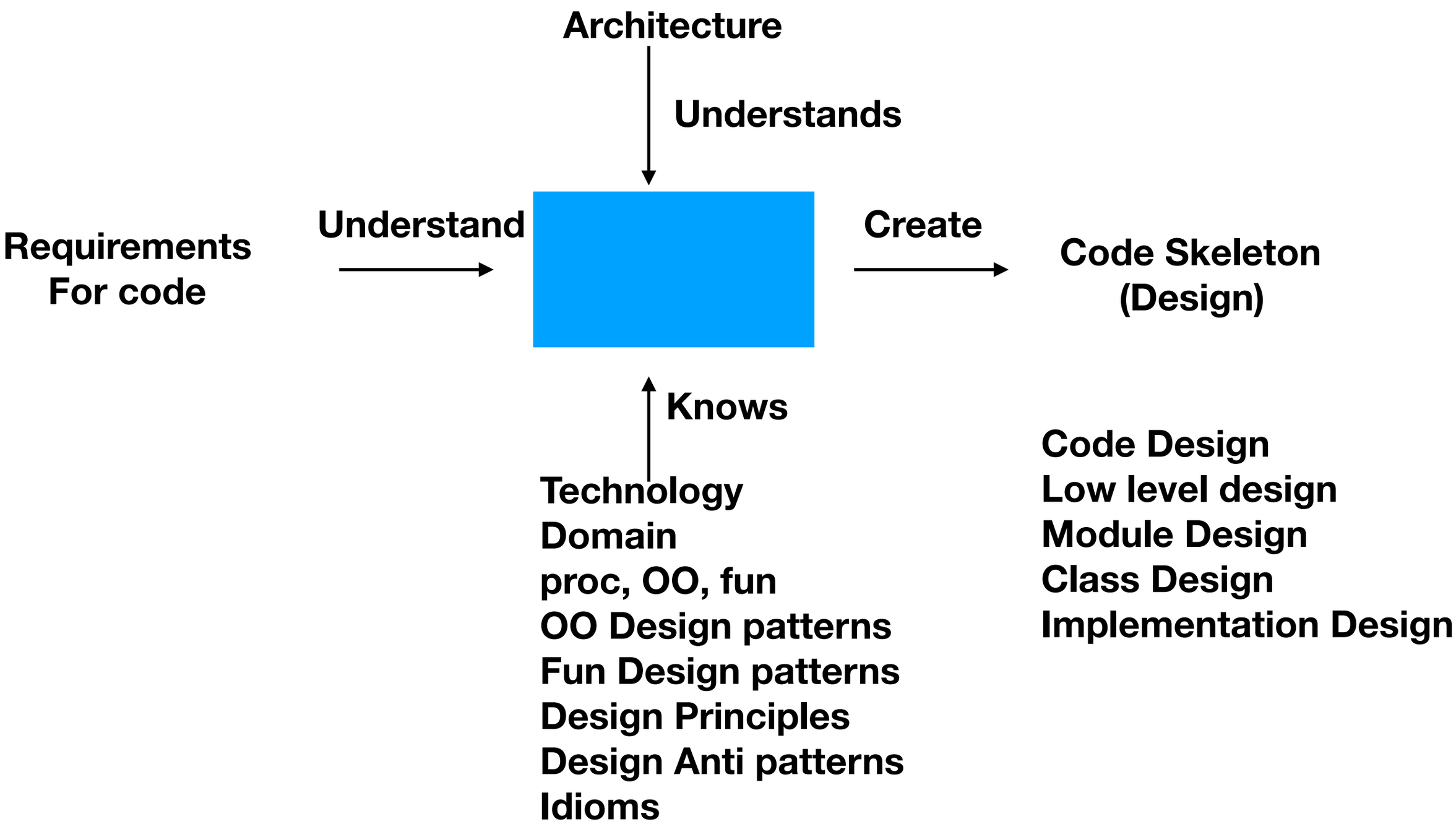
Down casting => Polymorphism/ Abstraction/ Interface


Collect
Choose
Quality
Approach

“system quality”



Code Maintainability



- 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{
    ?
}
```

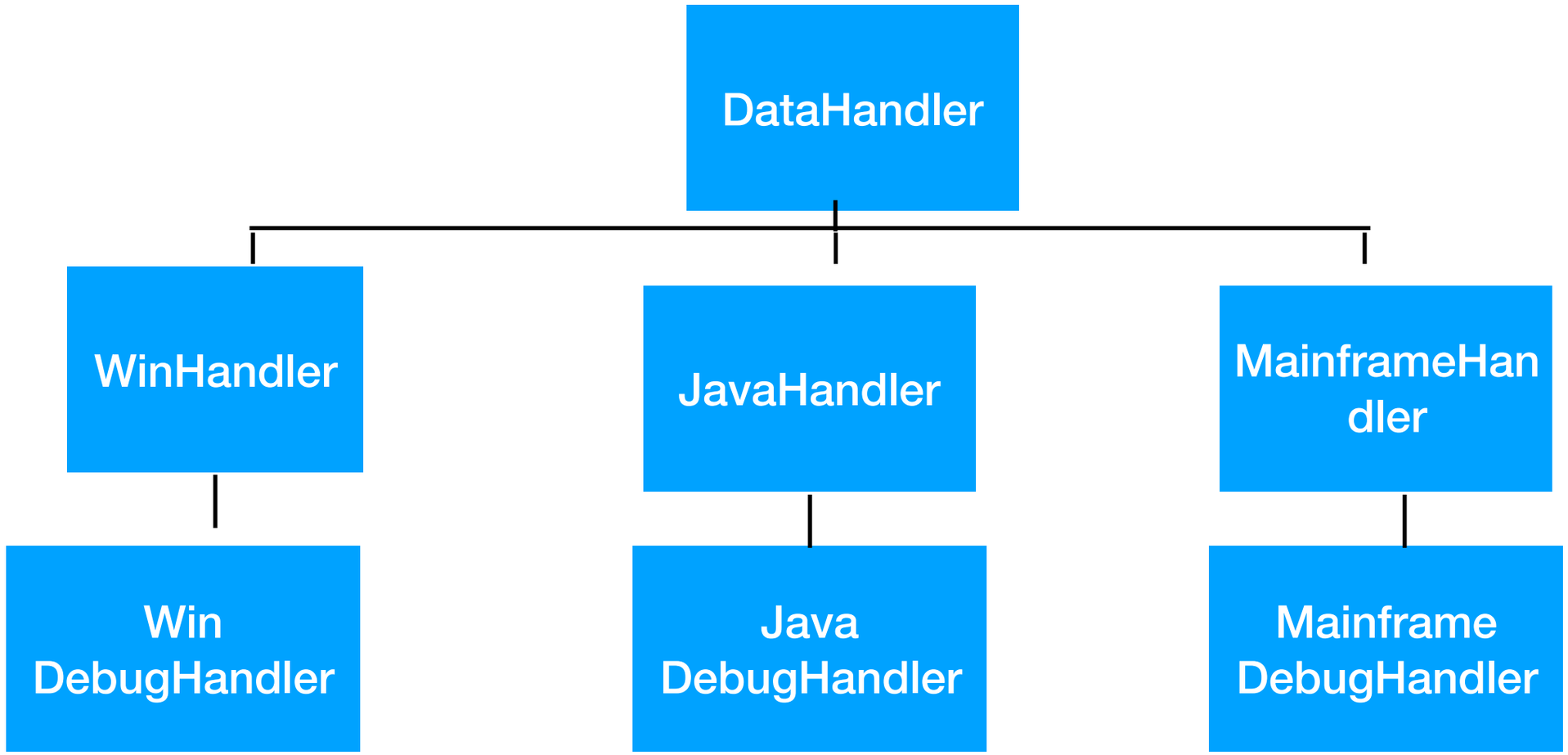
```
Interface Bird extends LivingThing{

}
```

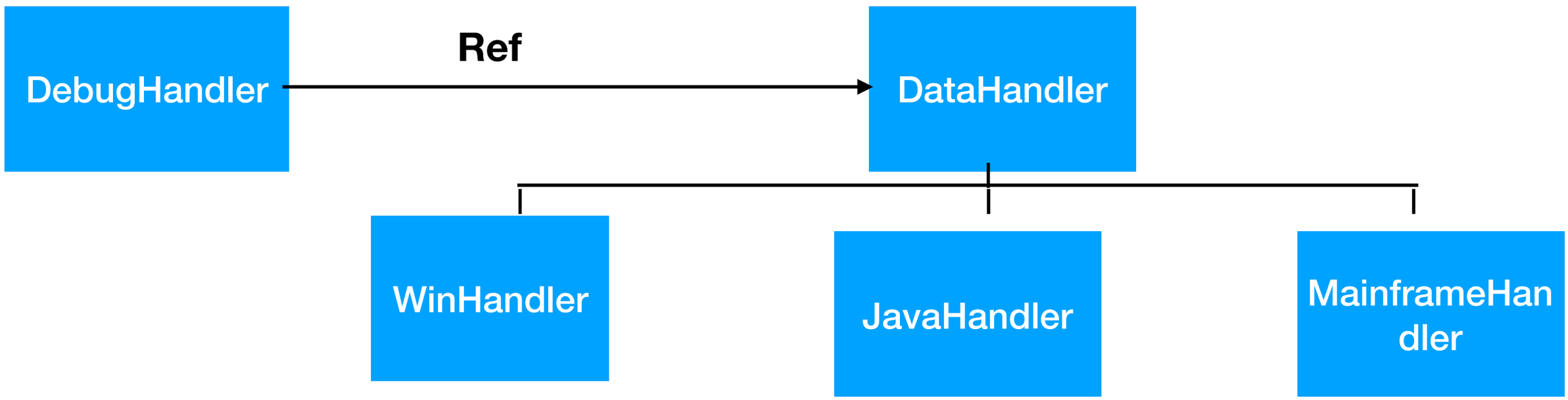
```
fun(Bird bird){

    ....

}
```

```
Class Parrot{
}
```

```
Interface Bird{
}
```

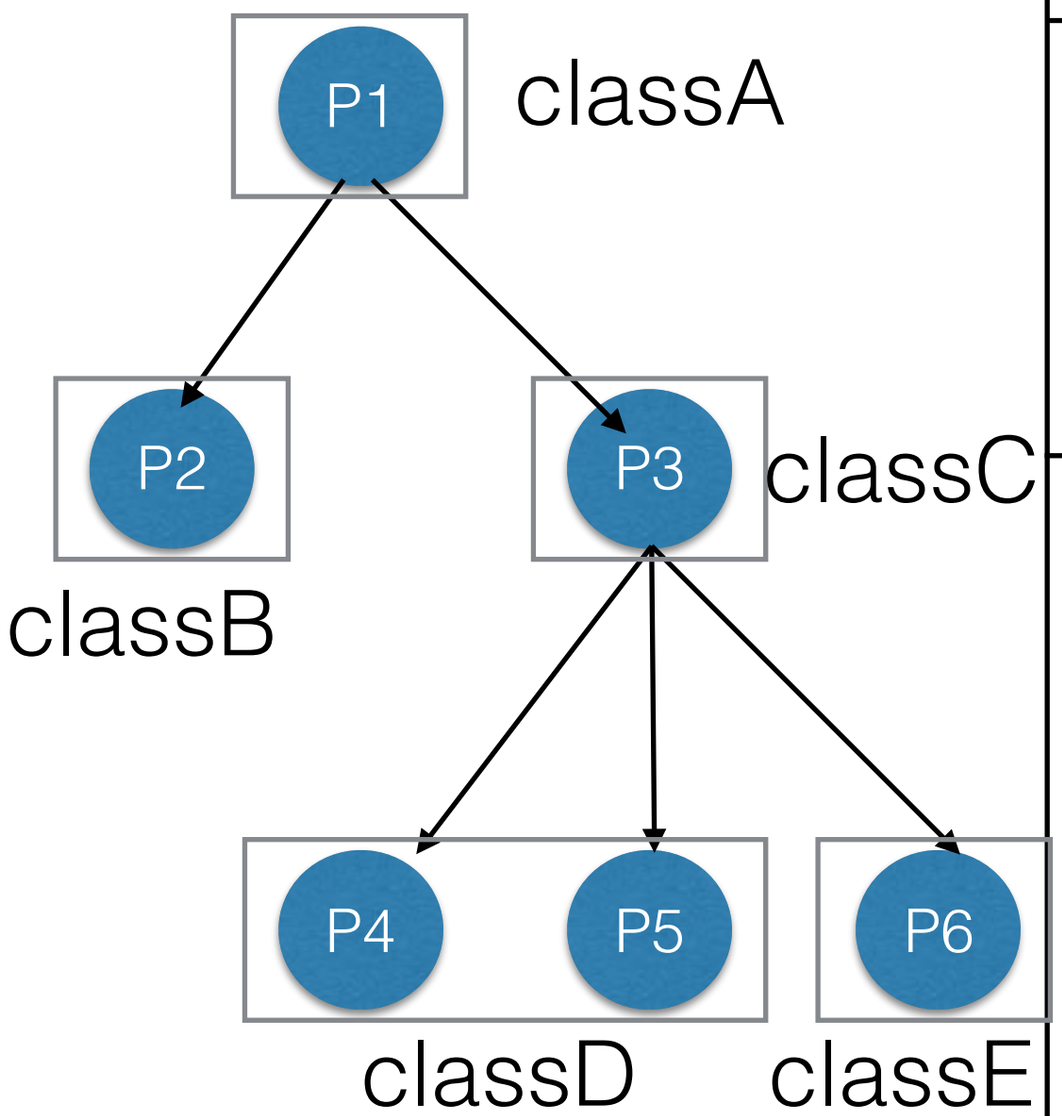
```
interface LivingThing{
    ...
}

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


...
   layEggs()
    swim()
   }
```


DataHandler
WinHandler
JavaHandler
MainframeHan
dler
Win
DebugHandler
Java
DebugHandler
Mainframe
DebugHandler

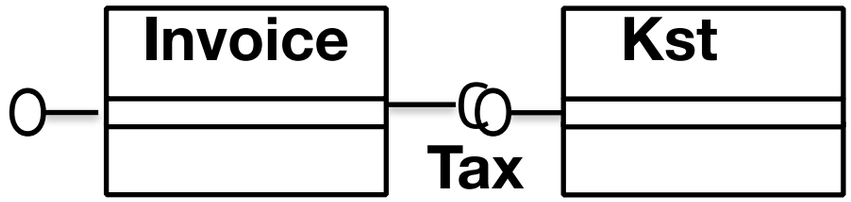

DebugHandler
Ref
DataHandler
WinHandler
JavaHandler
MainframeHan
dler

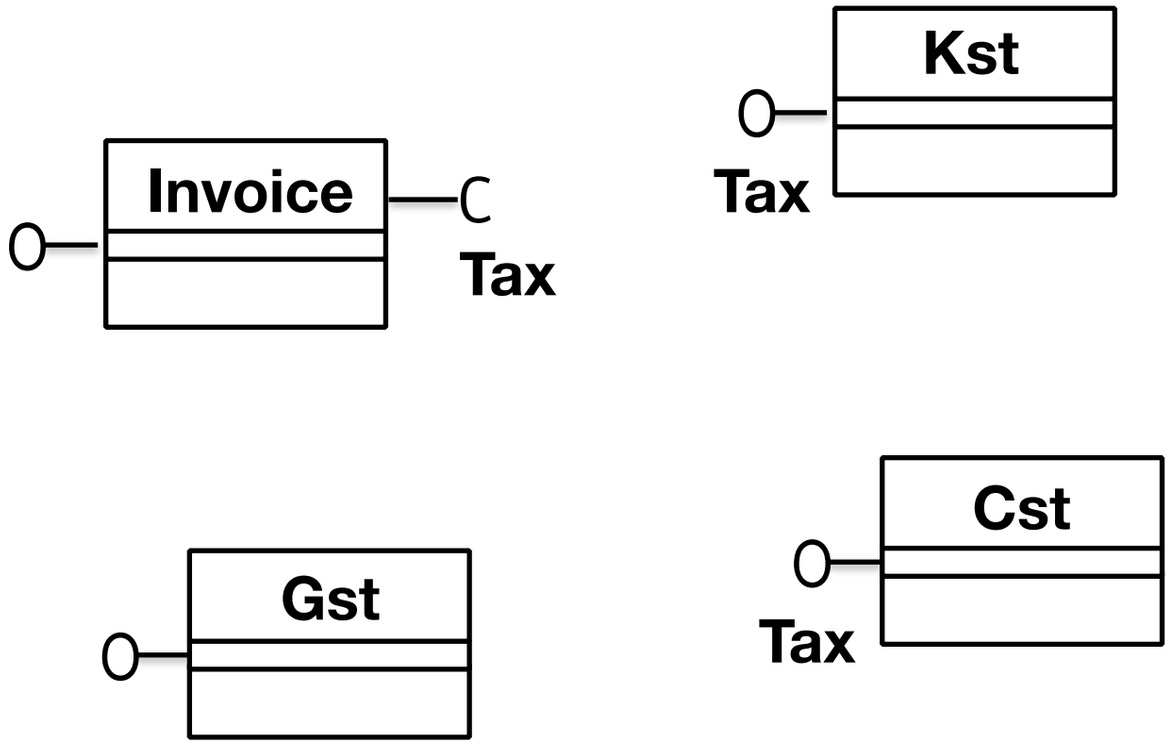
a+b	3 cpu cycles
Fun call	10 cpu cycles
Exception handling	1000 cpu cycles
Create thread	200,000 cpu cycles
Write to file	10,00,000 cpu cycles
Db call	40,00,000 cpu cycles

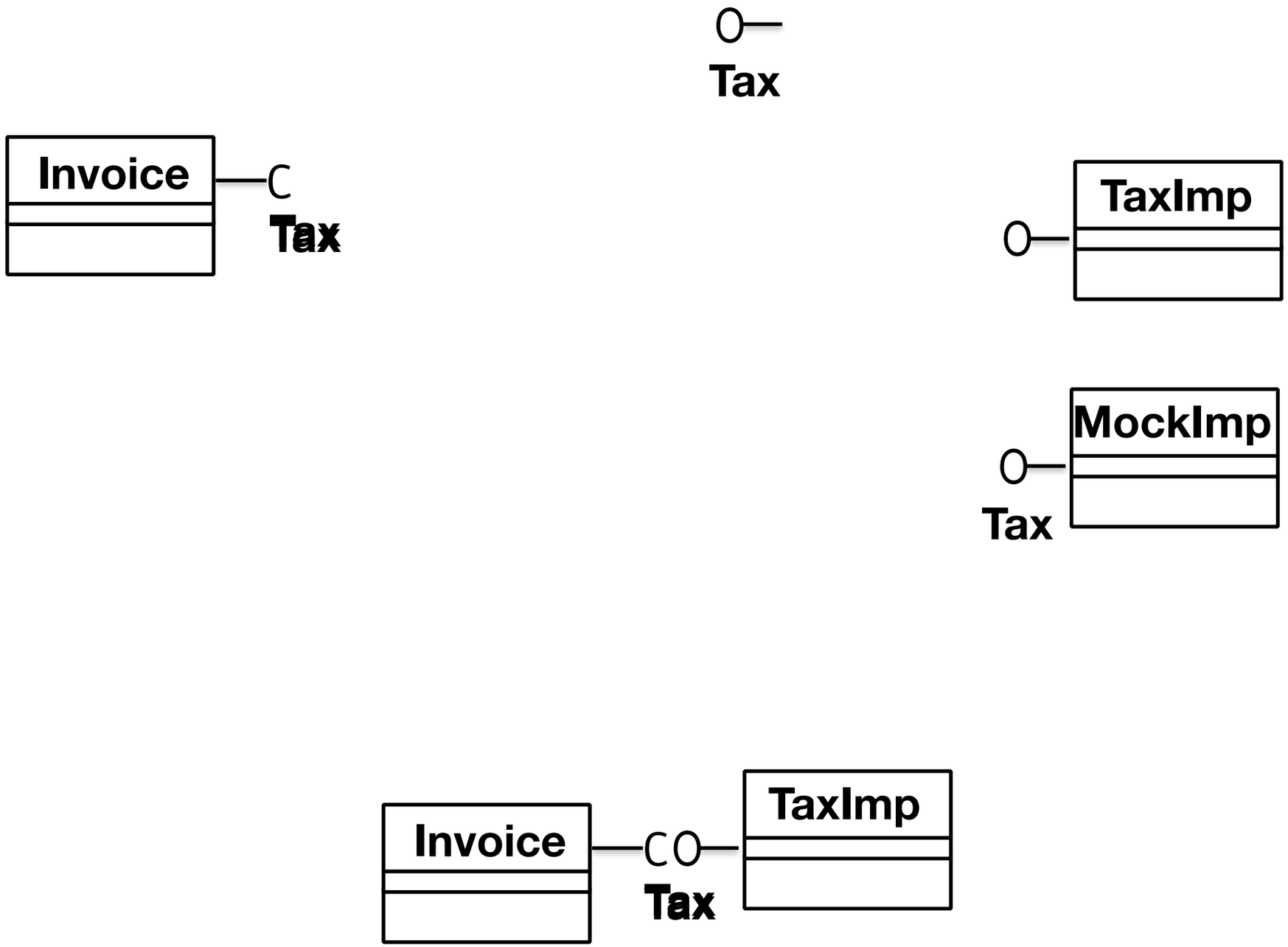
Procedural Prog
(tree)



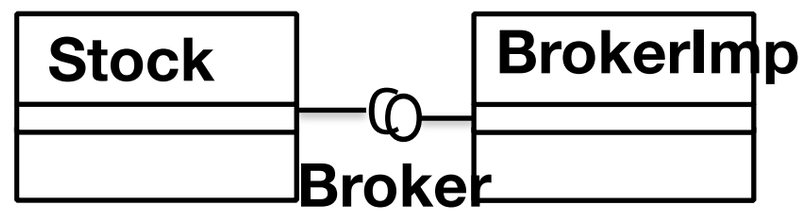
OO Prog
(Lego)



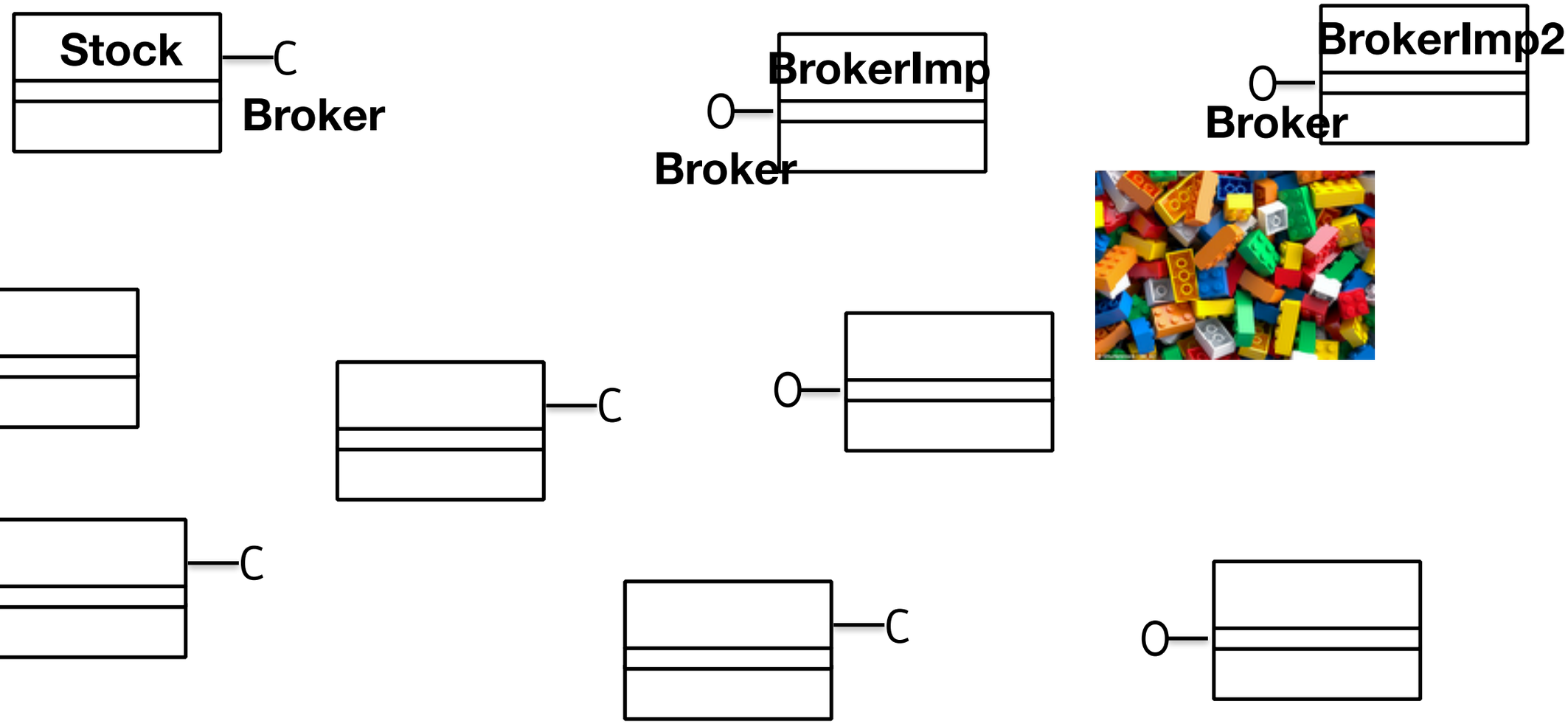



Tax
Invoice
Tax
TaxImp
MockImp
Tax
Invoice
Tax
TaxImp

Flow


Stock
BrokerImp
Broker

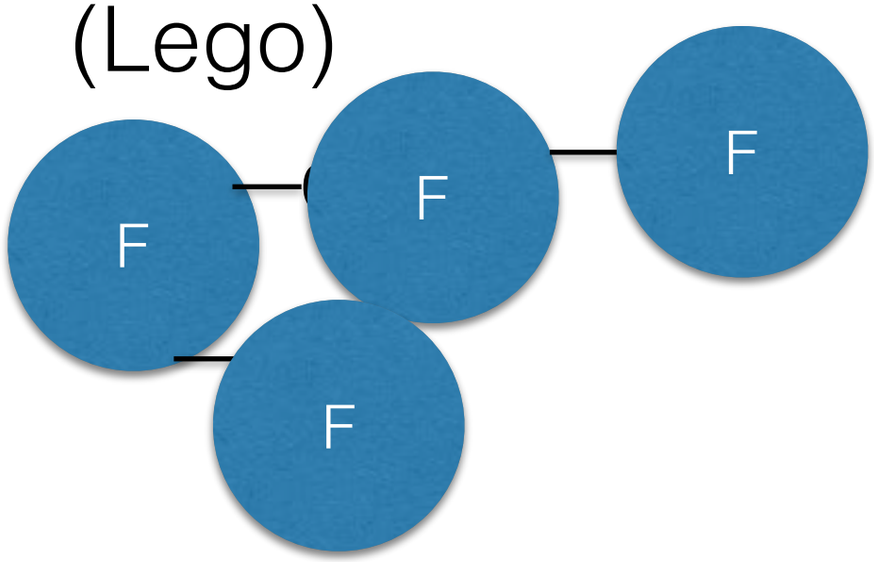
Steps


Stock
Broker
BrokerImp
Broker
BrokerImp2
Broker

	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	- -	- -	+ +

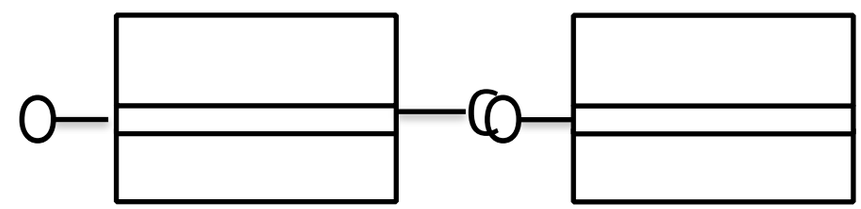
Functional Prog

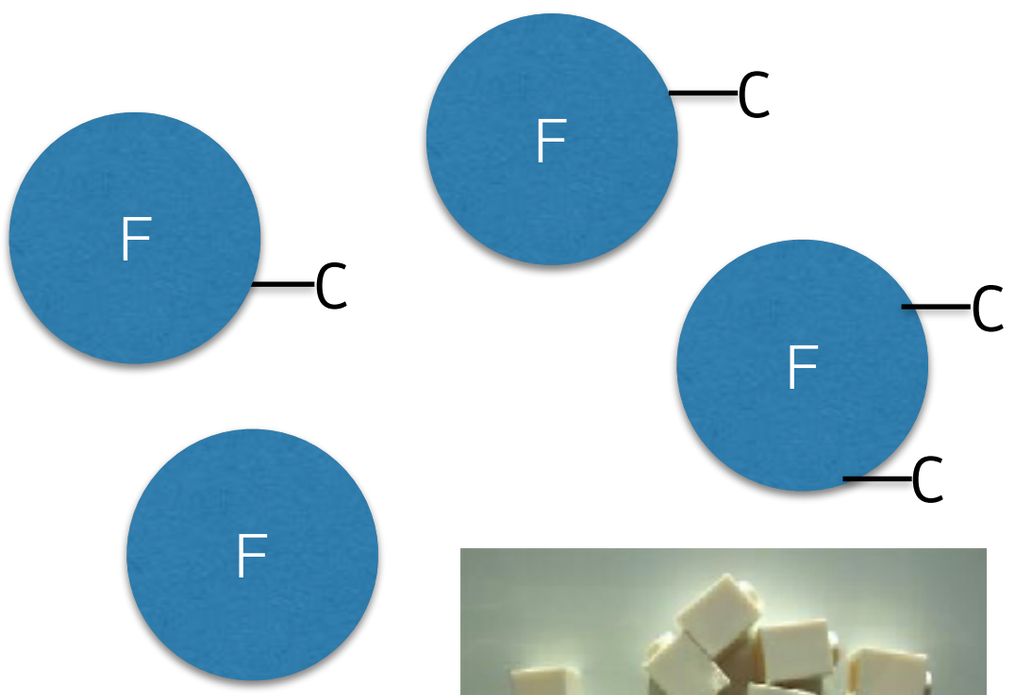
(Lego)

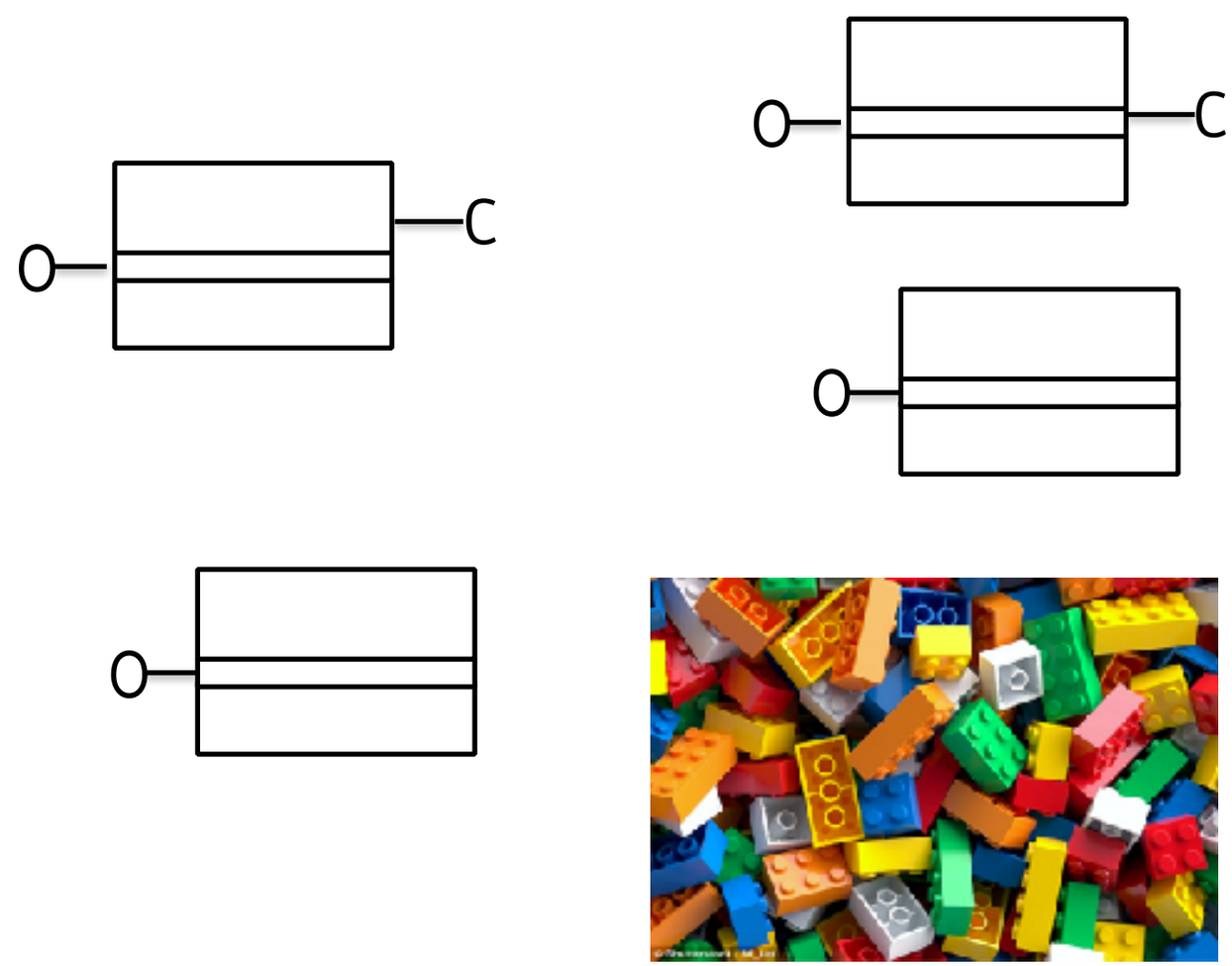


OO Prog

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

do(Parrot obj)
{
obj.fly();
}
	do(Bird obj)
	{
	obj.f1();
	}
		do(obj)
		{
		obj.f1();
		}
			do(Lamda f1)
			{
			f1();
			}

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

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();
		}

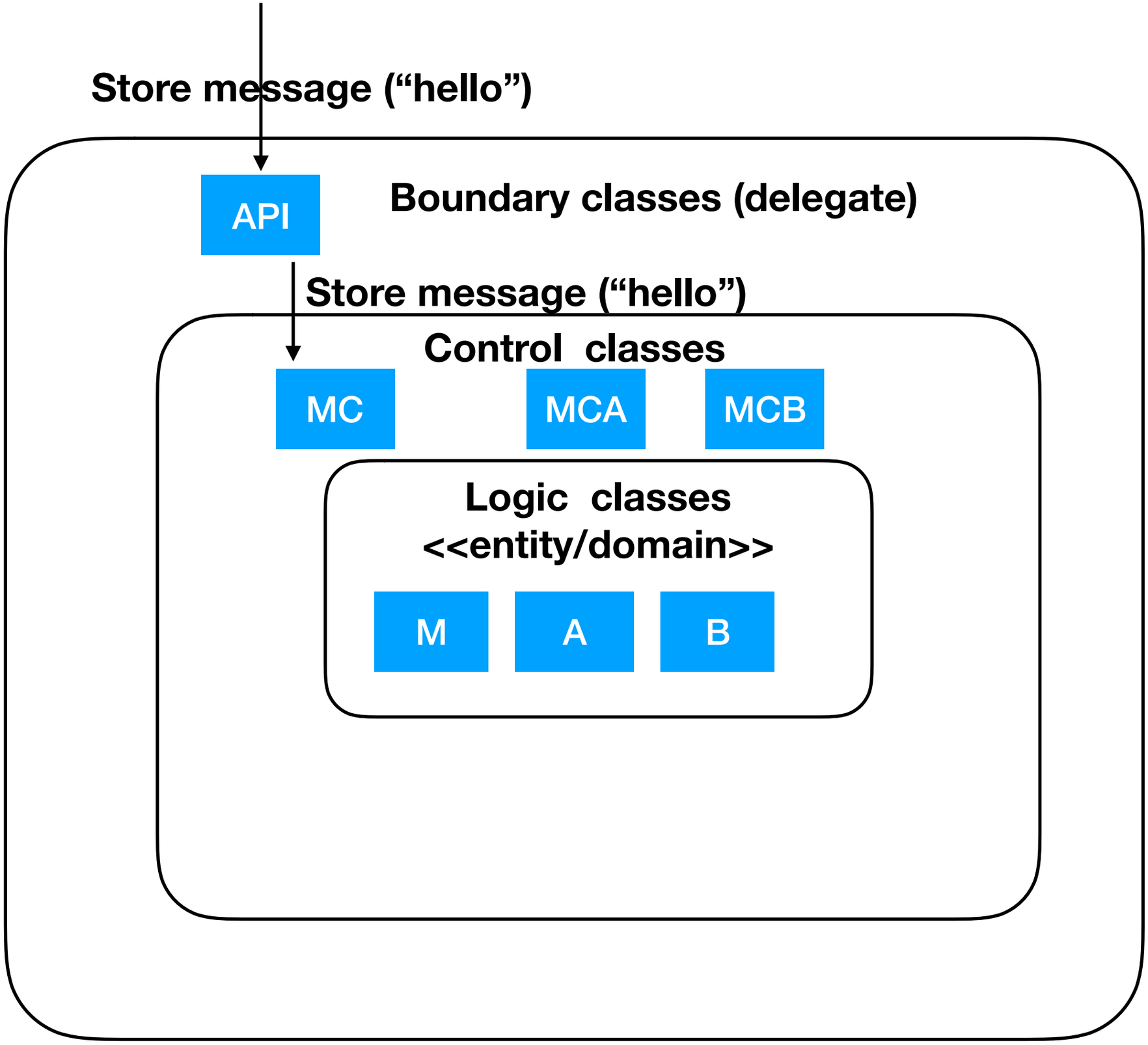
do(new Parrot( ))	do(new Parrot( ))	CA obj = new CA( ) do(()-> obj.fly())

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

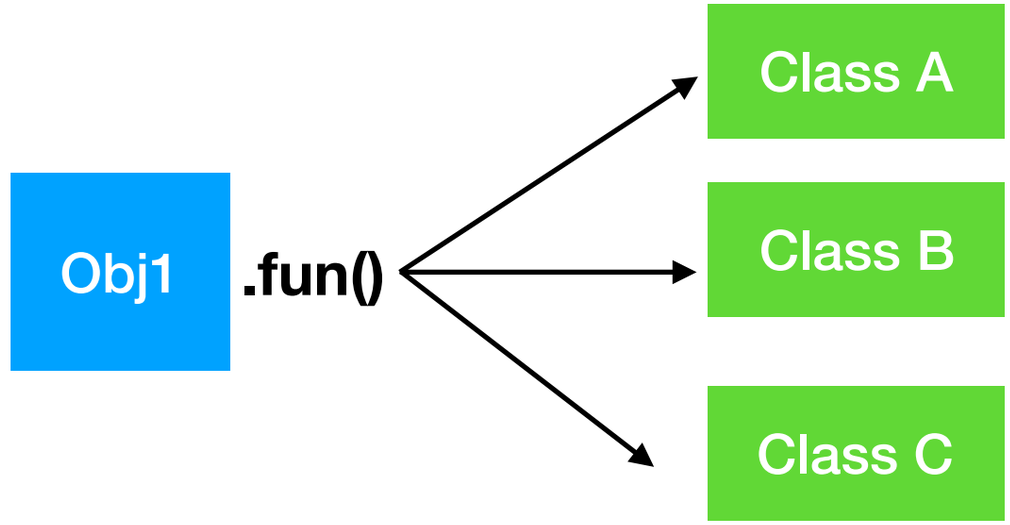
class Parrot
{
void fly(){
…
}
}
	interface Bird{
	void f1();
	}
	
	class Parrot implements Bird
	{
	void f1(){
	…
	}
	}
		class Parrot{
		void f1(){
		…
		}
		}
			class Parrot{
			void fly(){
			…
			}
			}
				class CA{
				void f1(){
				…
				}
				}

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”)

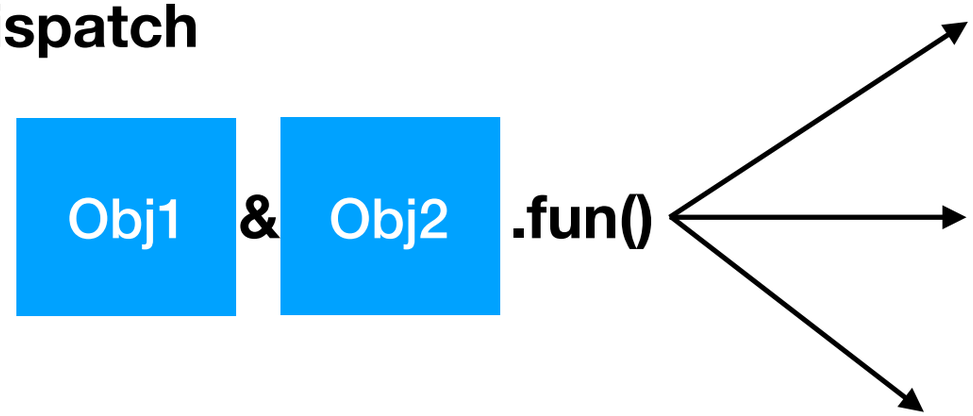

Store message (“hello”)
API
Boundary classes (delegate)
Store message (“hello”)
Control classes
MC
MCA
MCB
Logic classes
<<entity/domain>>
M
A
B

Single dispatch

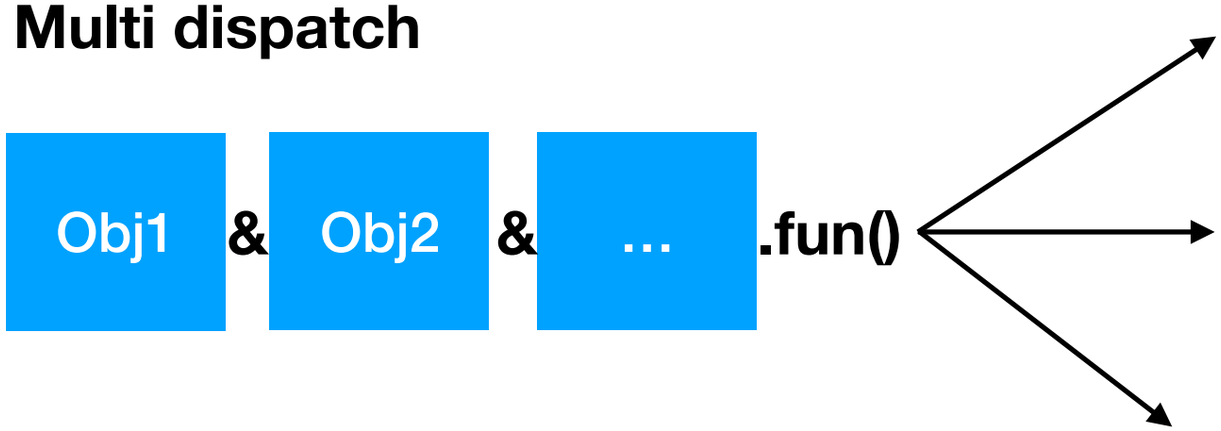


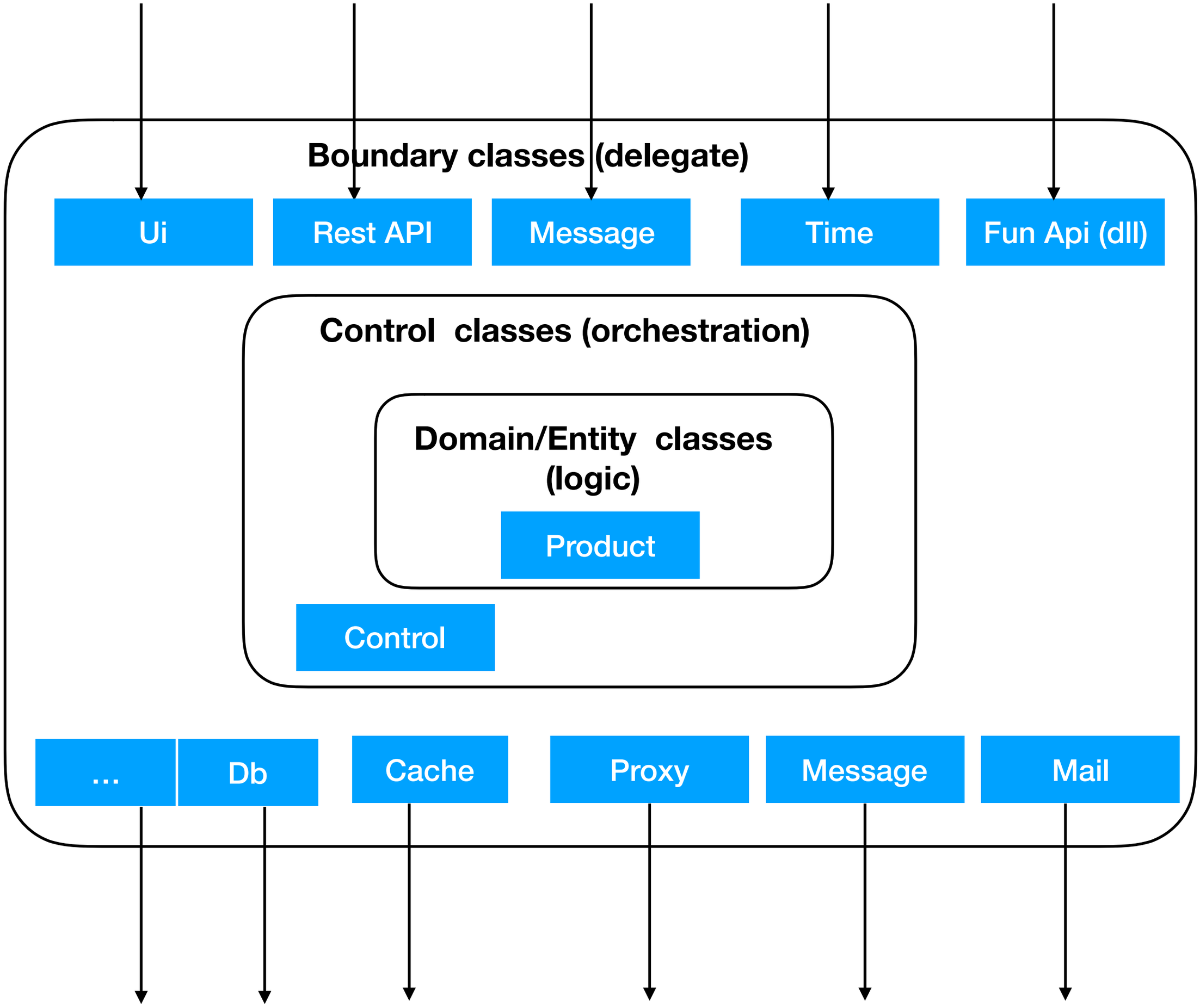
Java, c++, py, ..

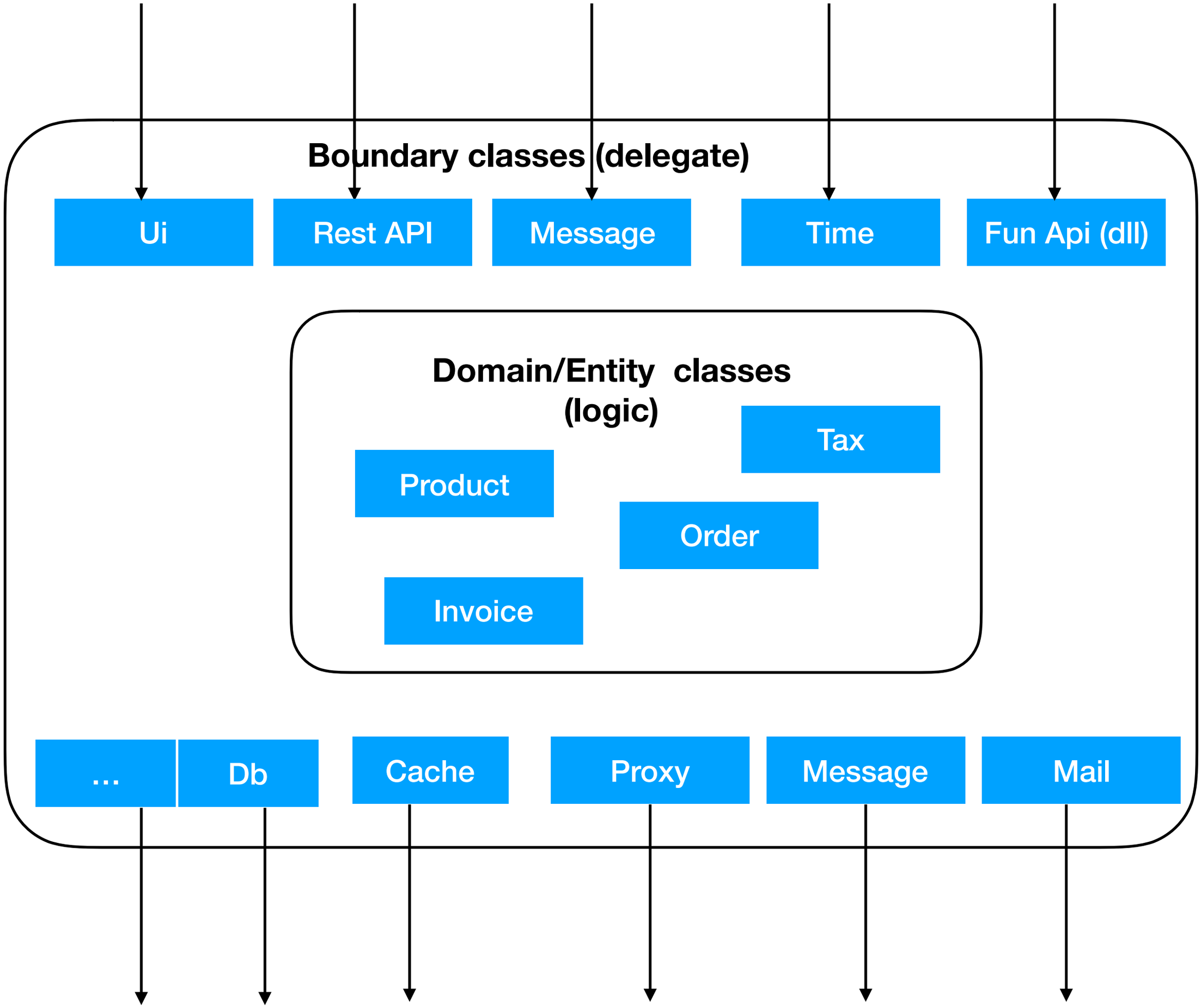
Dual dispatch

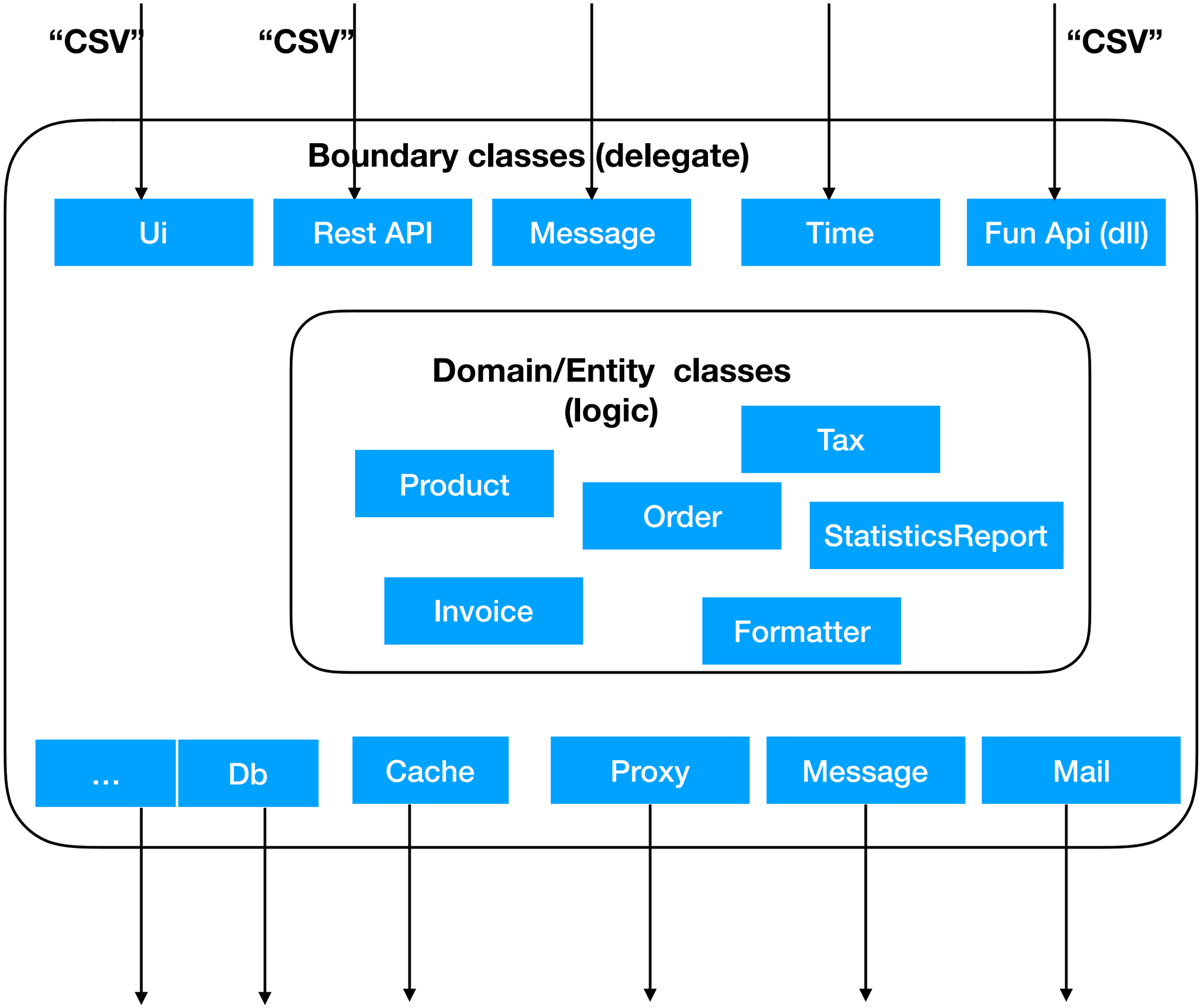


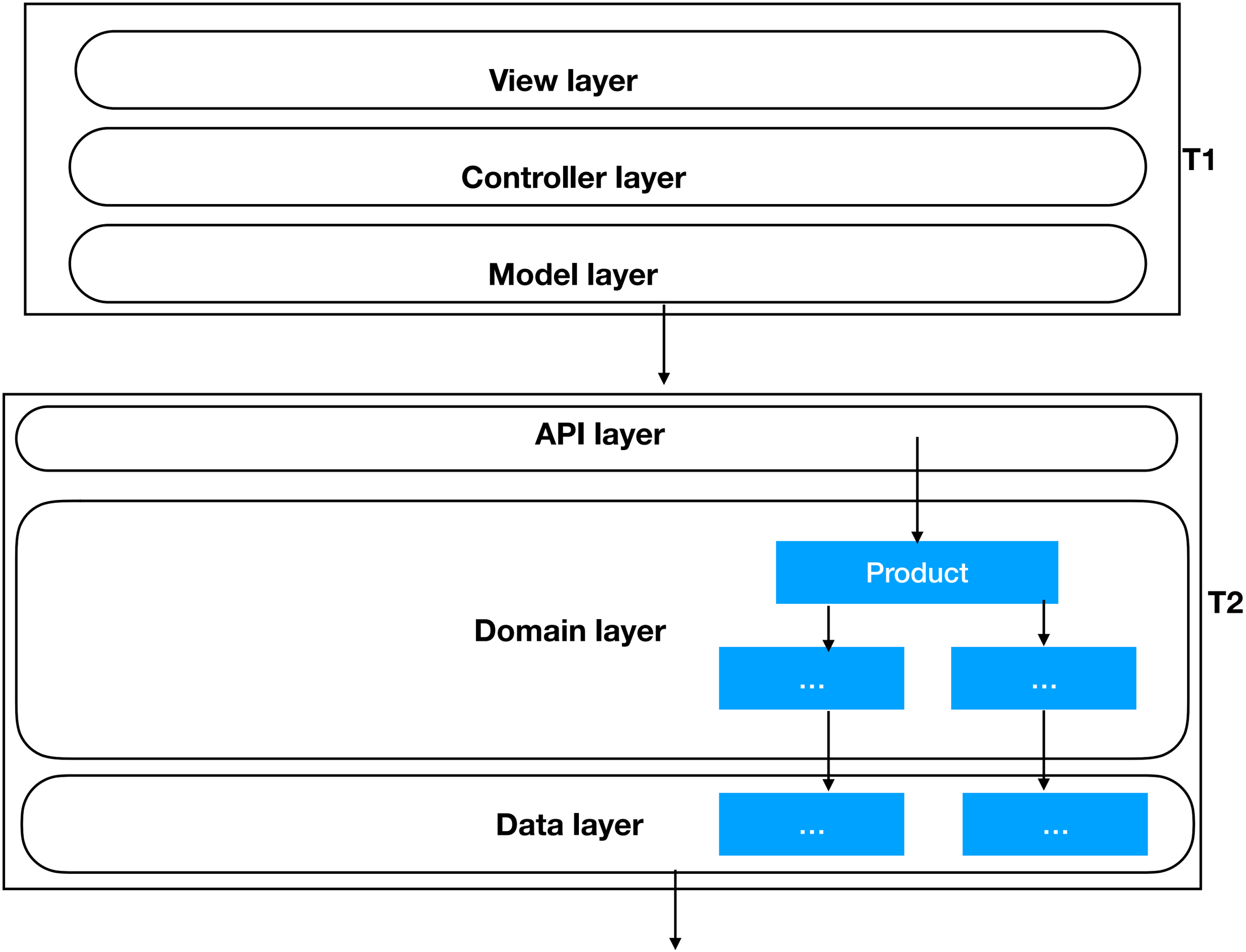
Multi dispatch

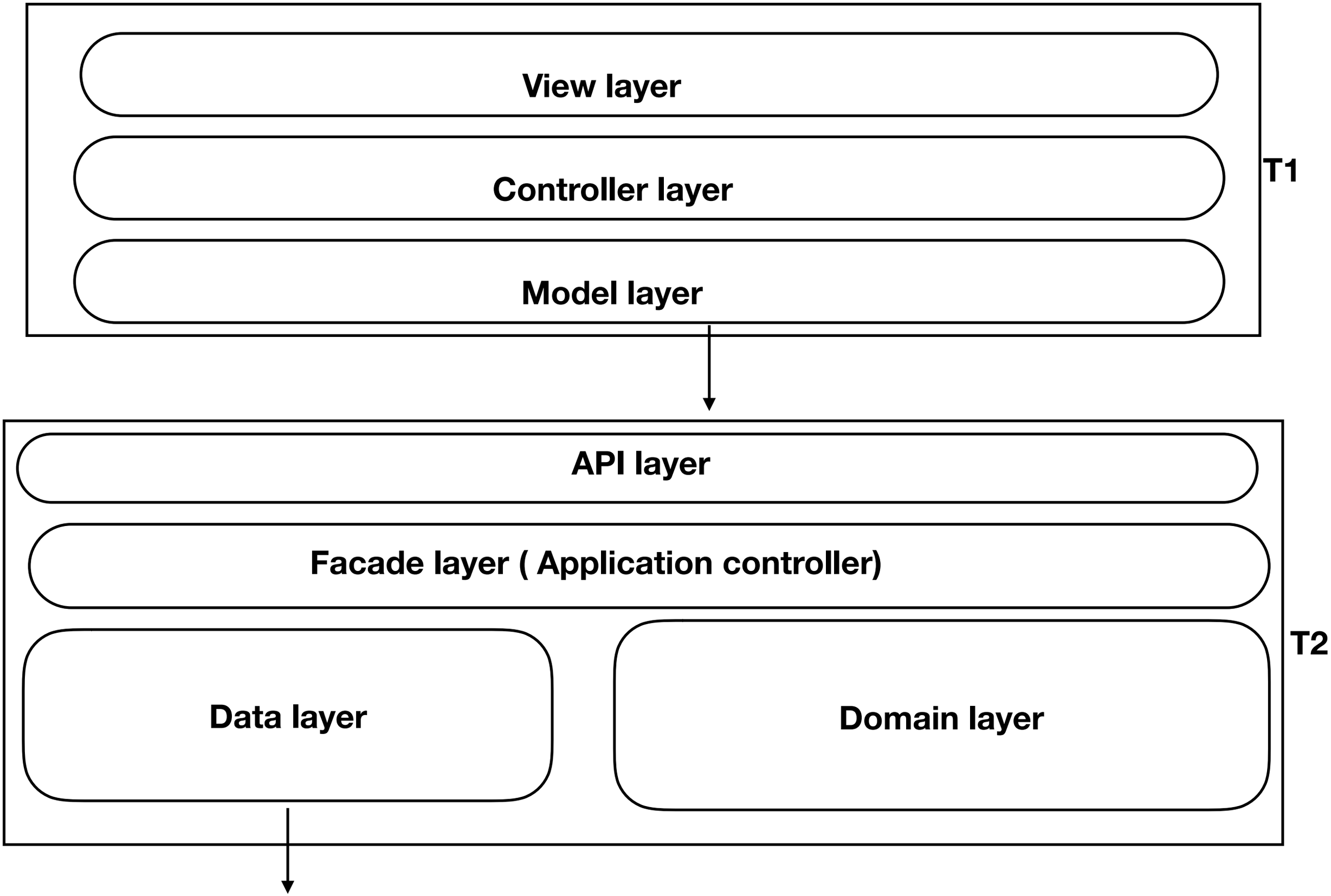


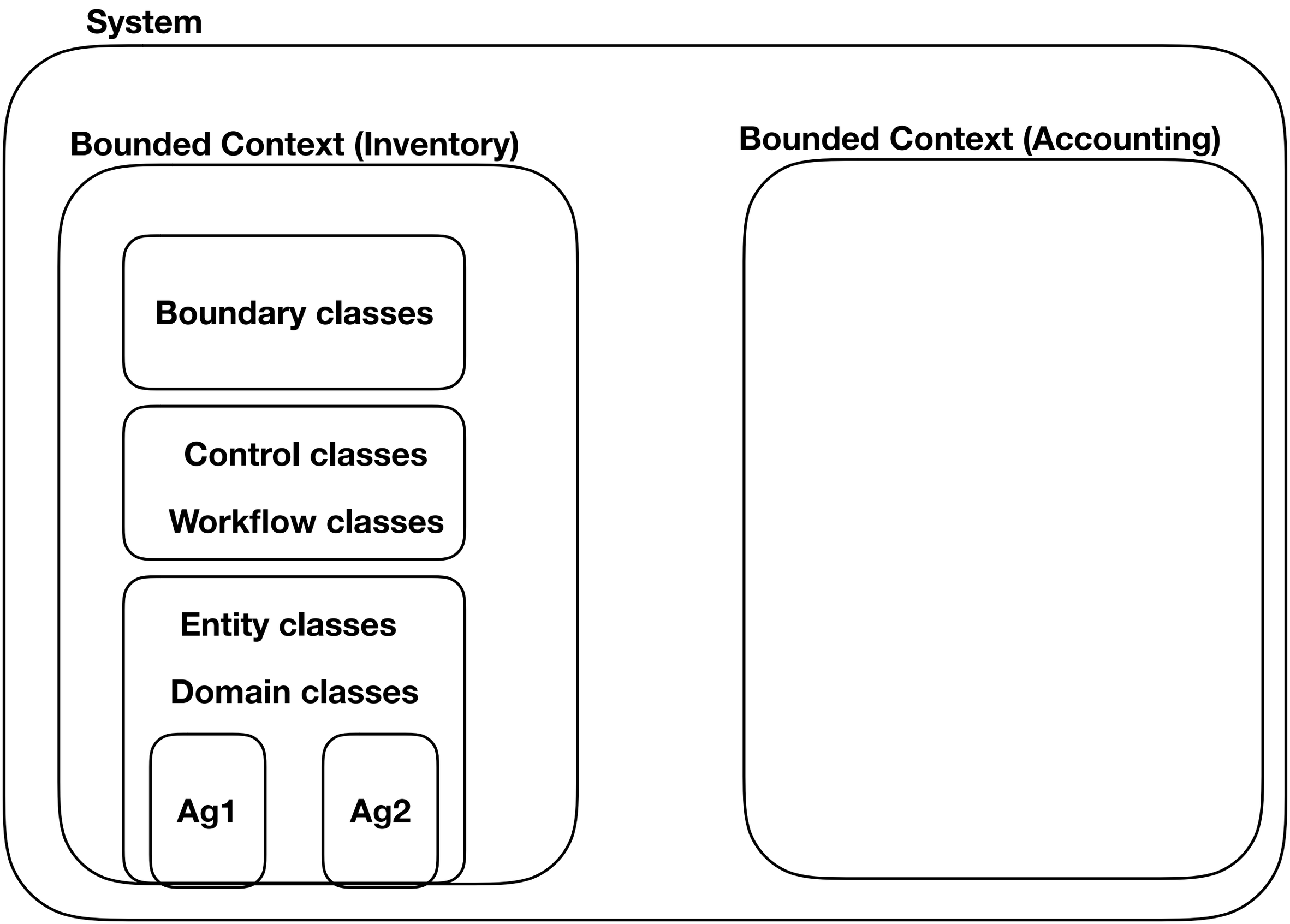

Boundary classes (delegate)
Ui
Rest API
Message
Time
Fun Api (dll)
Control classes (orchestration)
Domain/Entity classes (logic)
Product
Control
...
Db
Cache
Proxy
Message
Mail

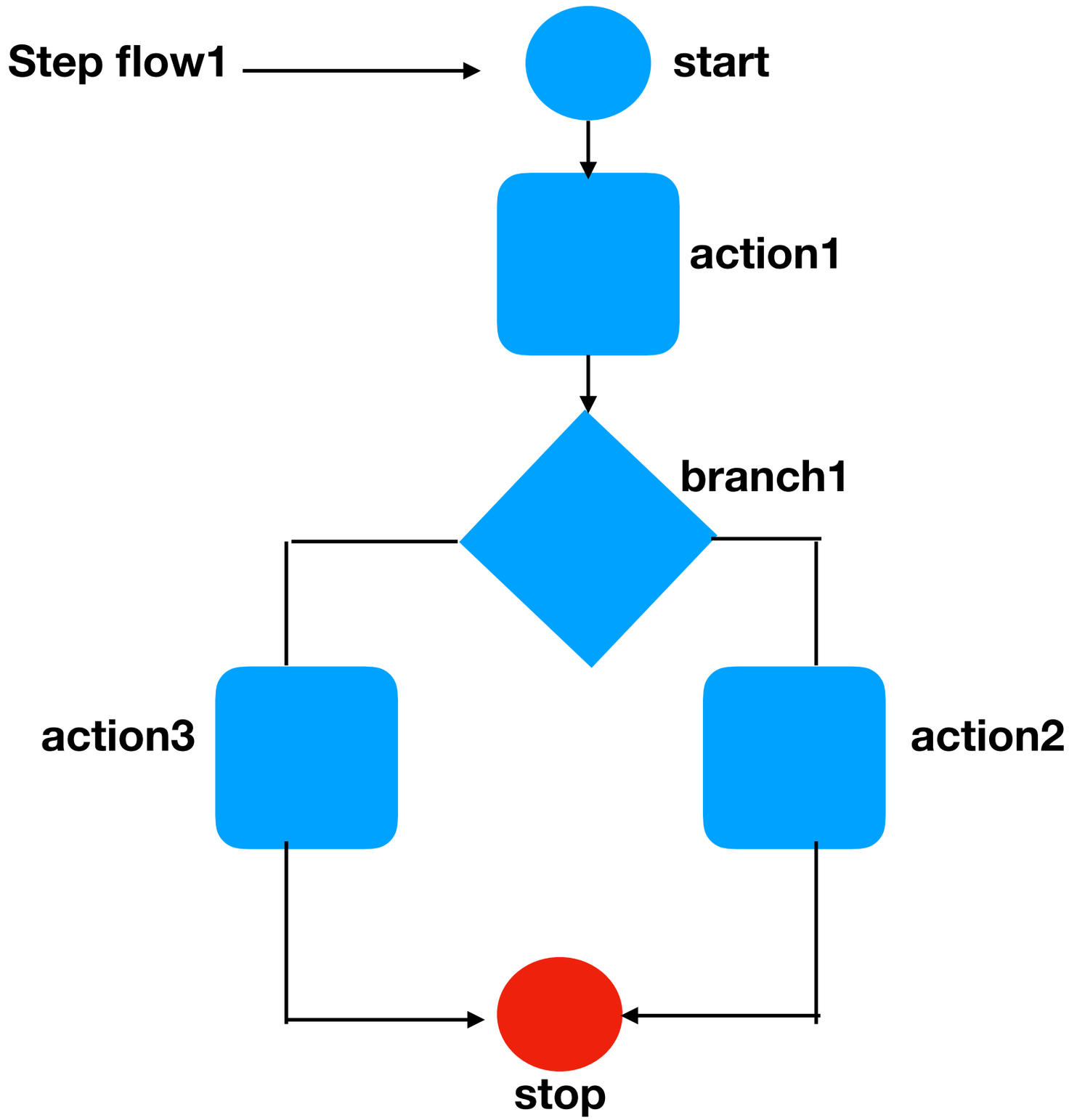

Boundary classes (delegate)
Ui
Rest API
Message
Time
Fun Api (dll)
Domain/Entity classes
(logic)
Tax
Product
Order
Invoice
...
Db
Cache
Proxy
Message
Mail


"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

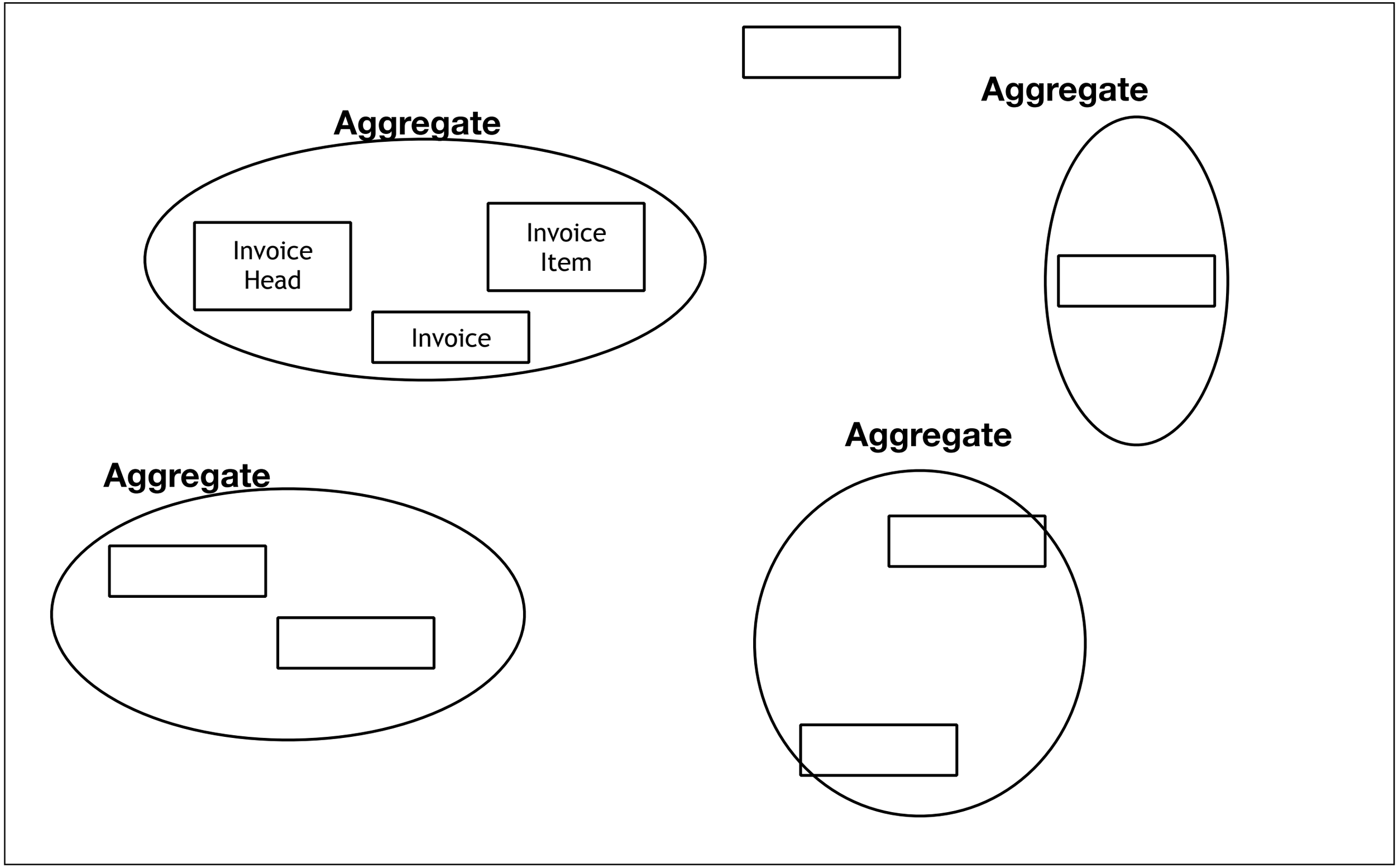

View layer
Controller layer
Model layer
T1
API layer
Product
Domain layer
...
...
T2
Data layer
...
...

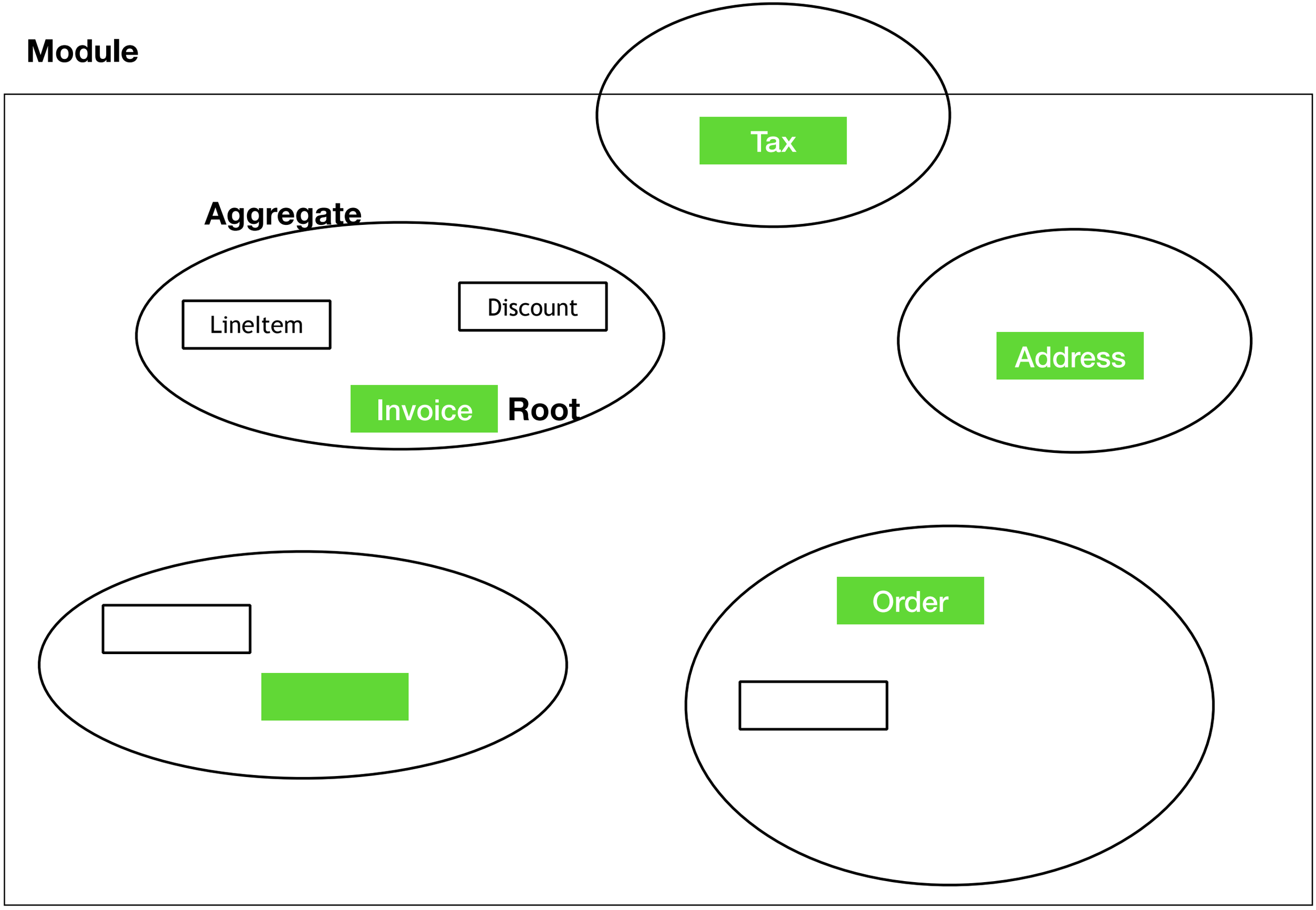

View layer
Controller layer
Model layer
T1
API layer
Facade layer (Application controller)
Data layer
Domain layer
T2


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

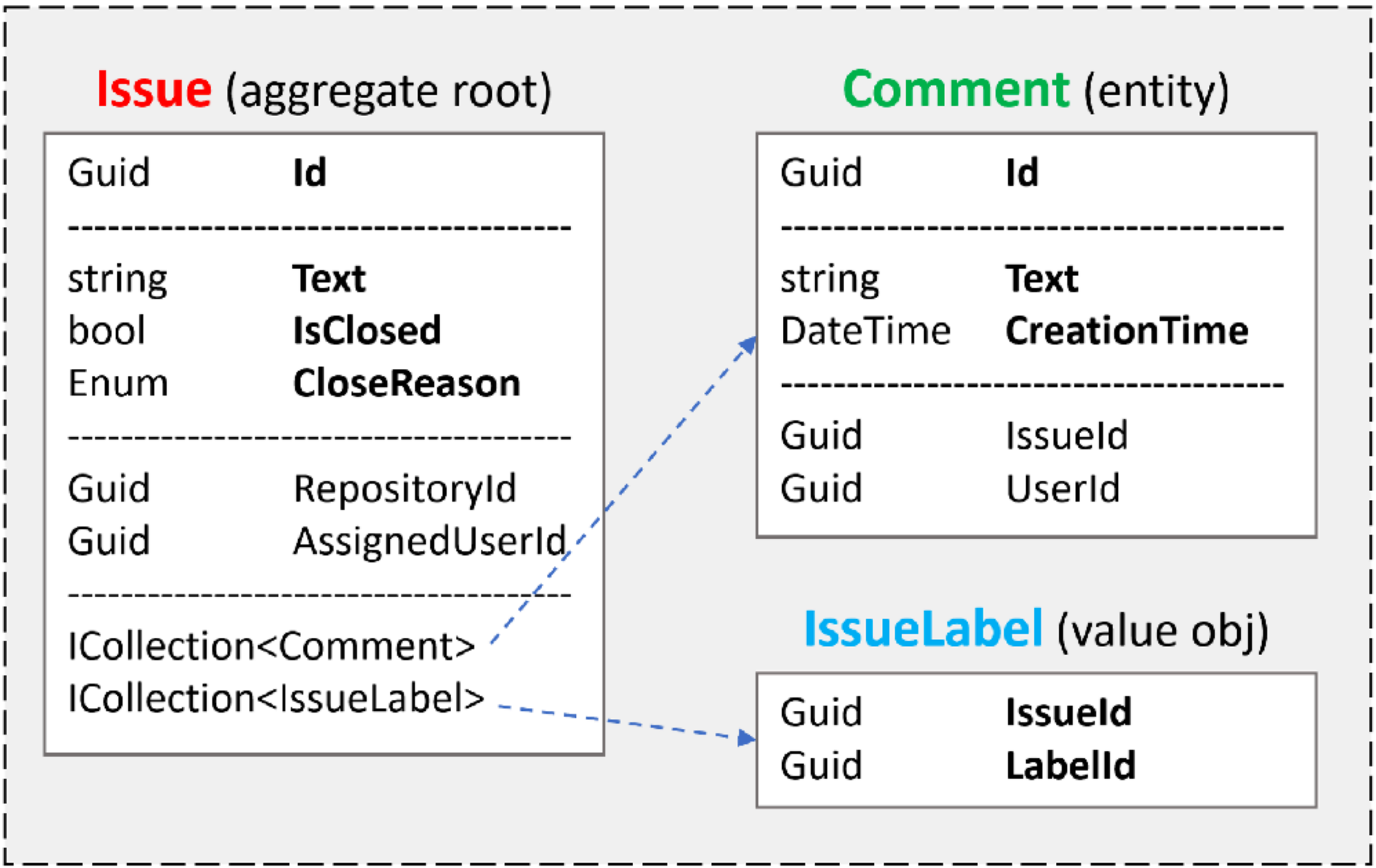

Step flow1
start
action1
branch1
action3
action2
stop

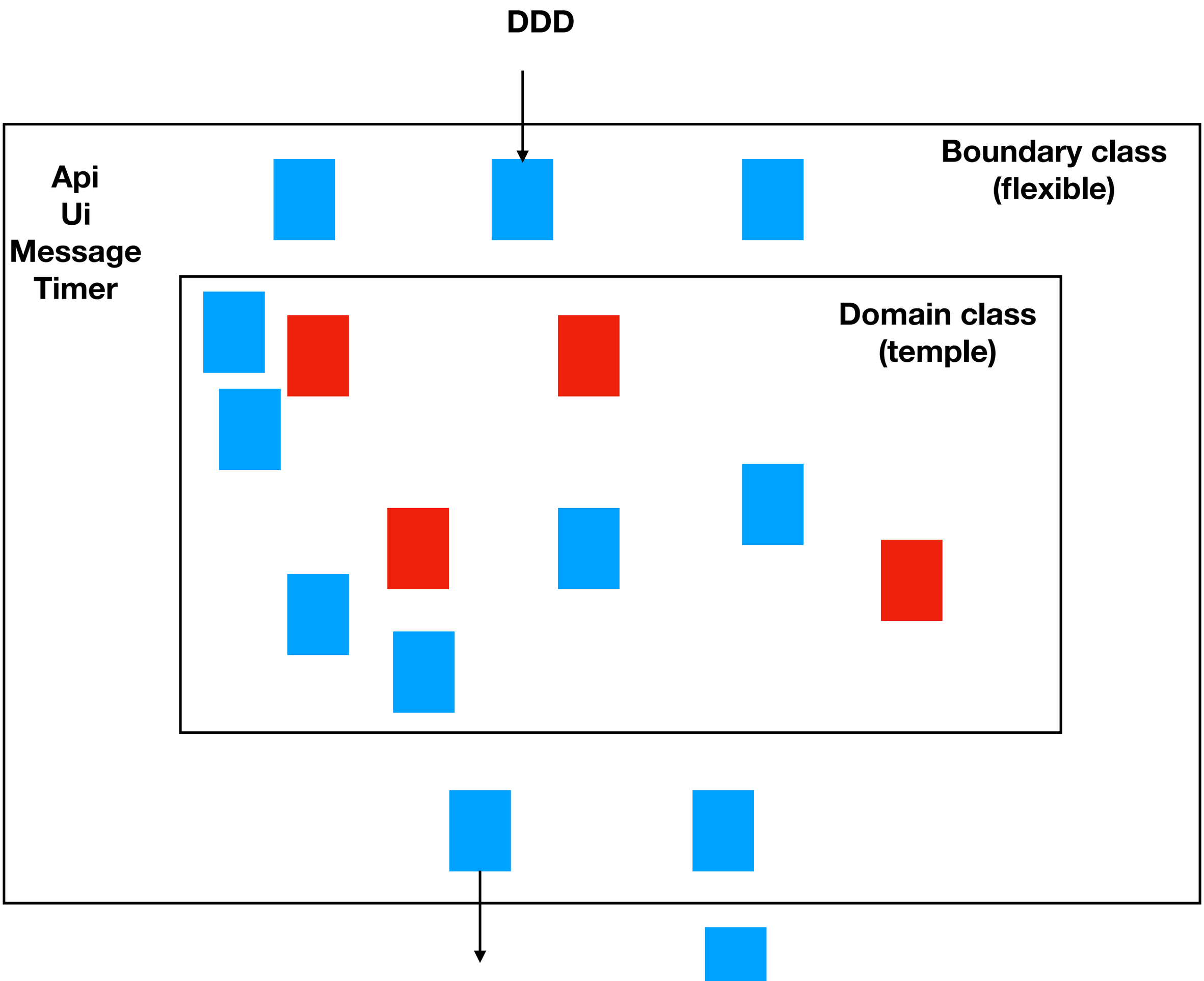
DDD

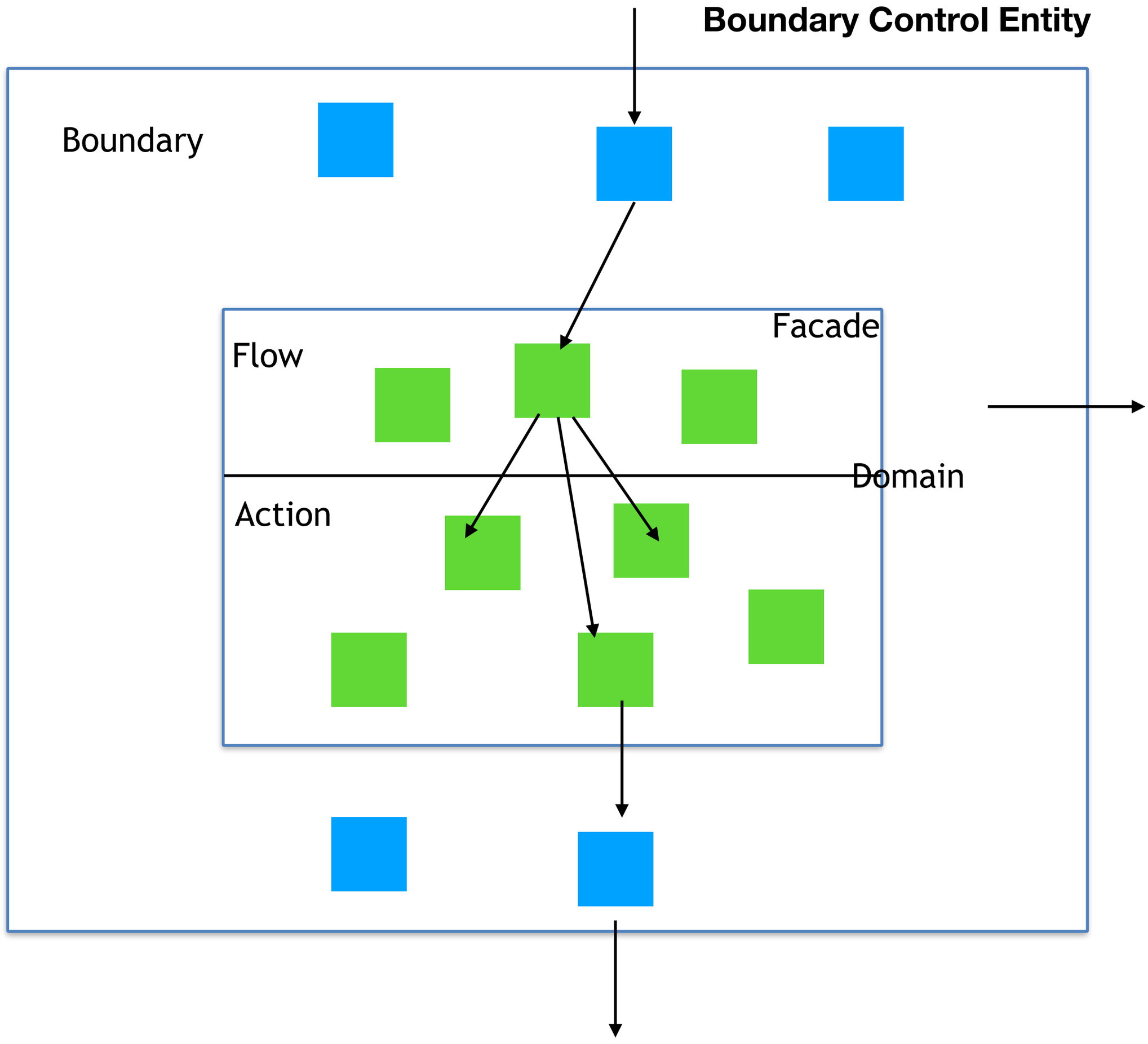

Aggregate
Invoice Head
Invoice Item
Invoice
Aggregate
Aggregate
Aggregate

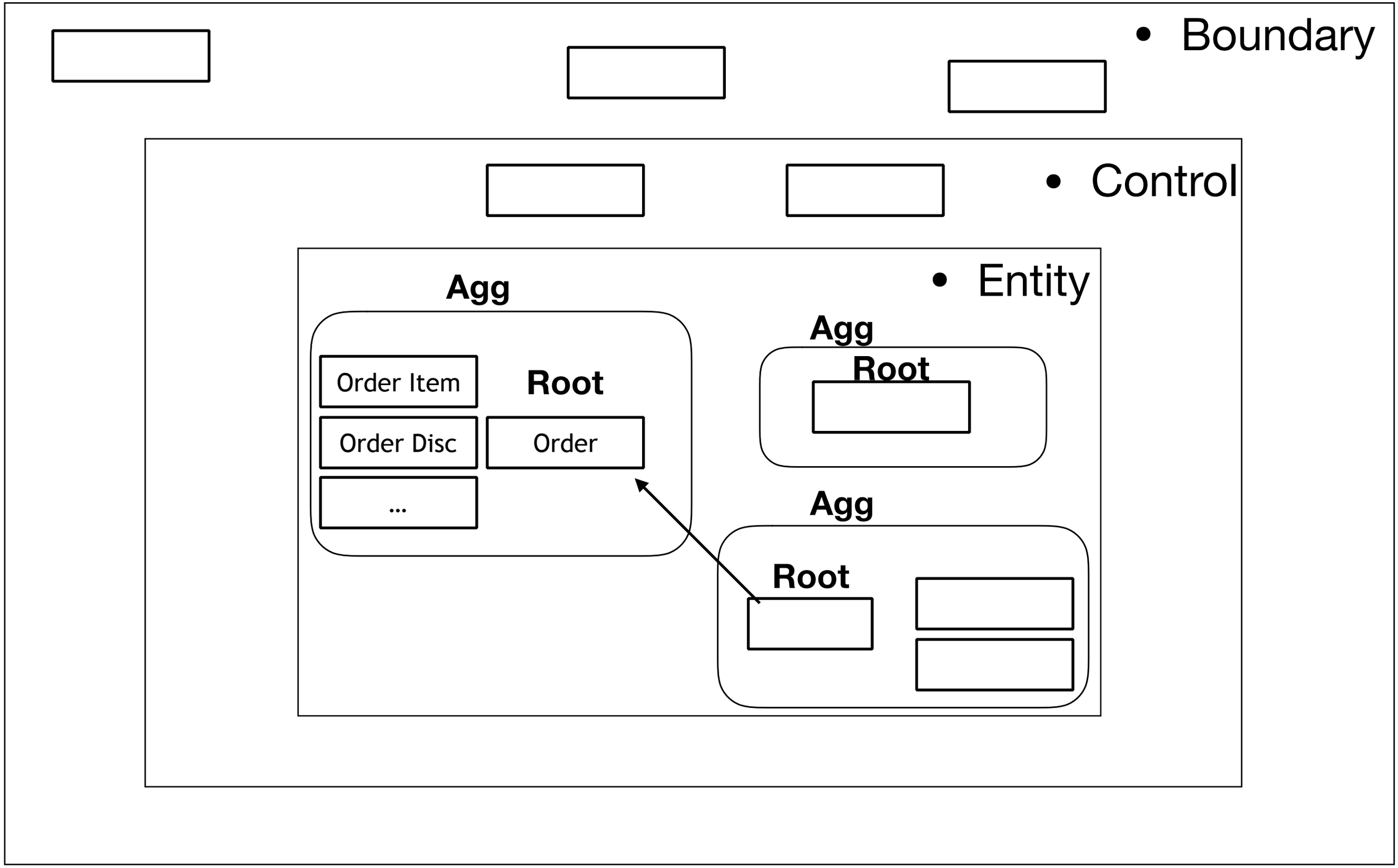

Module
Tax
Aggregate
LineItem
Discount
Invoice
Root
Address
Order

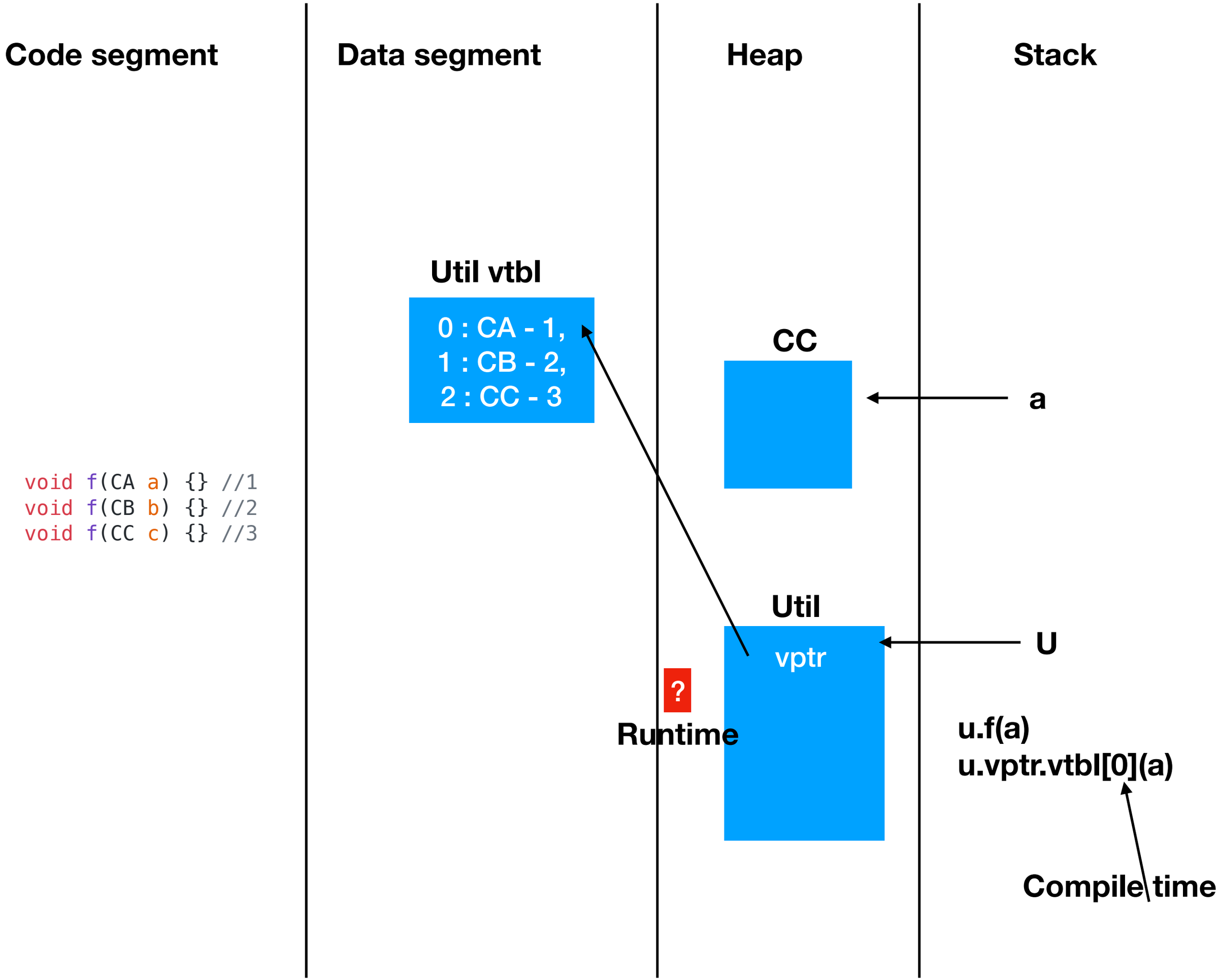
Issue Aggregate

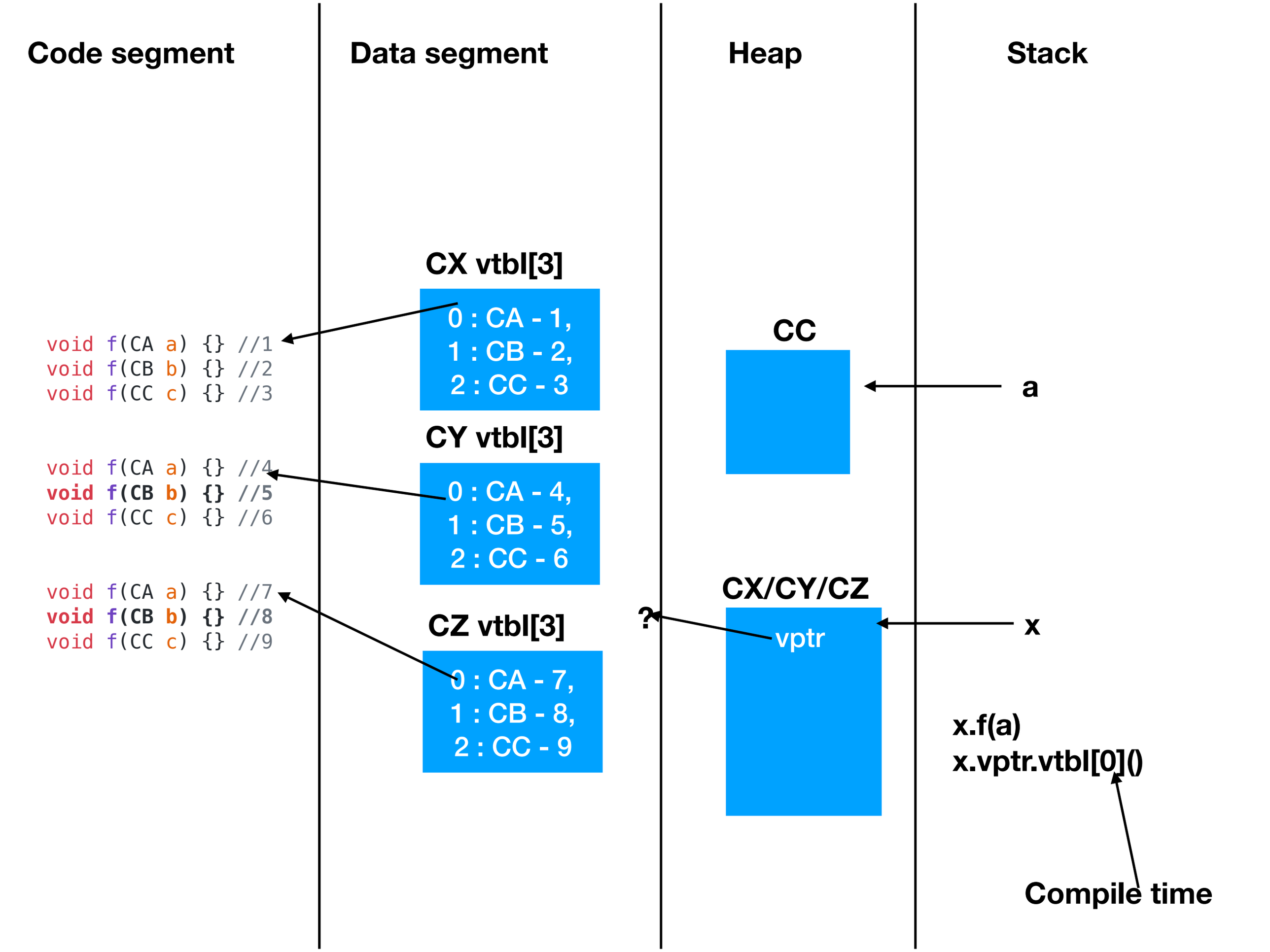


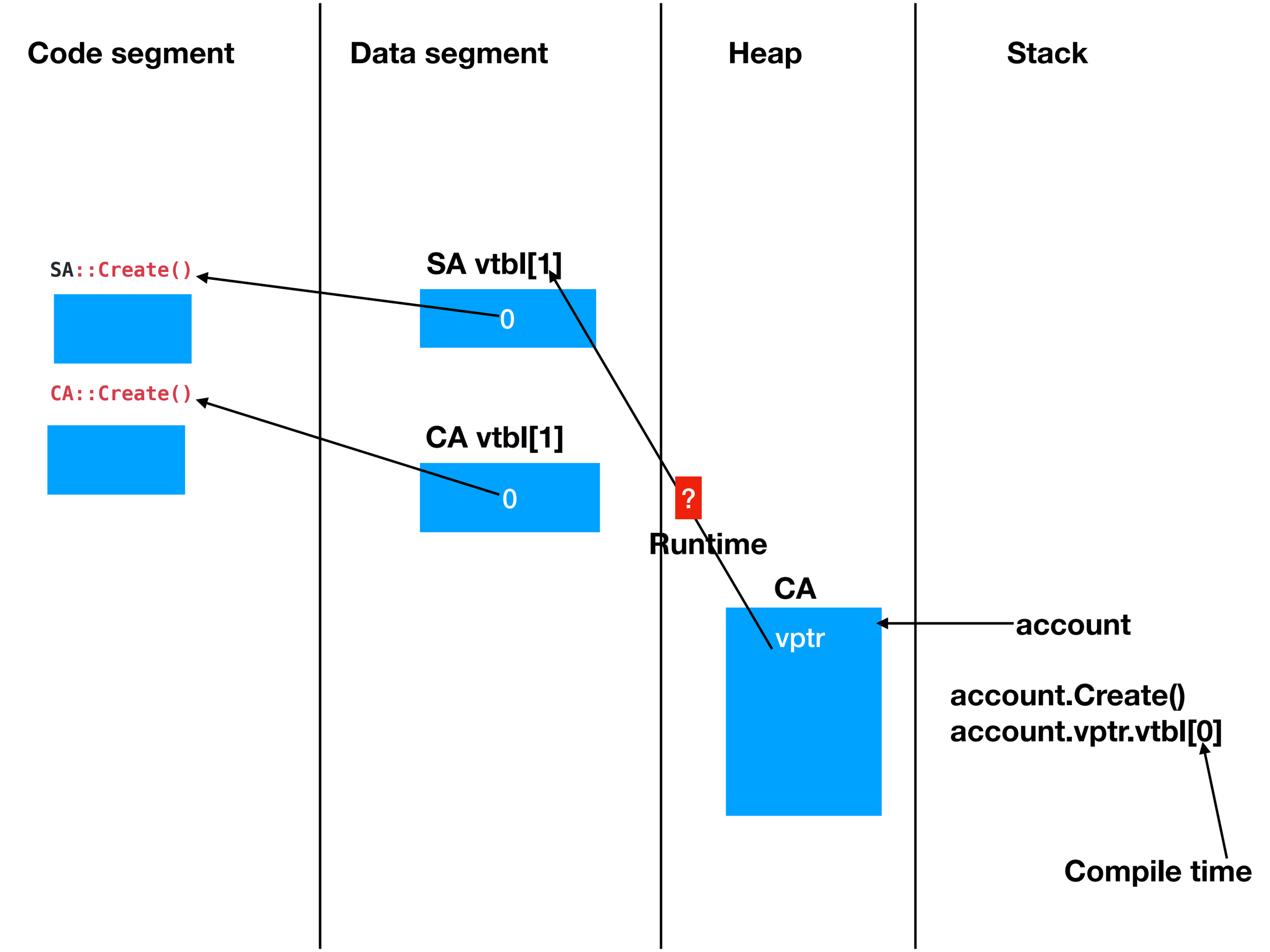

DDD
Boundary class
(flexible)
Api
Ui
Message
Timer
Domain class
(temple)

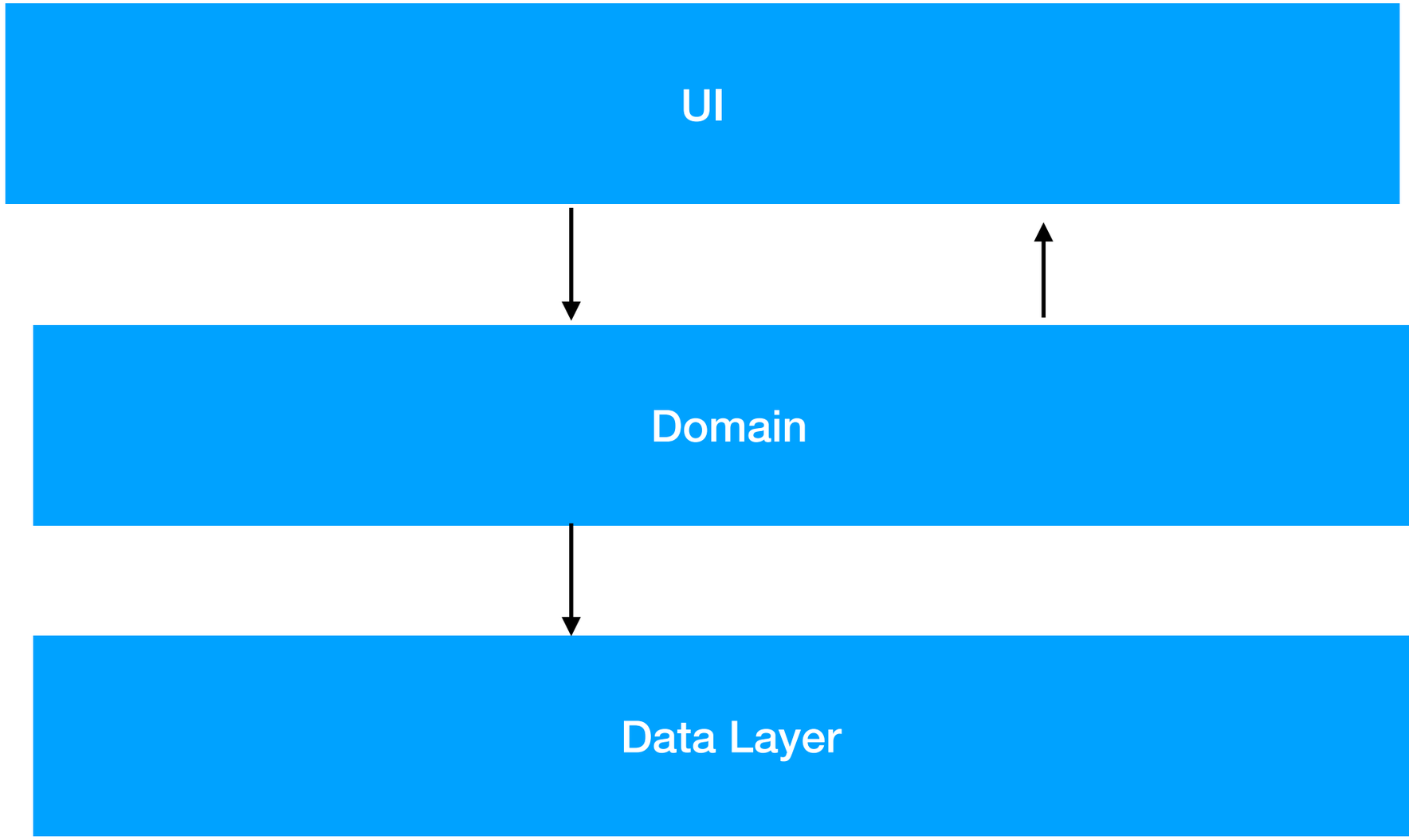

Boundary Control Entity
Boundary
Facade
Flow
Domain
Action

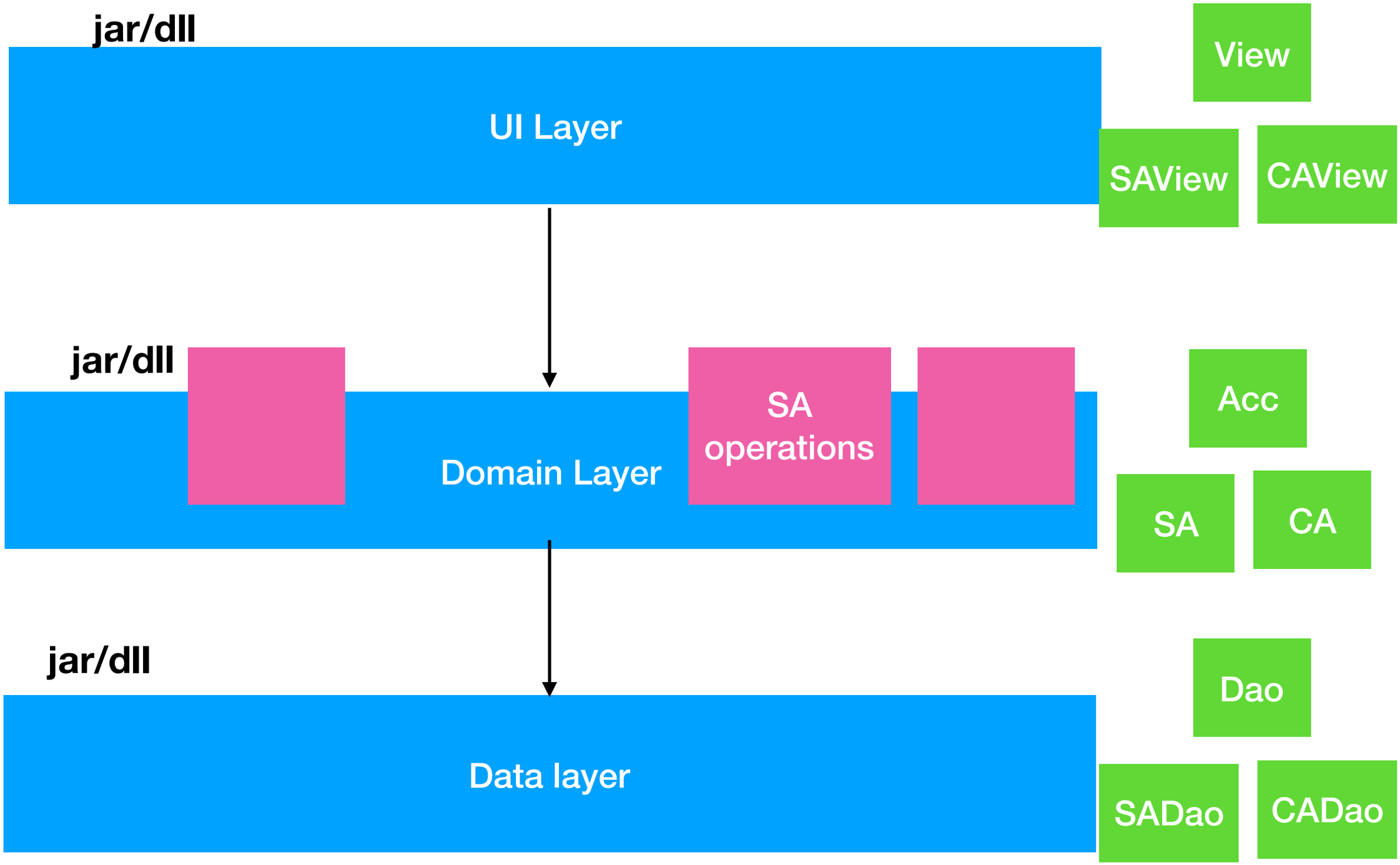

Boundary
Control
Entity
Agg
Order Item
Root
Order Disc
Order
...
Agg
Root
Agg
Root


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
CX/CY/CZ
?
vptr
x
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


UI
Domain
Data Layer


jar/dll
UI Layer
View
SAView
CAView
jar/dll
Domain Layer
SA
operations
Acc
SA
CA
jar/dll
Data layer
Dao
SADao
CADao