



Mubarak



- » 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
- Concurrency design
- Expectations
- Years of Exp
- Technology stack

Review of code

class is having multiple resp like managing game, game logic. We can have separate 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 winning lines and can improve readability

Duplicate code in printBoard

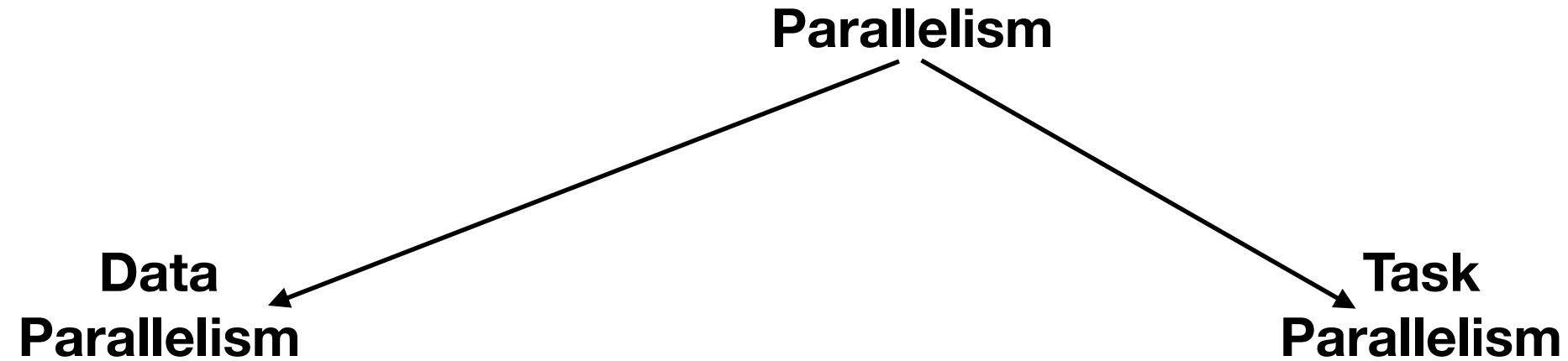
Good

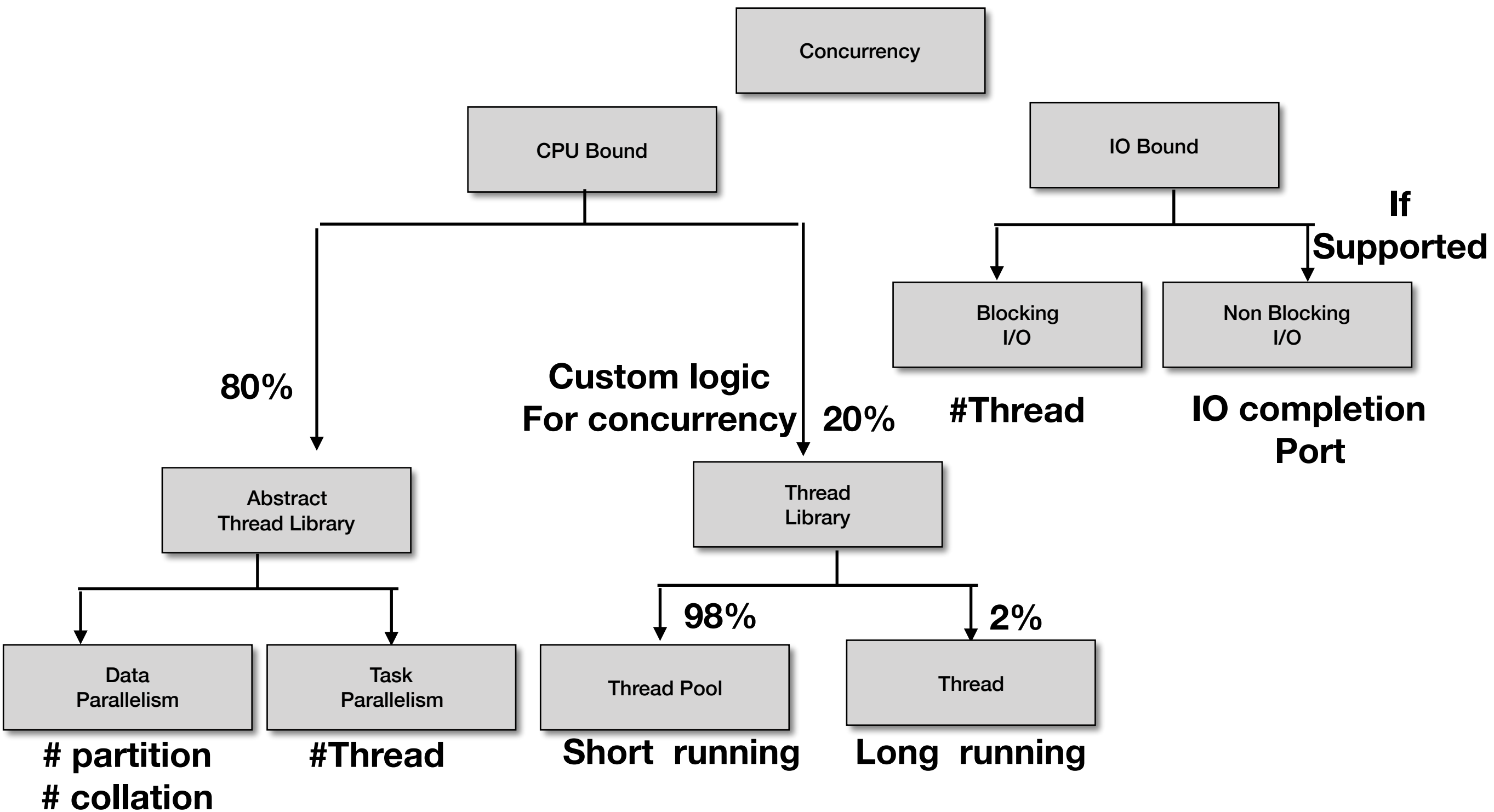
- SRP (***)
- Low coupling (***)
- Unit testability
- LSP
- ISP
- Upcasting/abstraction
- DRY
-
- Prefer composition over inheritance
- Boundary control entity (*)
- YAGNI
- KISS
- Program to an Interface
- DDD
 - Aggregates

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)

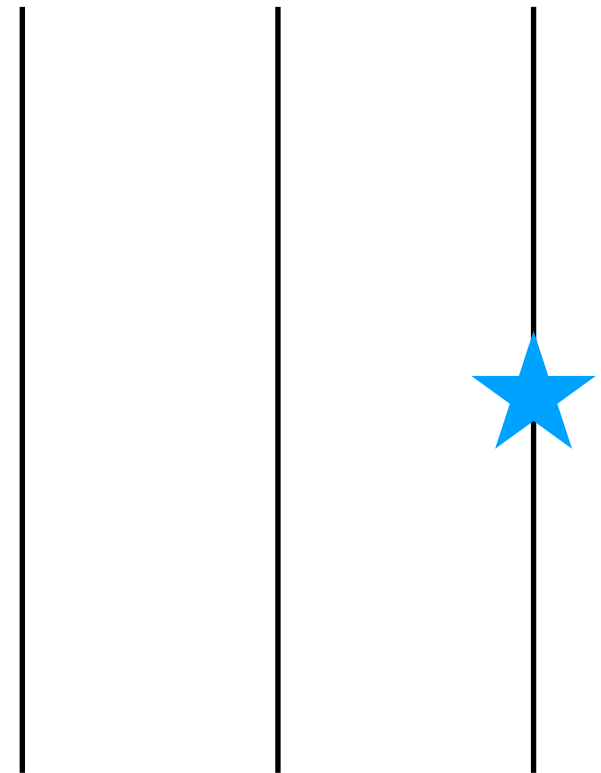
Good (concurrency)





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

Only Data changes
Logic remains same
In each path

==

Logic changes
In each Path

Error
Handling logic

Domain
Rules

if(error == true)

if(salary > 5000)

Templates/
Generics

One class
Multiple Objects
for each change

Interface/ Duck

Exception

Specification
Pattern

Functional
Interface

Function
Object

If Flag



Only Data Type changes
Logic remains same
In each Path

Only Data changes
Logic remains same
In each path

Error
Handling logic

Domain
Rules

==

Logic changes
In each Path

if(error == true)

if(salary > 5000)

Templates/
Generics

One class
Multiple Objects
for each change

Exception

Specification
Pattern



Functional
Interface

Interface/ Duck

Function
Object

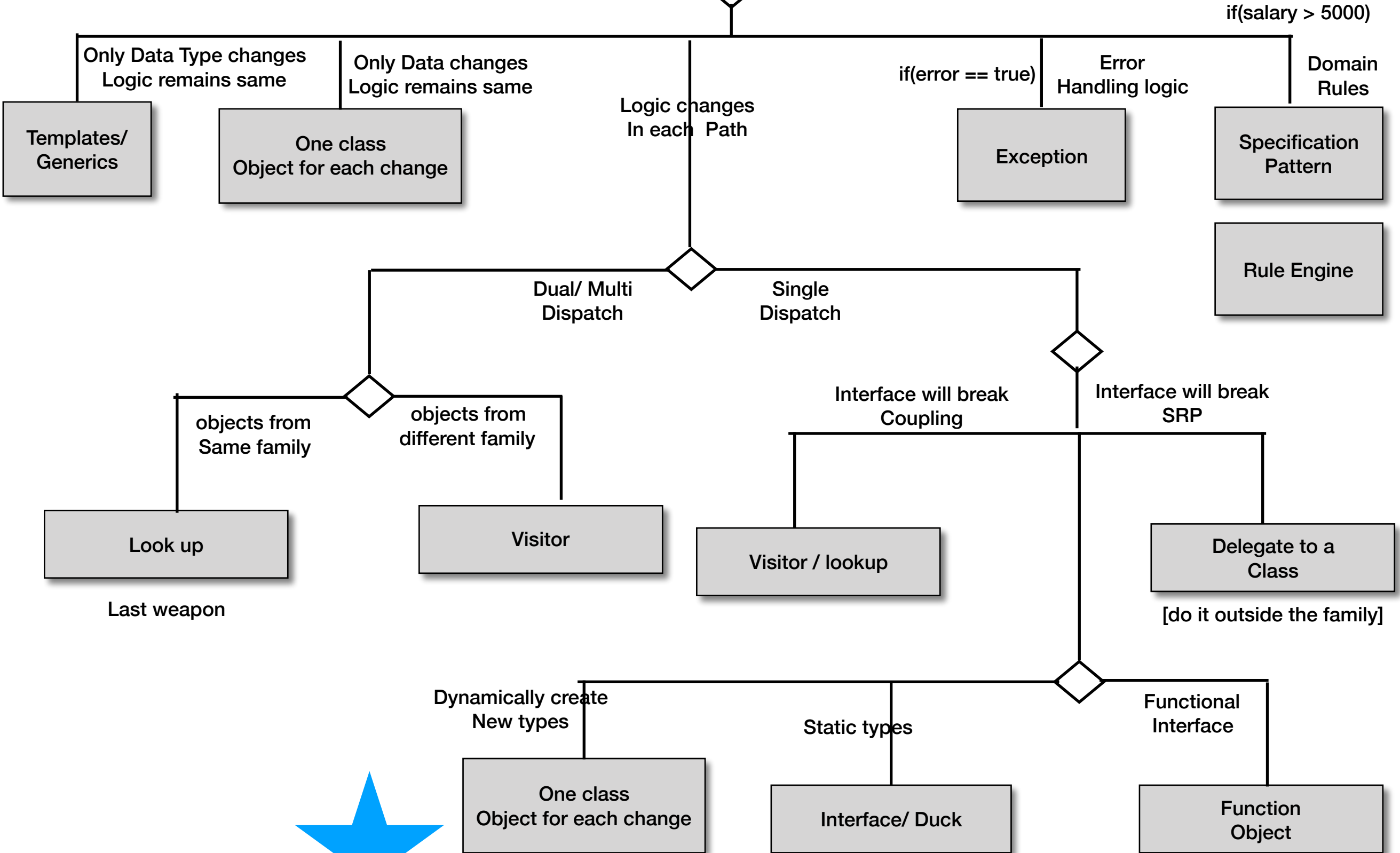
Interface will break
SRP

Delegate to a
Class

[do it outside the family]

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

If Flag ==



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

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


	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

Architecture vs Design

Architecture [Design] vs [Code] Design

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

readability

unit test

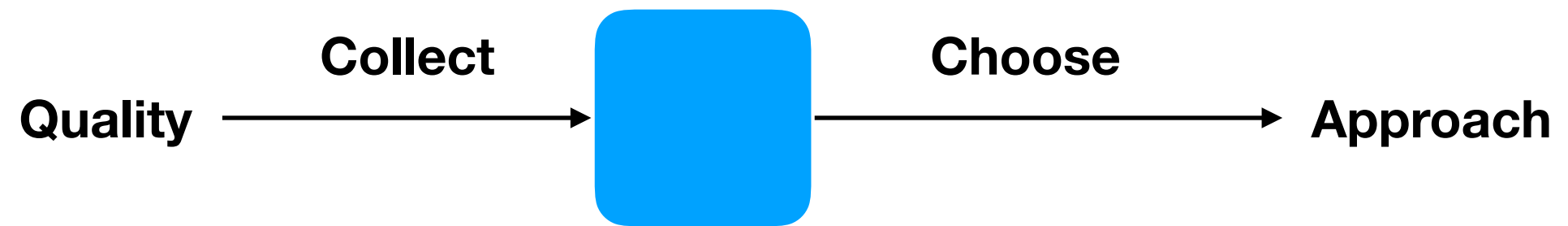
OCP

1

**Flag => Polymorphism/
Abstraction/
Interface**

**Coupling => Polymorphism/
Abstraction/
Interface**

**Down casting => Polymorphism/
Abstraction/
Interface**



“system quality”

Domain

Understands

Collect

Choose

Approach

Which : Qualities

How much: Measure

<< Architecture >>

Knows

Architecture design

Blue print

HLD

System Design

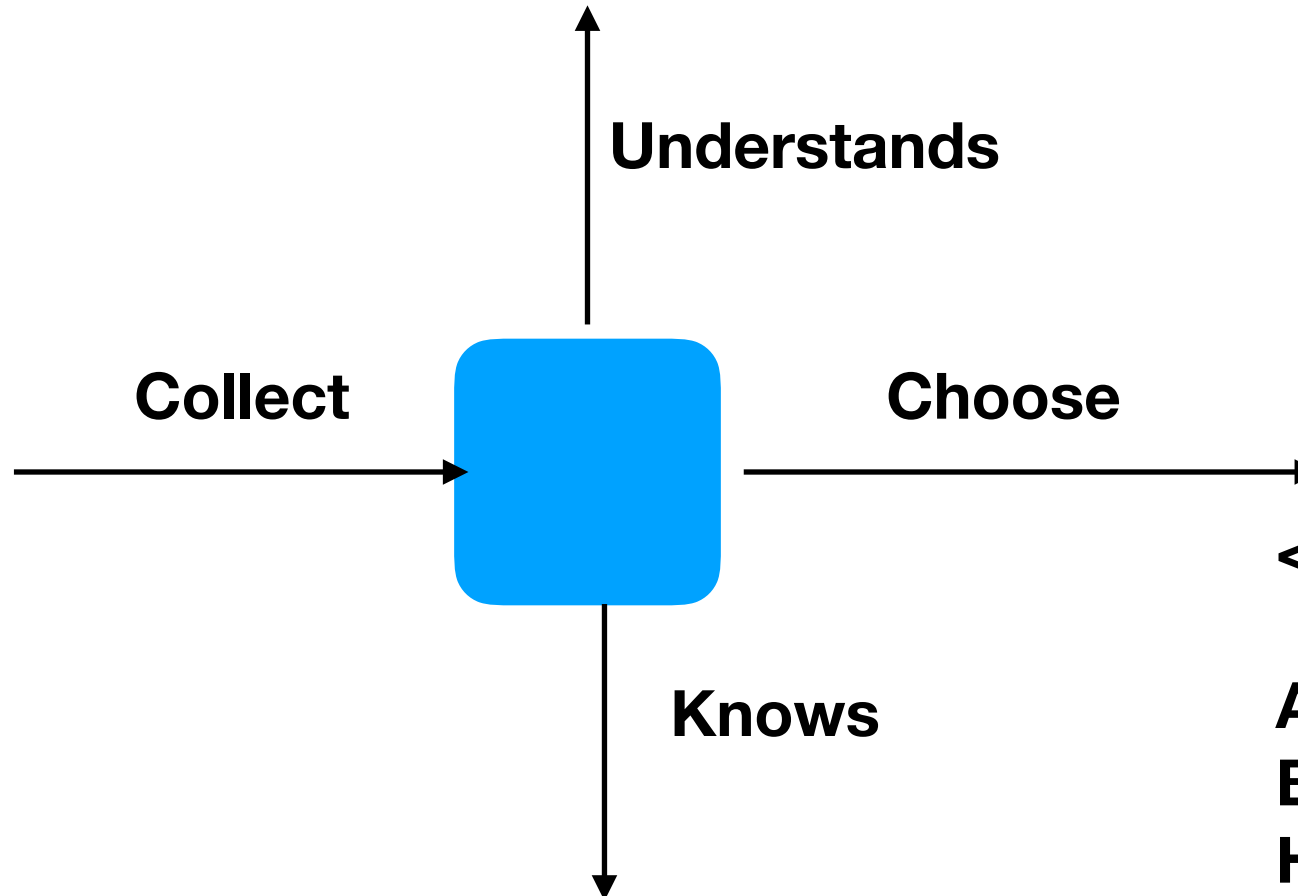
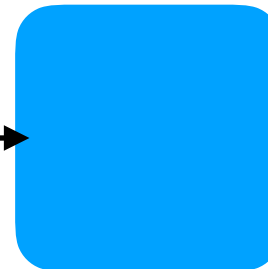
...

Architectural patterns, styles, tactics

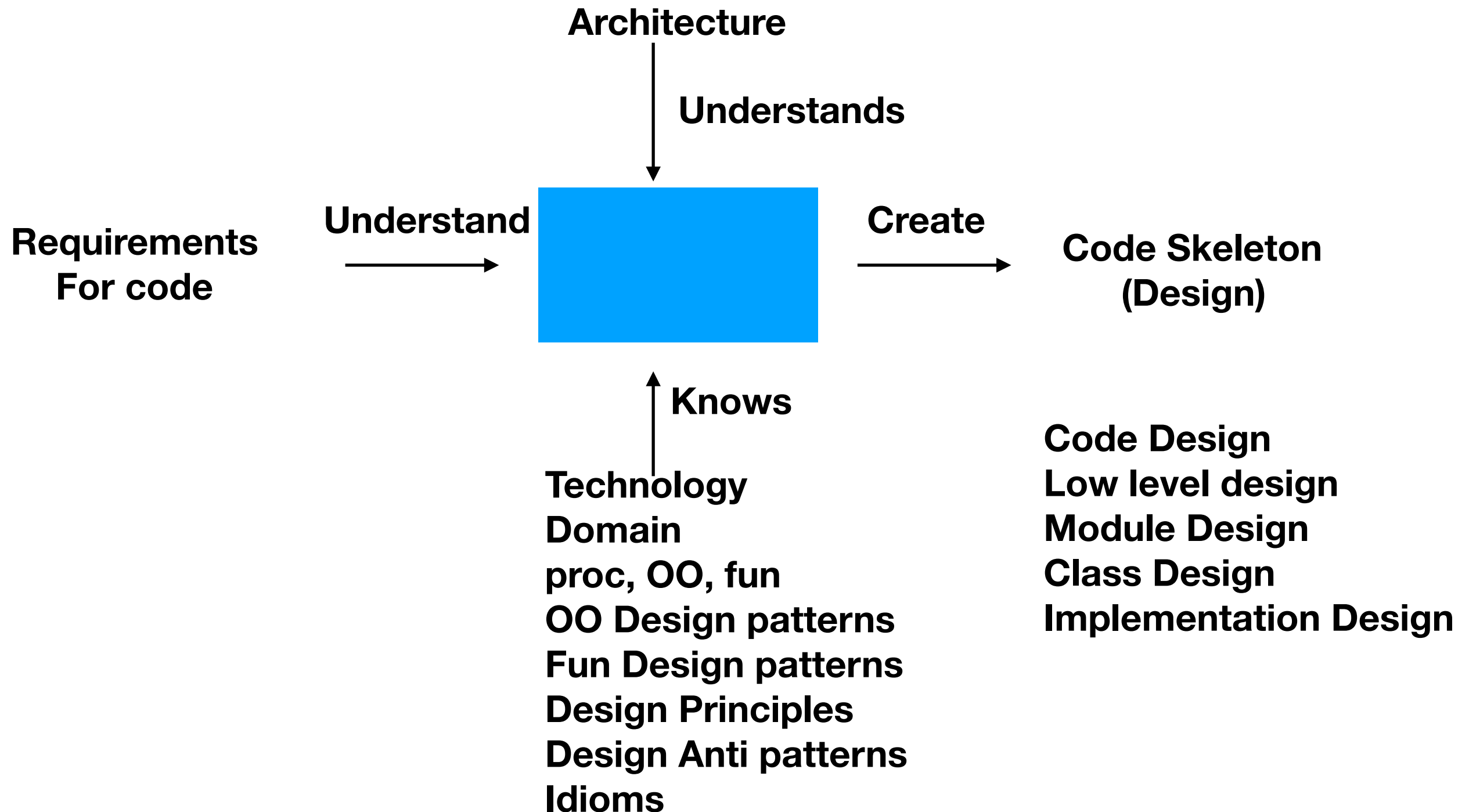
Reference architecture,

Architectural anti patterns,

Technology, domain, ...



Code Maintainability



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

```
fun(Bird bird){  
    ....  
}
```

```
Interface LivingThing{  
}
```

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

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

```
fun(Bird bird){  
  
    if(....)  
        bird.fly();  
    ....  
}
```

```
Class Parrot{  
}
```

```
Interface Bird{  
}
```

```
interface LivingThing{
```

```
    ...
```

```
}
```

```
interface Bird extends LivingThing{
```

```
    chirp
```

```
    sound()
```

```
}
```

```
Interface FlyingBird extends Bird{
```

```
    fly()
```

```
}
```

```
Interface NestBuildingBird extends Bird{
```

```
    makeNest()
```

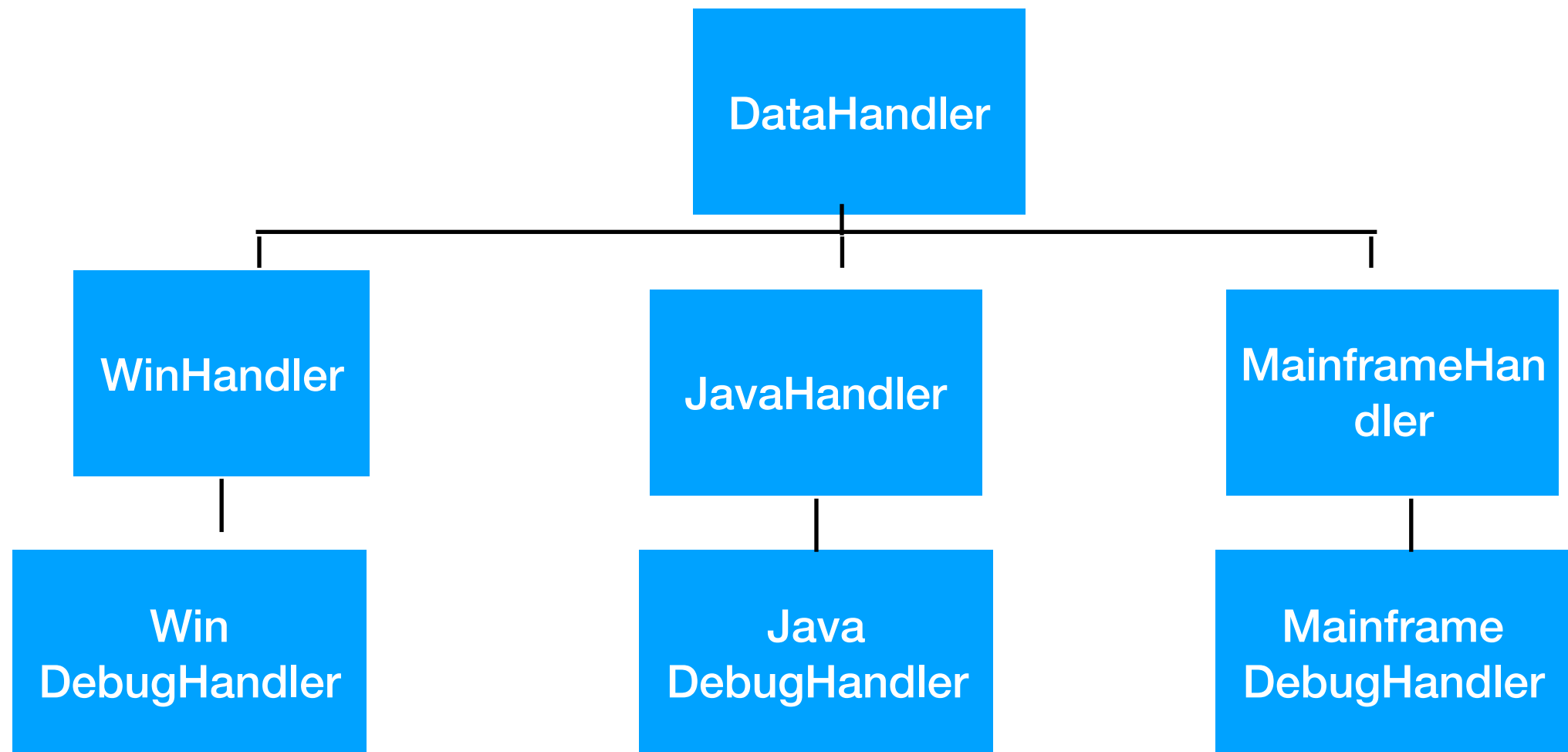
```
}
```

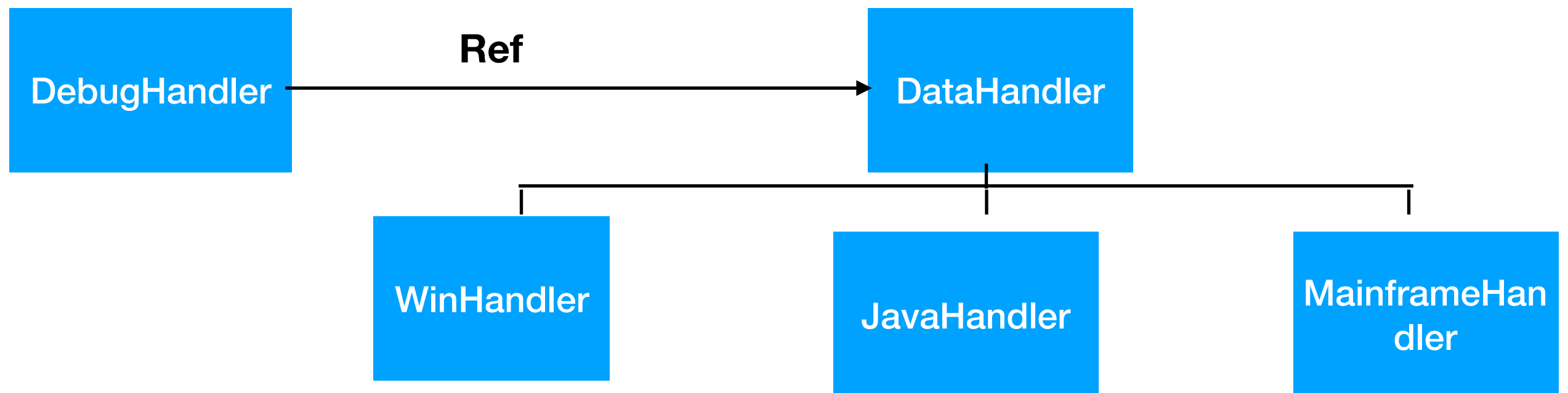
```
...
```

```
    layEggs()
```

```
    swim()
```

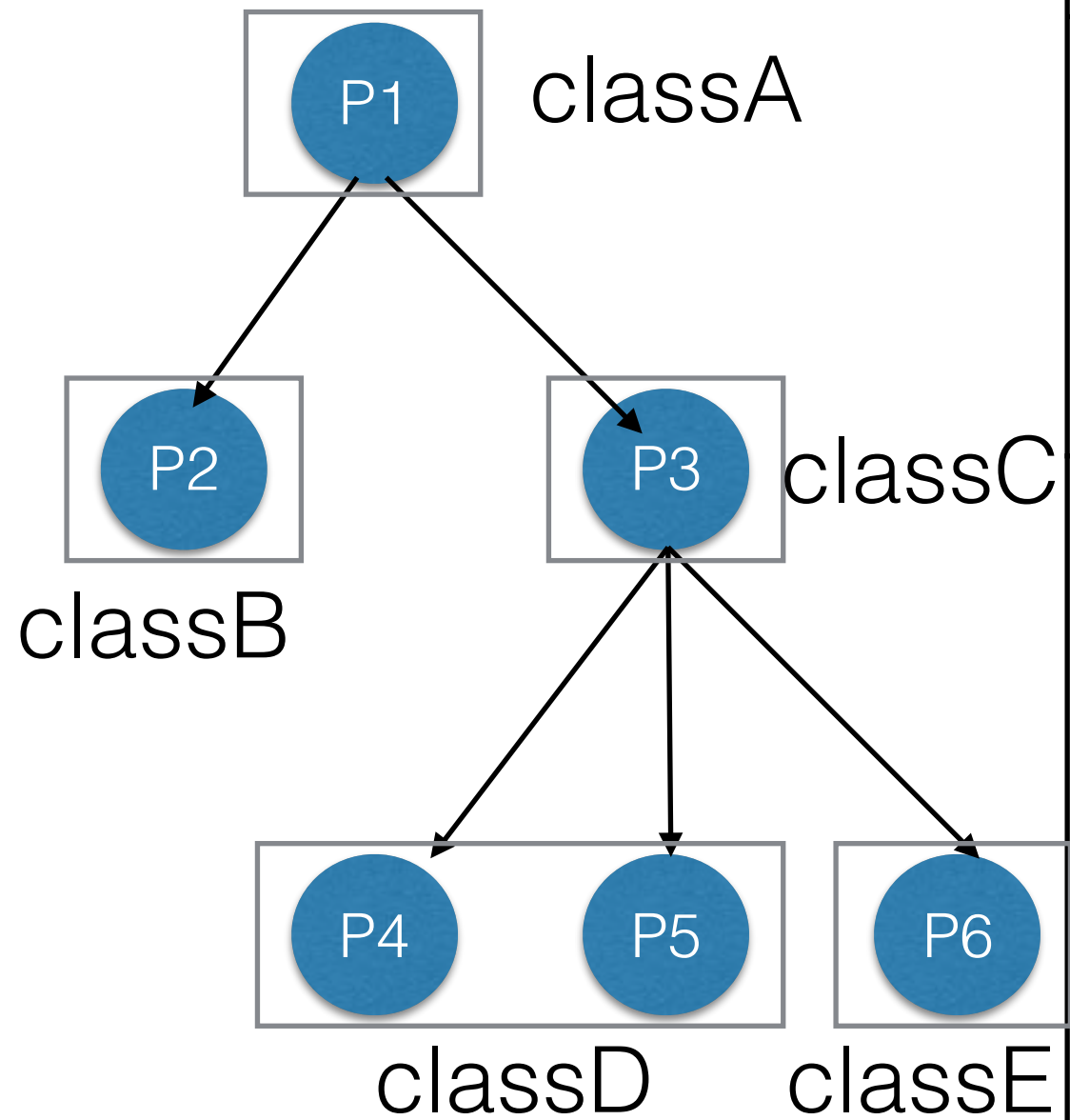
```
}
```



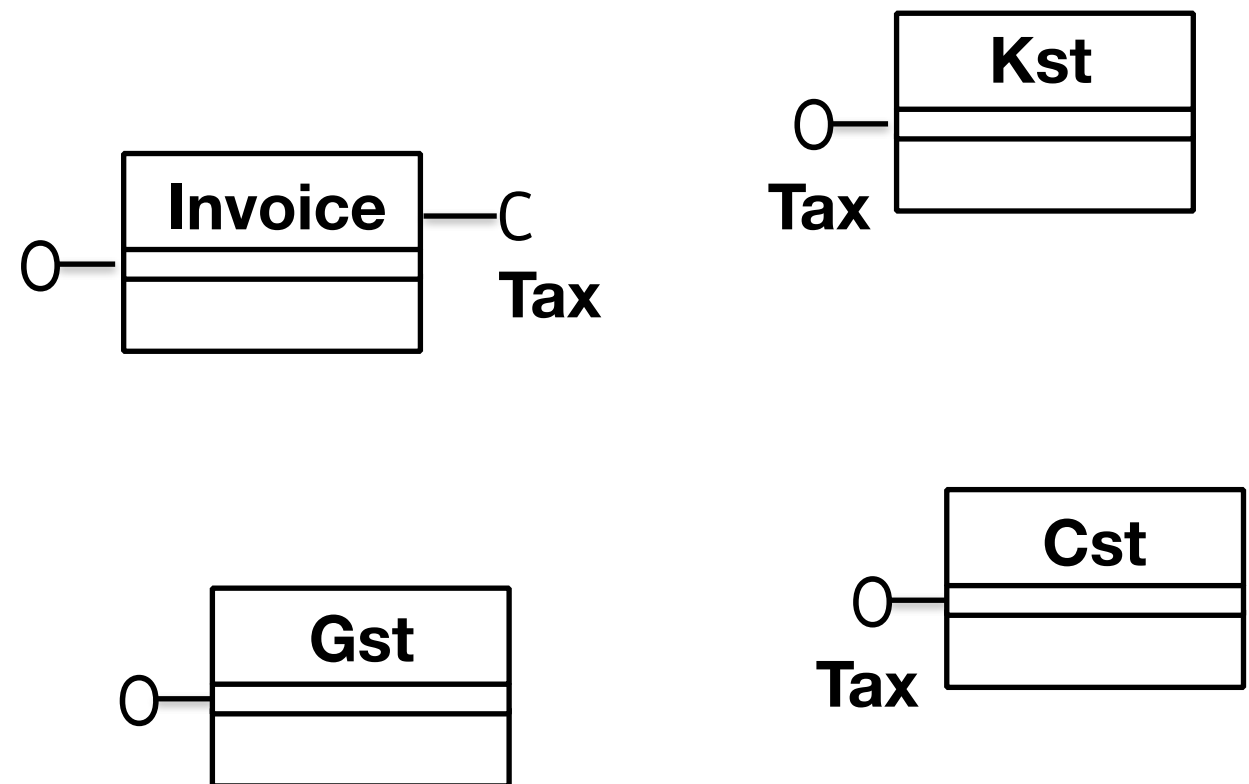
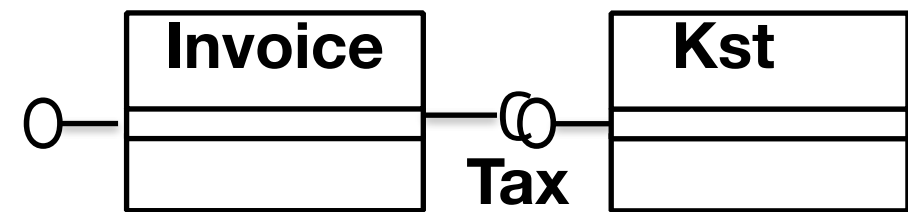



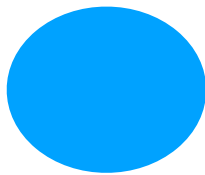
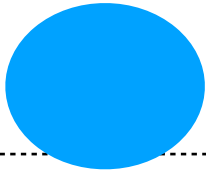
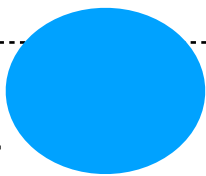
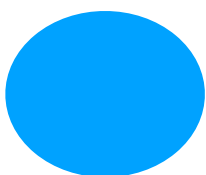

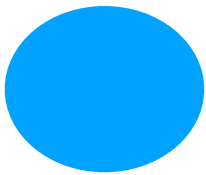
a+b	3 cpu cycles
Fun call	10 cpu cycles
Exception handling	1000 cpu cycles
Create thread	200,000 cpu cycles
Write to file	10,00,000 cpu cycles
Db call	40,00,000 cpu cycles

Procedural Prog (tree)

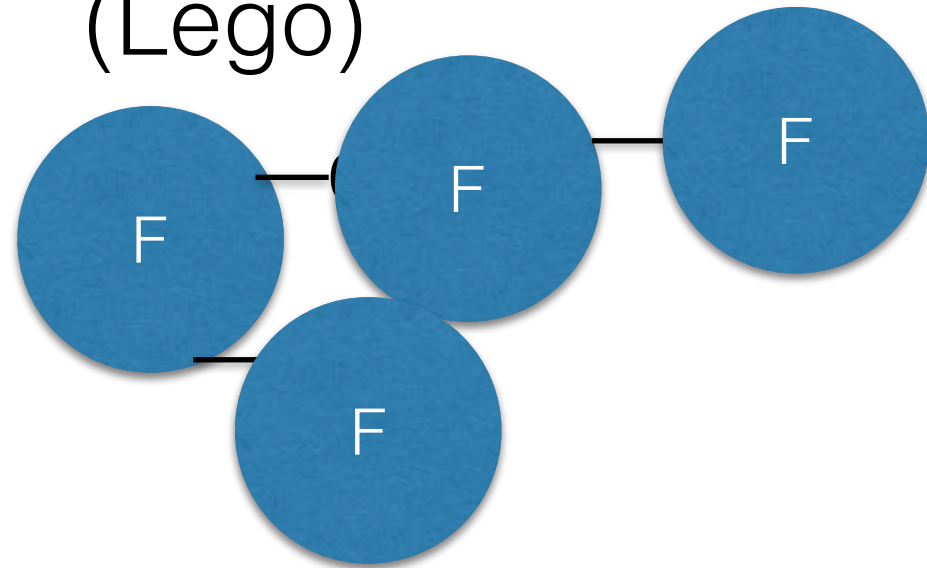


OO Prog (Lego)

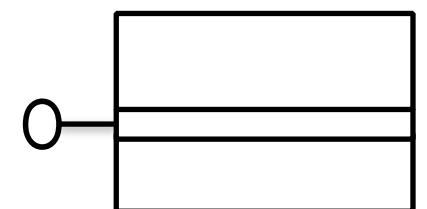
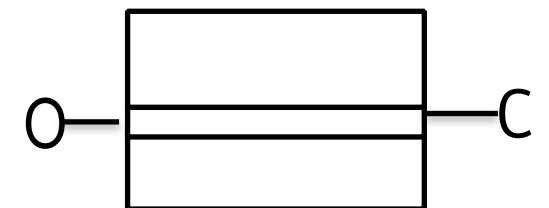
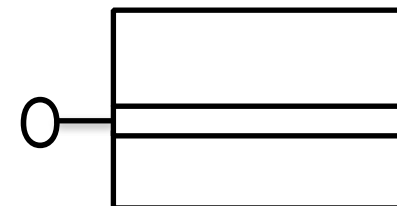
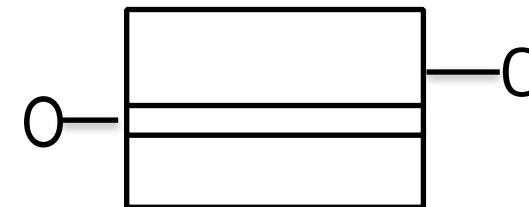
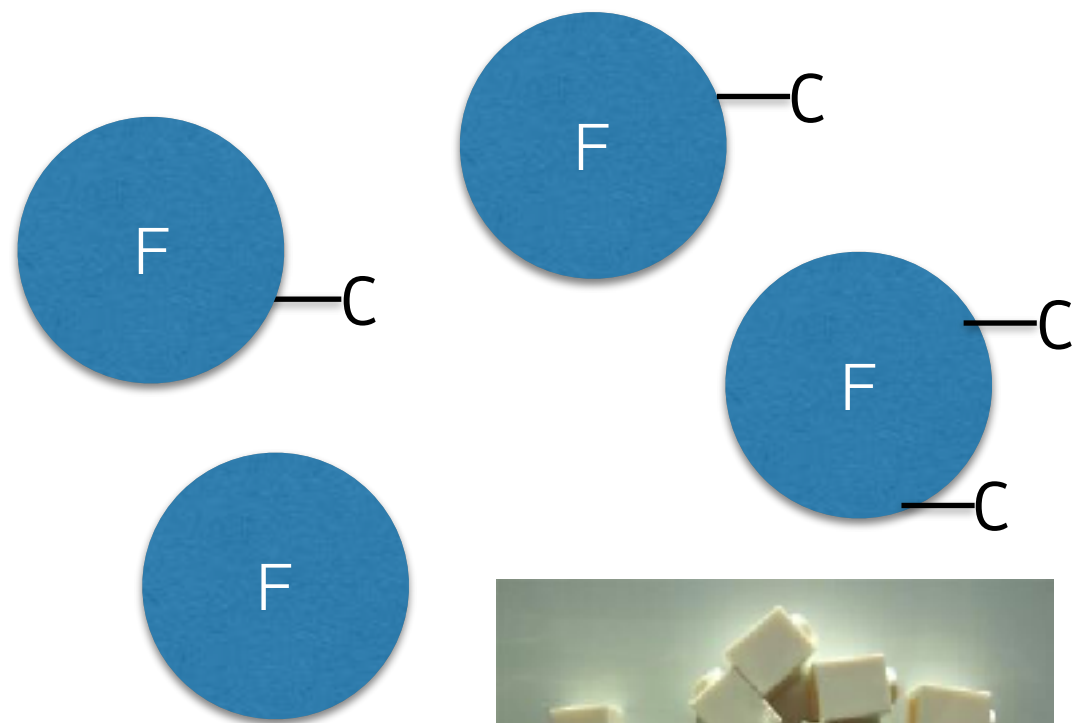
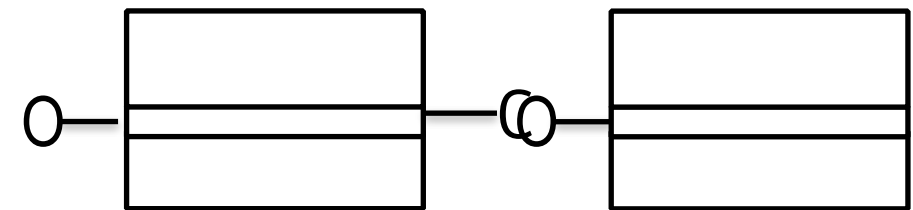


	Proc	OO	Functional
Performance	n/a	n/a	+ +
Security	n/a	n/a	n/a
Learning Curve	++ 	- -	-
Development Effort	++ 	- -	-
Unit test	- -	+ + 	+ + + 
Less Coupling	- -	+ +	+ +
Manage large code	- -	++ 	+ 
Concurrency	- -	- -	+ + 

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();
}
```

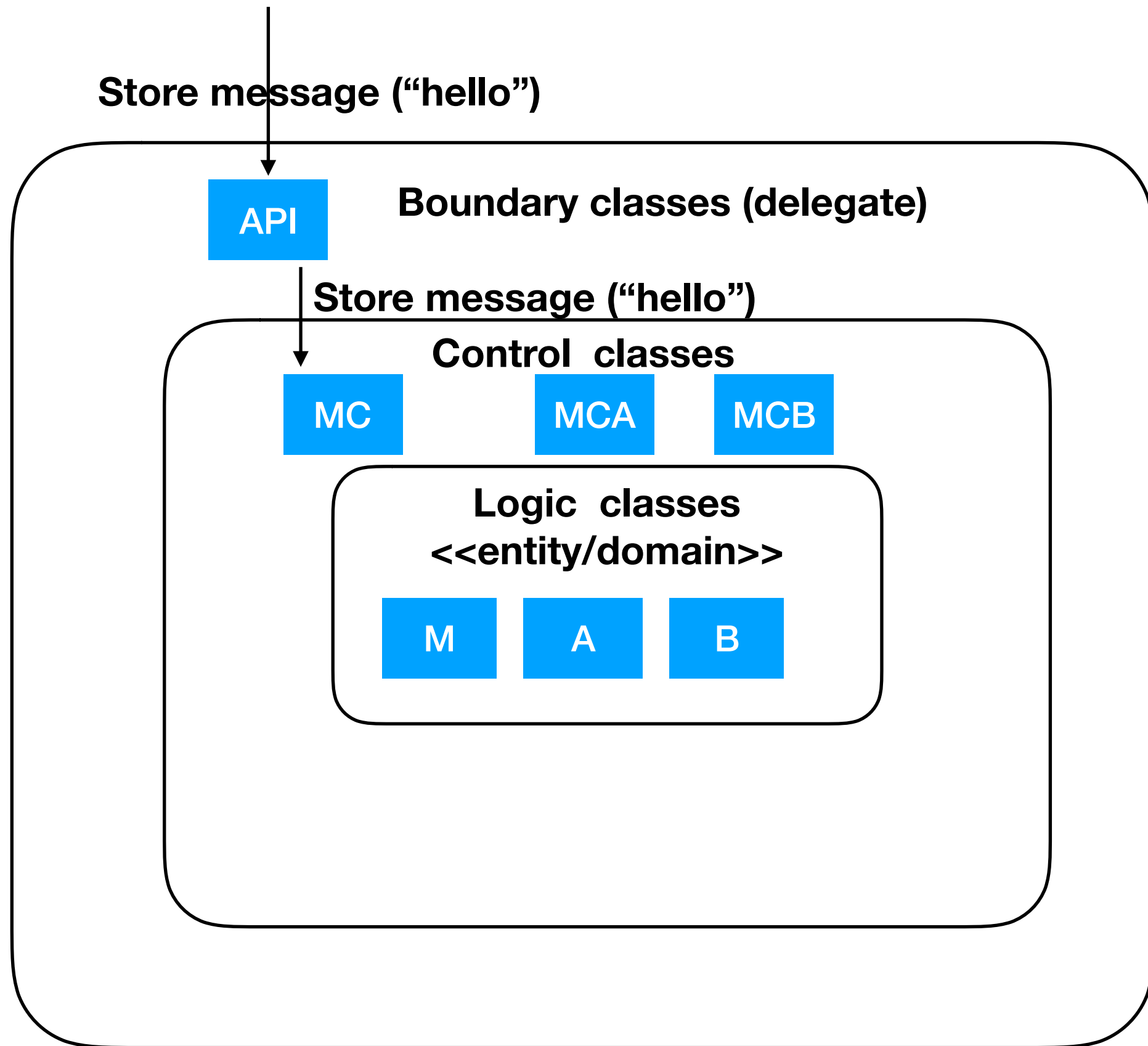
do(new Parrot())

do(new Parrot())

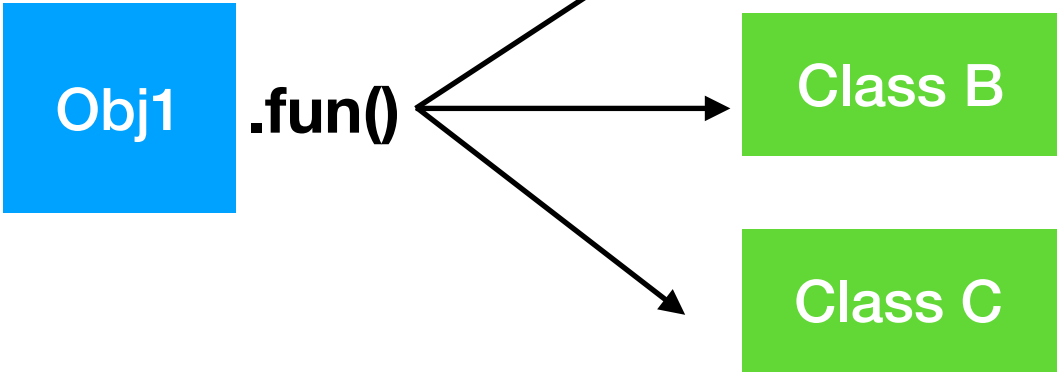
do(new Parrot())

Tight coupling	Interface typing (java, c++)	Duck typing (py, js)	Lamda (py,js, java)
<pre> class Parrot { void fly(){ ... } } </pre>	<pre> interface Bird{ void f1(); } class Parrot implements Bird { void f1(){ ... } } </pre>	<pre> class Parrot{ void f1(){ ... } } </pre>	<pre> class Parrot{ void fly(){ ... } } </pre>
<pre> do(Parrot obj) { obj.fly(); } </pre>	<pre> do(Bird obj) { obj.f1(); } </pre>	<pre> do(obj) { obj.f1(); } </pre>	<pre> do(Lamda f1) { f1(); } </pre>
do(new Parrot())	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
<pre>class Parrot { void fly(){ ... } }</pre>	<pre>interface Bird{ void f1(); } class Parrot implements Bird { void f1(){ ... } }</pre>	<pre>class Parrot{ void f1(){ ... } }</pre>	<pre>class Parrot{ void fly(){ ... } }</pre>	<pre>class CA{ void f1(){ ... } }</pre>
<pre>do(Parrot obj) { obj.fly(); }</pre>	<pre>do(Bird obj) { obj.f1(); }</pre>	<pre>do(obj) { obj.f1(); }</pre>	<pre>do(Lamda f1) { f1(); }</pre>	<pre>do(string cn,string fn){ Class c = class.forName(cn); m = c.getMethod(fn); ... m.invoke(obj,[]); }</pre>
<pre>do(new Parrot())</pre>	<pre>do(new Parrot())</pre>	<pre>do(new Parrot())</pre>	<pre>CA obj = new CA() do()-> obj.fly())</pre>	<pre>do("Parrot","fly")</pre>

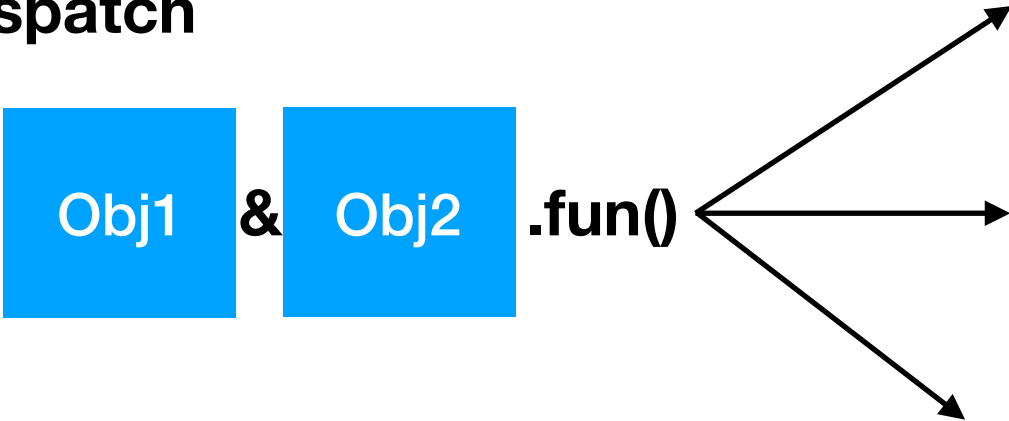


Single dispatch



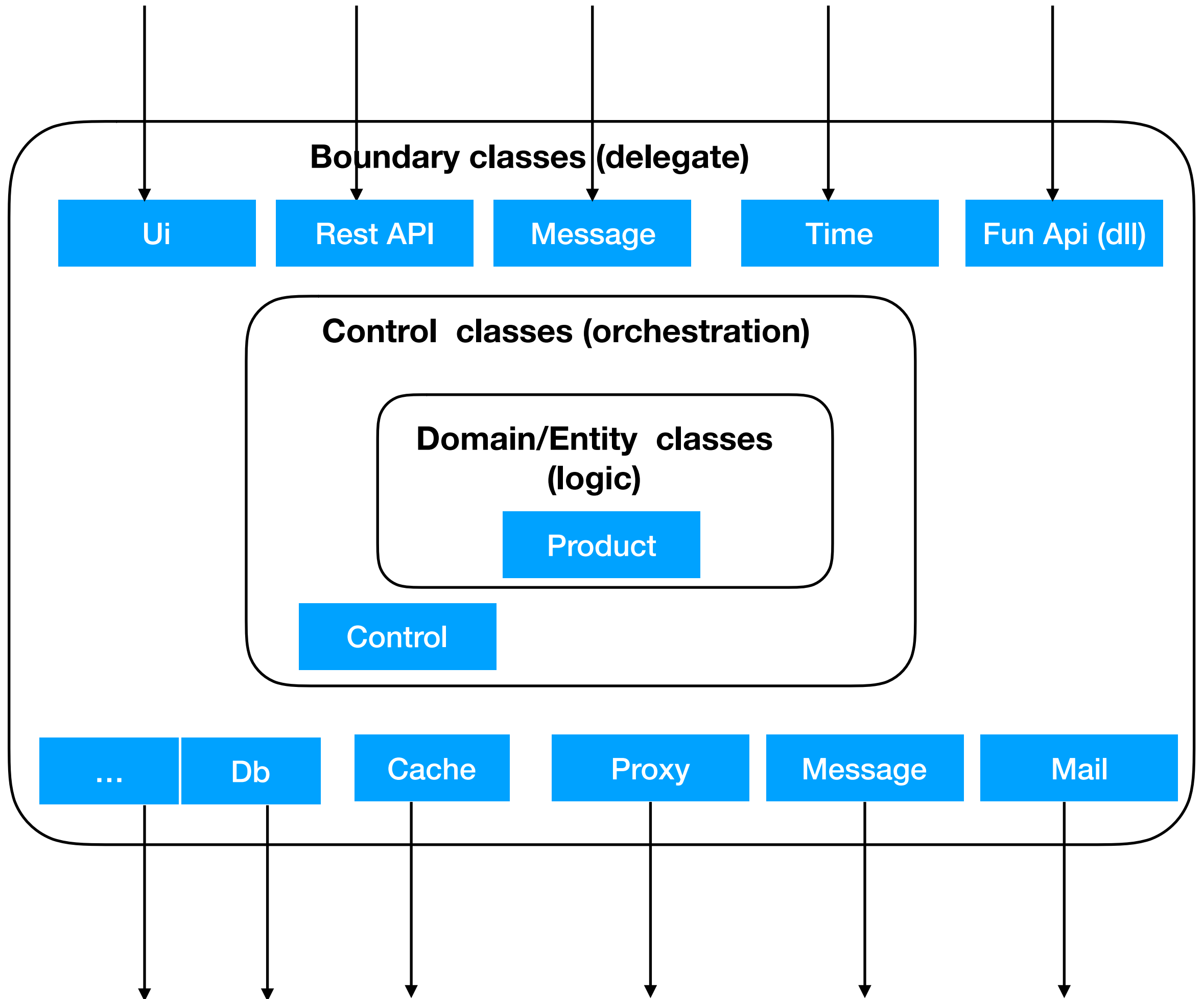
Java, c++, py, ..

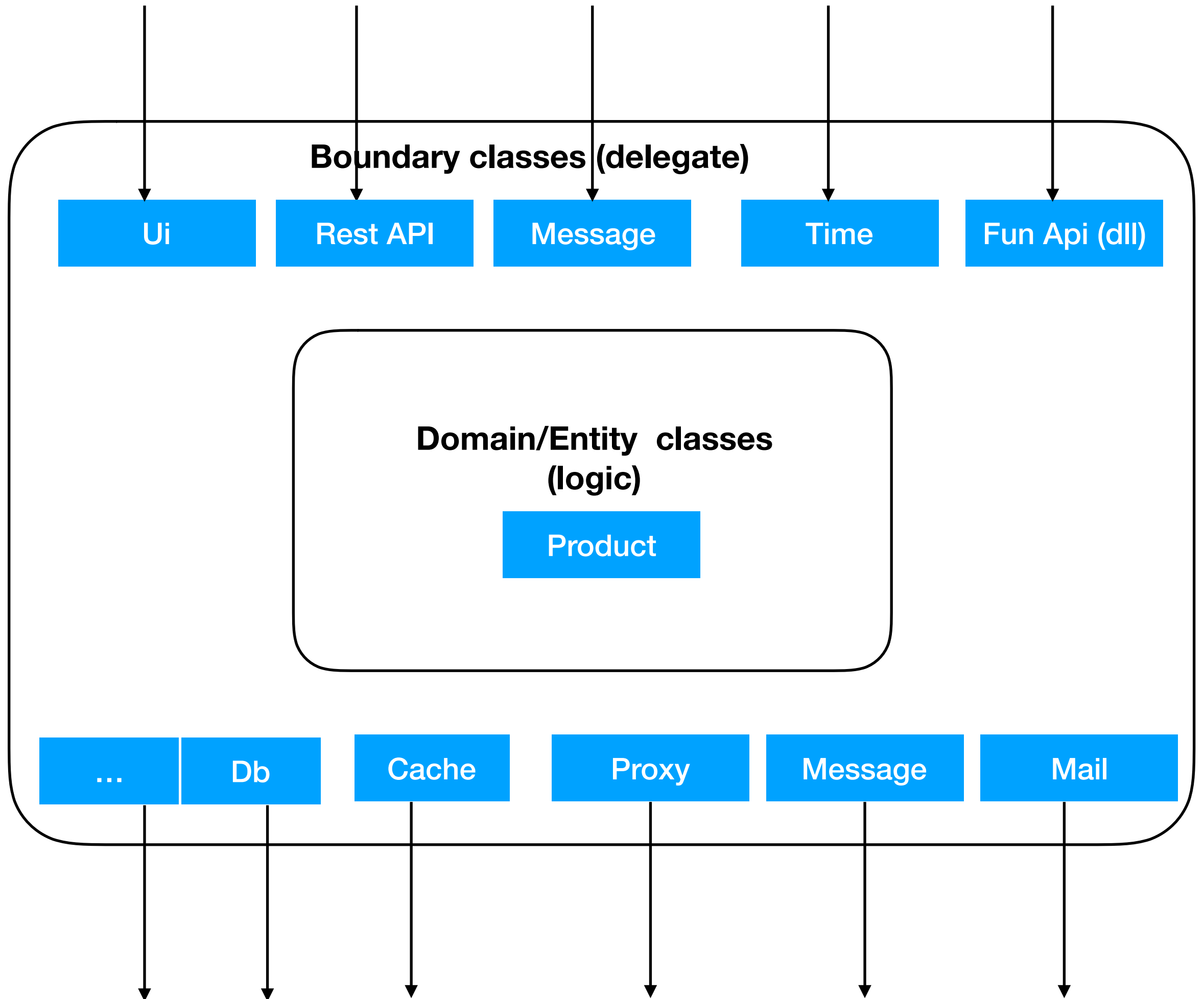
Dual dispatch



Multi dispatch







System

Bounded Context (Inventory)

Boundary classes

Control classes

Workflow classes

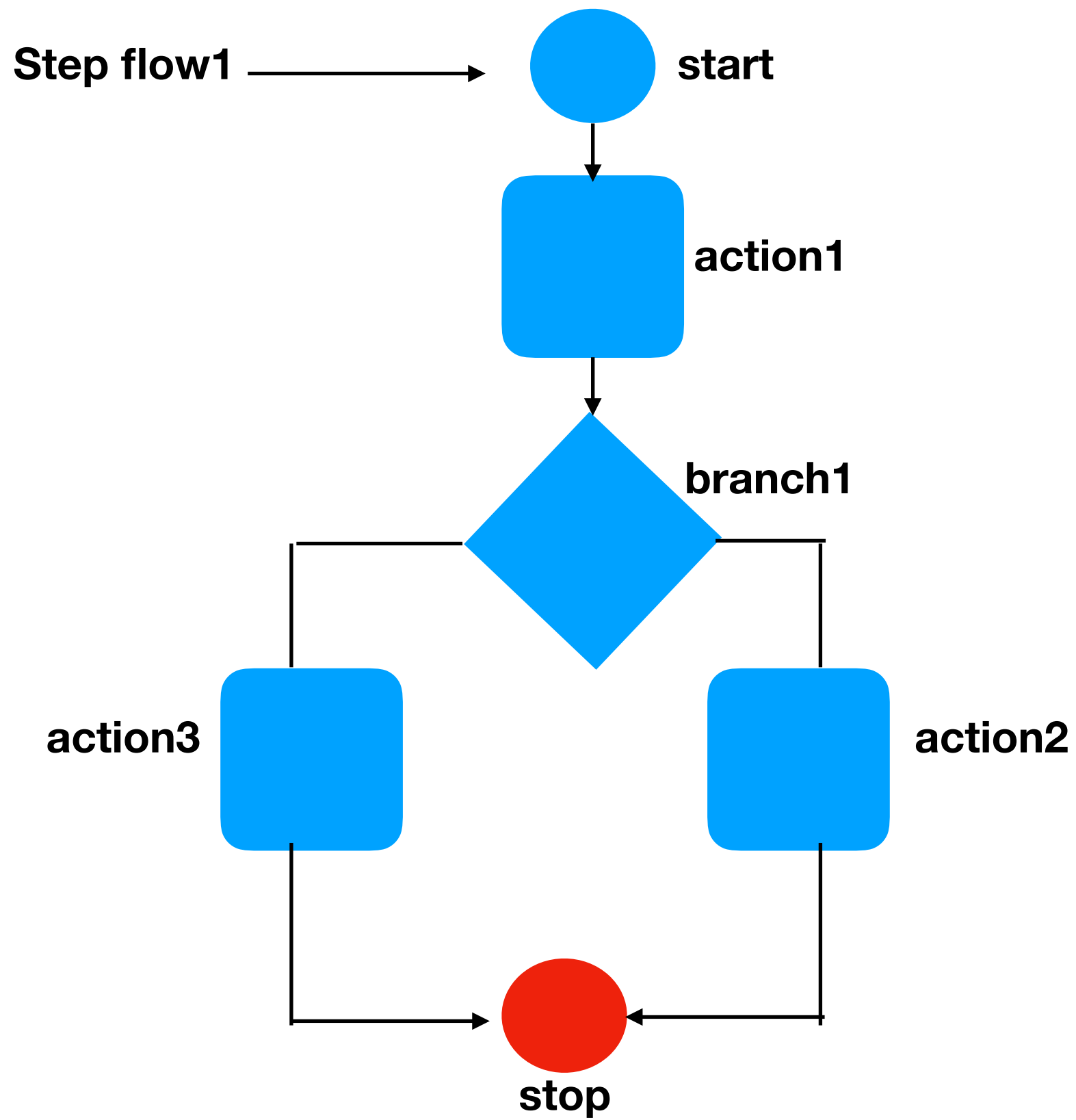
Entity classes

Domain classes

Ag1

Ag2

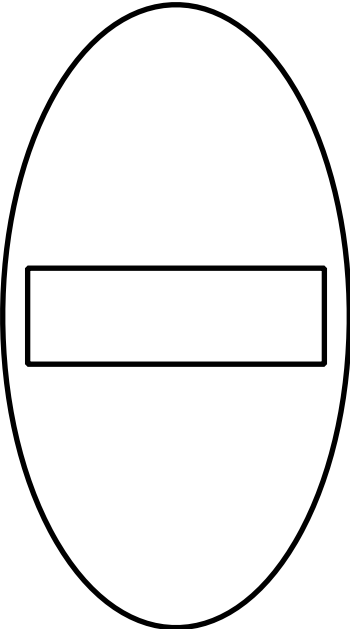
Bounded Context (Accounting)



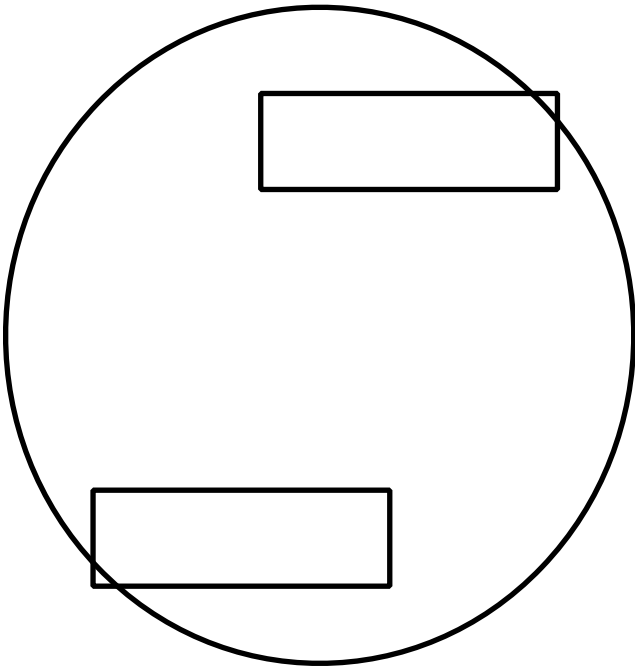
DDD



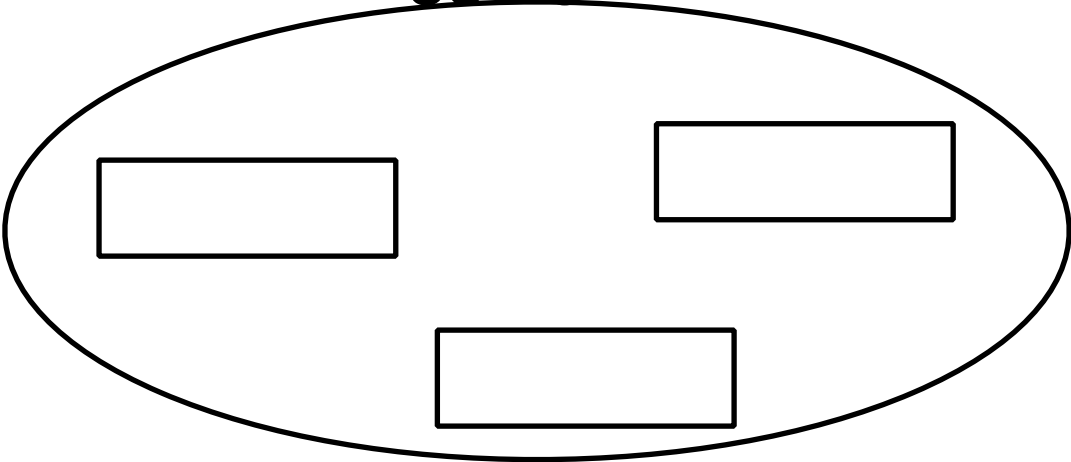
Aggregate



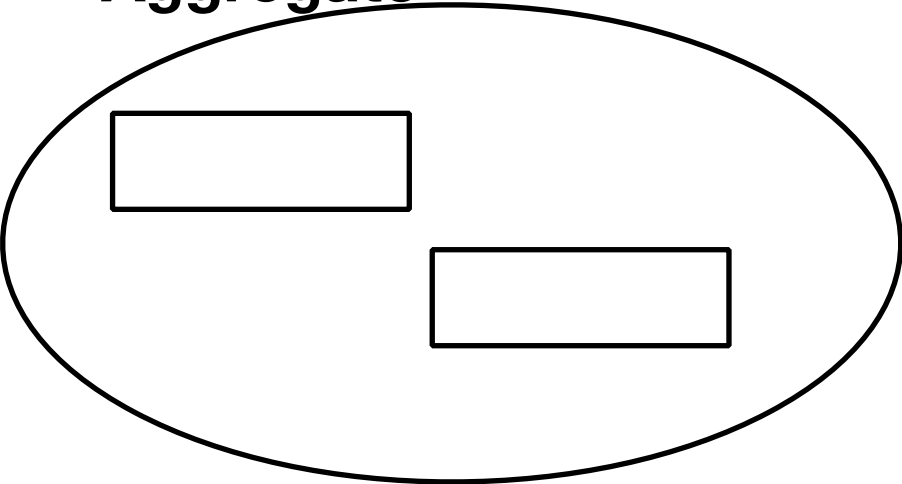
Aggregate



Aggregate

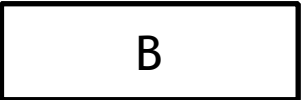


Aggregate



Module

Aggregate



Root



Issue Aggregate

Issue (aggregate root)

Guid	Id

string	Text
bool	IsClosed
Enum	CloseReason

Guid	RepositoryId
Guid	AssignedUserId

ICollection<Comment>	
ICollection<IssueLabel>	

Comment (entity)

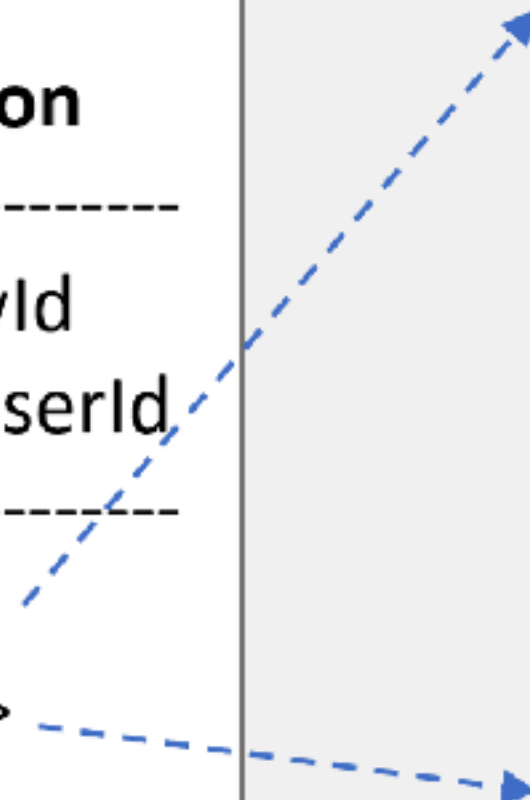
Guid	Id

string	Text
DateTime	CreationTime

Guid	IssueId
Guid	UserId

IssueLabel (value obj)

Guid	IssueId
Guid	LabelId



DDD

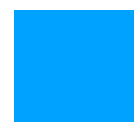


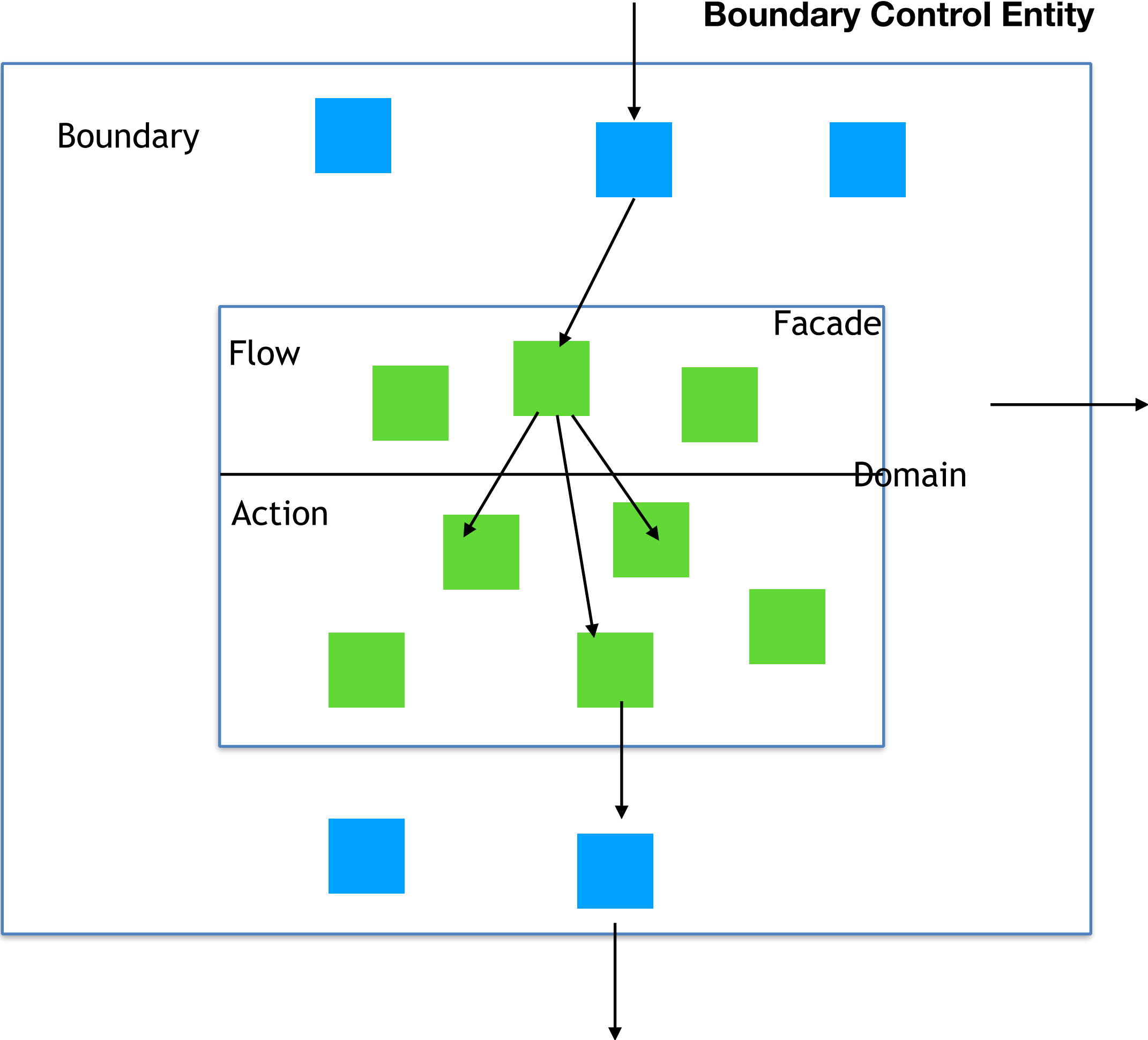
**Boundary class
(flexible)**



**Api
Ui
Message
Timer**

**Domain class
(temple)**



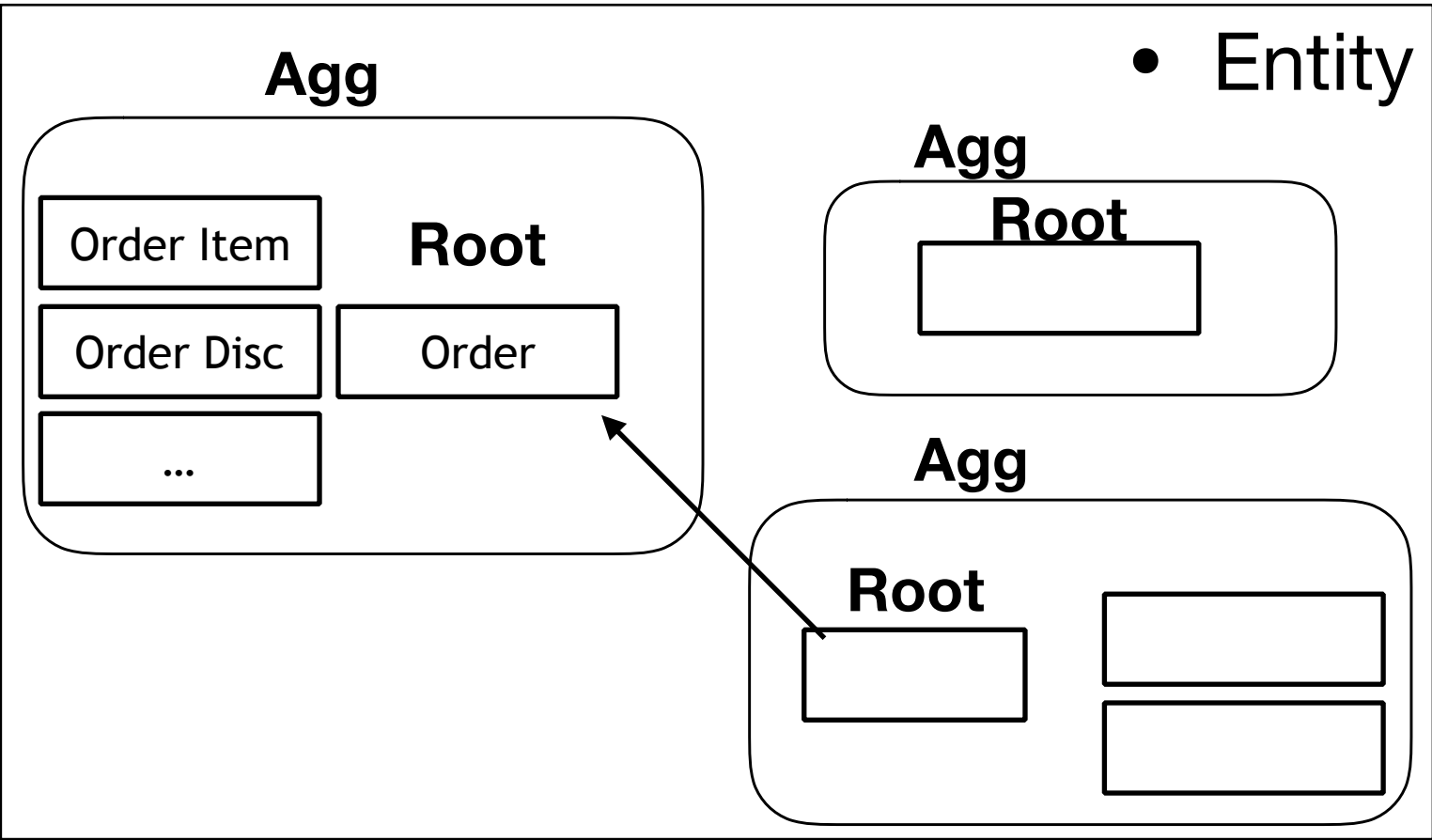




• Boundary



• Control



• Entity

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

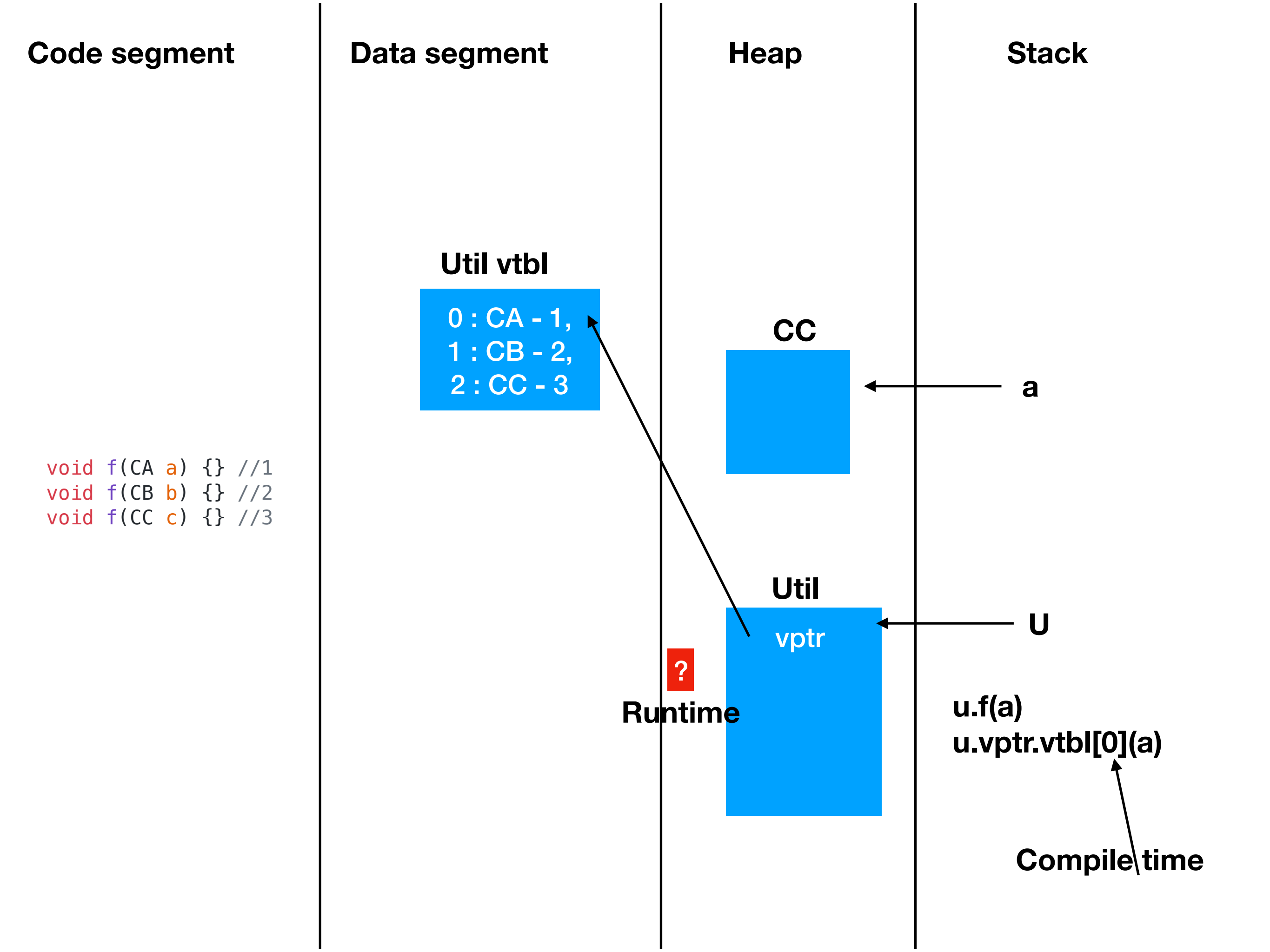
u

?

Runtime

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

Compile time



Code segment

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

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

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

Data segment

CX vtbl[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

Heap

CC

CZ

vptr

Stack

a

x

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

Compile time

Code segment

Data segment

Heap

Stack

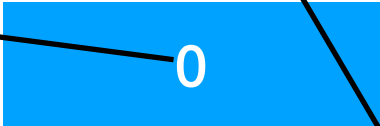
SA::Create()



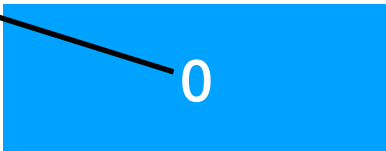
CA::Create()



SA vtbl[1]



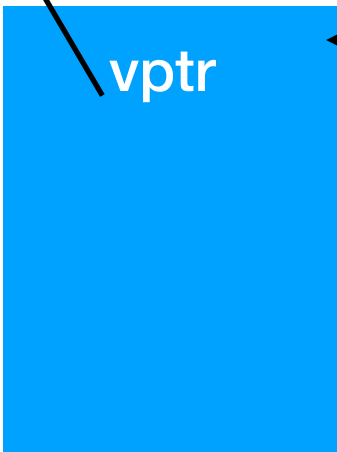
CA vtbl[1]



?

Runtime

CA



vptr

account

account.Create()

account.vptr.vtbl[0]

Compile time

