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
- Expectations
- Years of Exp
- Technology stack

https://forms.gle/nKn4mG1bVVX1U16x8

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

- LSP
- ISP
- Upcasting/abstraction
-

Bad

- Type check
- Flag check
- dont use overloading
  on Family of types
- Downcasting
-

Good

- SRP (***) - { soc, size }
- Low coupling (***)
- LSP
- Upcast / abstraction /interface
- ISP
- Boundary Control Entity (*)
- DRY
- Unit testable
- DDD
- Aggregates => unit
- Concurrency friendly
- Low cyclomatic complexity < 10
- KISS
- Prefer composition over inheritance
- YAGNI

Bad

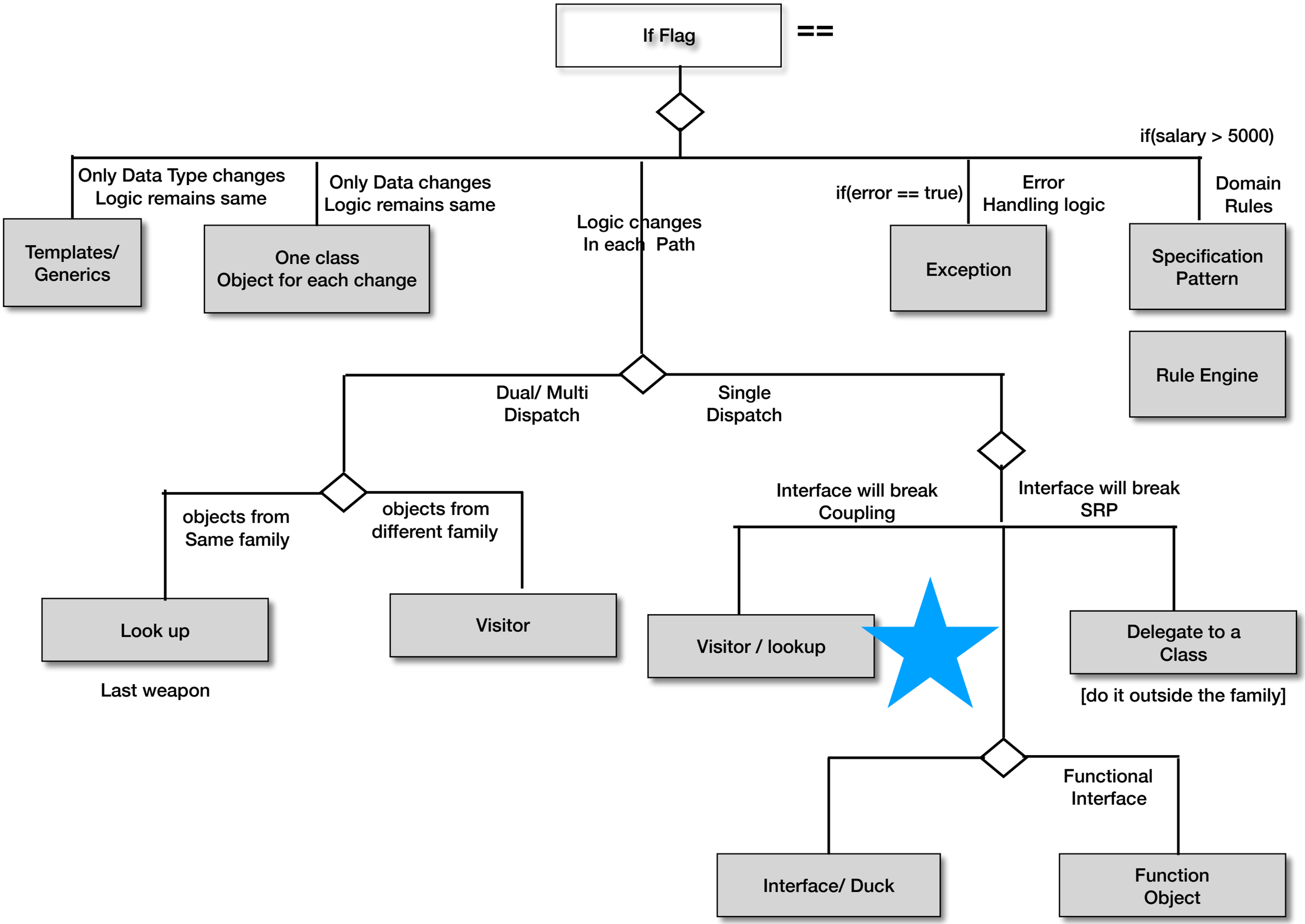
- Overloading family of types
- Type check
- Down casting
- Flag
- Magic strings/numbers
- Arrow code
- bool/int /null for error handling
- Inheritance
- Duplicate code (*)
- Dead Code(*)
- Commented Code (*)
- Static methods
- Functional interface ~
- God class (*)
- Cyclic coupling (*)
- * to * coupling (**)

SOC

- Things which do not change together should not be kept together
- Domain logic and domain rules
- Boundary logic and domain logic
- Domain rules 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
Templates/
Generics
Only Data changes
Logic remains same
One class
Object for each change
Logic changes
In each Path
if(error == true)
Error
Handling logic
Exception
if(salary > 5000)
Domain
Rules
Specification
Pattern
Rule Engine
Dual/ Multi
Dispatch
Single
Dispatch
objects from
Same family
objects from
different family
Look up
Visitor
Last weapon
Interface will break
Coupling
Interface will break
SRP
Visitor / lookup
Delegate to a
Class
[do it outside the family]
Functional
Interface
Interface/ Duck
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 # 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

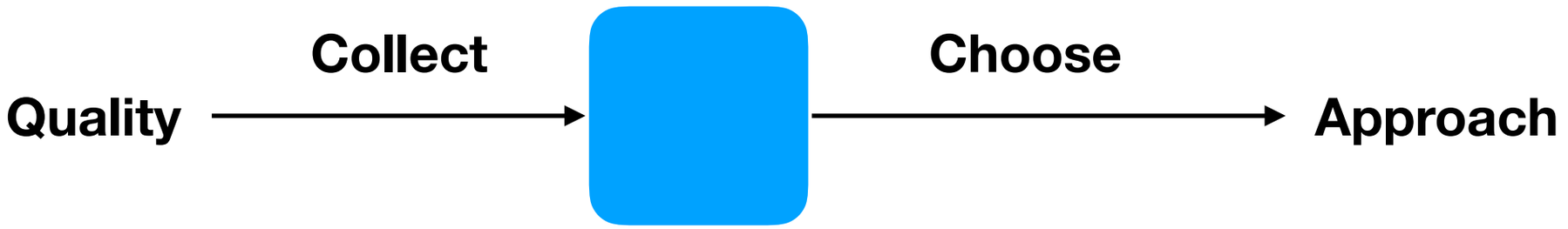
 # low cyclomatic complexity
 # readablity
 # unit test
 # OCP

1

Flag => Polymorphism/ Abstraction/ Interface

Coupling => Polymorphism/ Abstraction/ Interface

Down casting => Polymorphism/ Abstraction/ Interface


Collect
Quality
Choose
Approach

“system quality”

Domain

Understands

Which : Qualities

How much: Measure

Collect

Choose

Approach

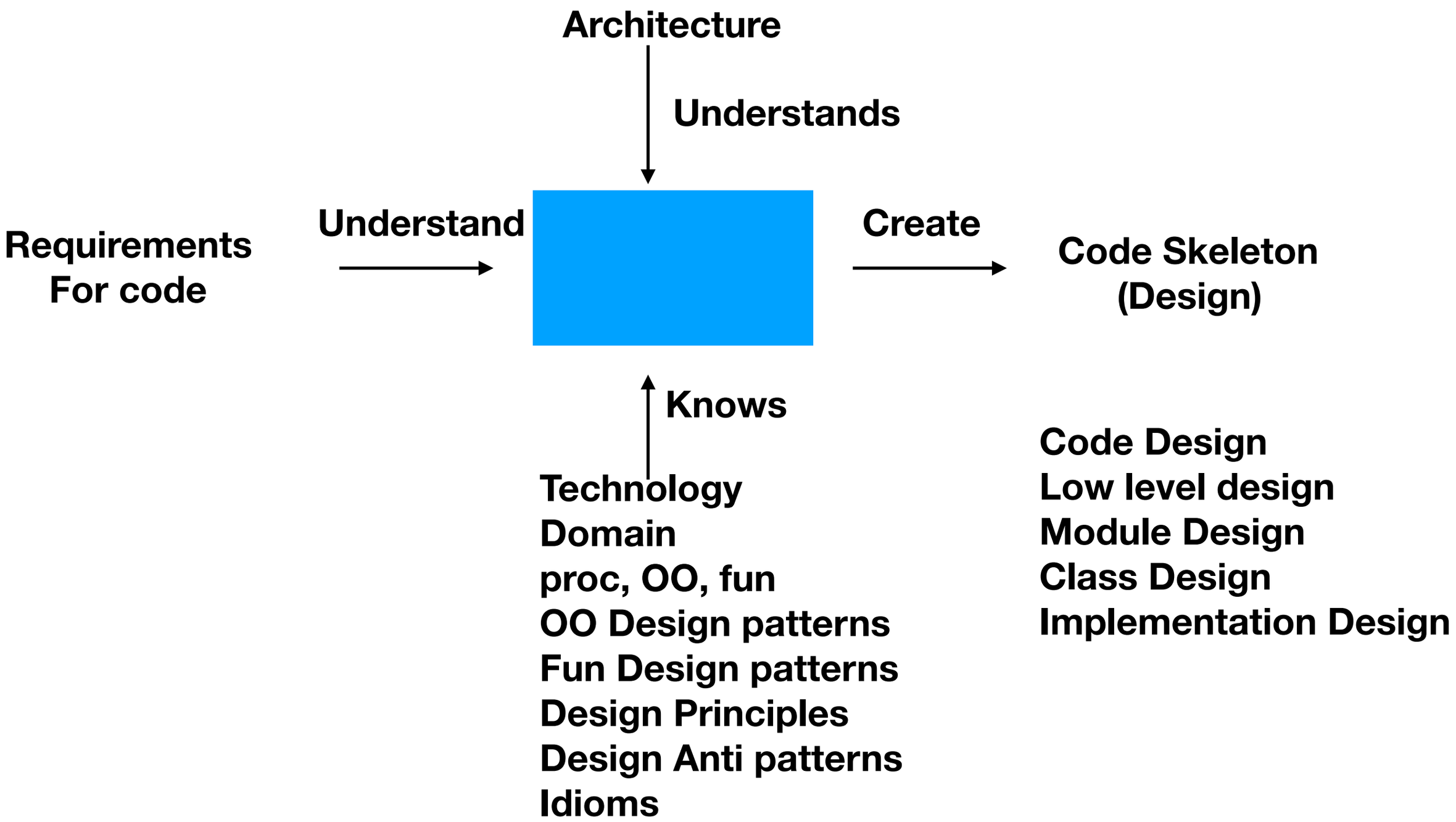
<< Architecture >>

Architecture design
Blue print
HLD
System Design
…

Knows

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{
   ?
}
```

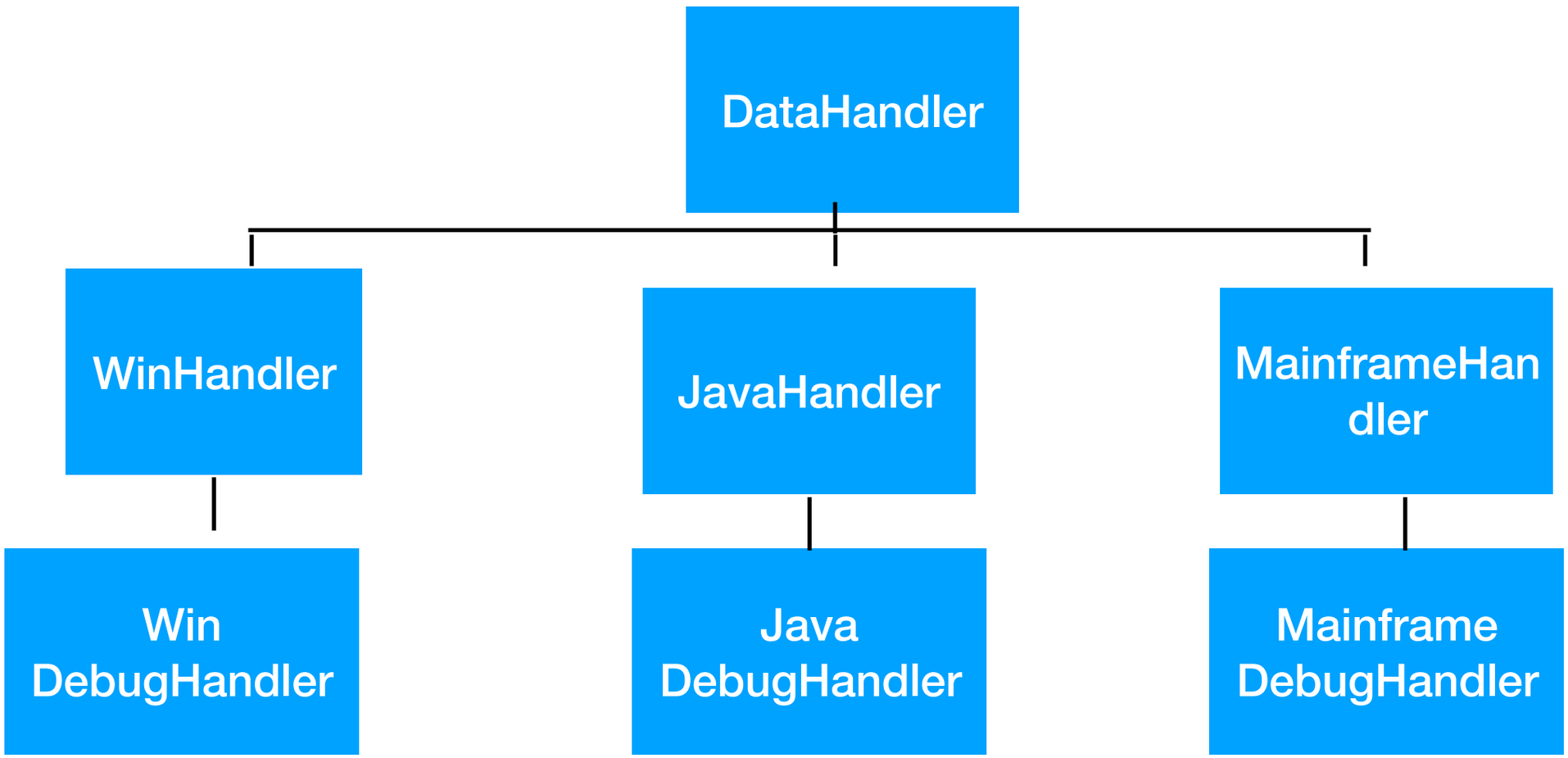
```
fun(Bird bird){

    if(....)
        bird.fly();
    ....

}
```

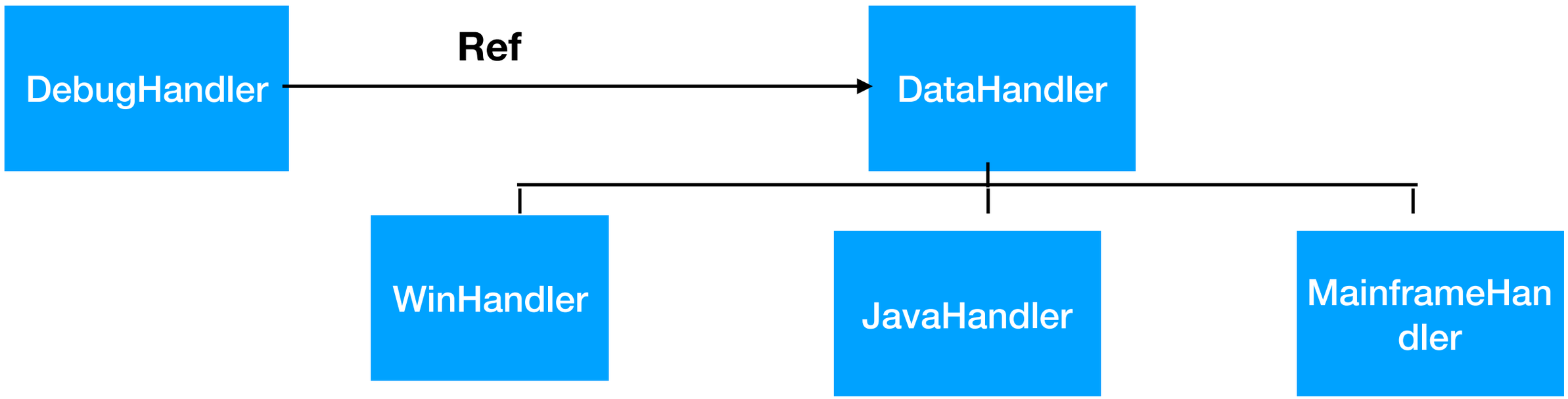
```
Interface Bird extends LivingThing{

}
```

```
Class Parrot{
}
```

```
Interface Bird{
}
```

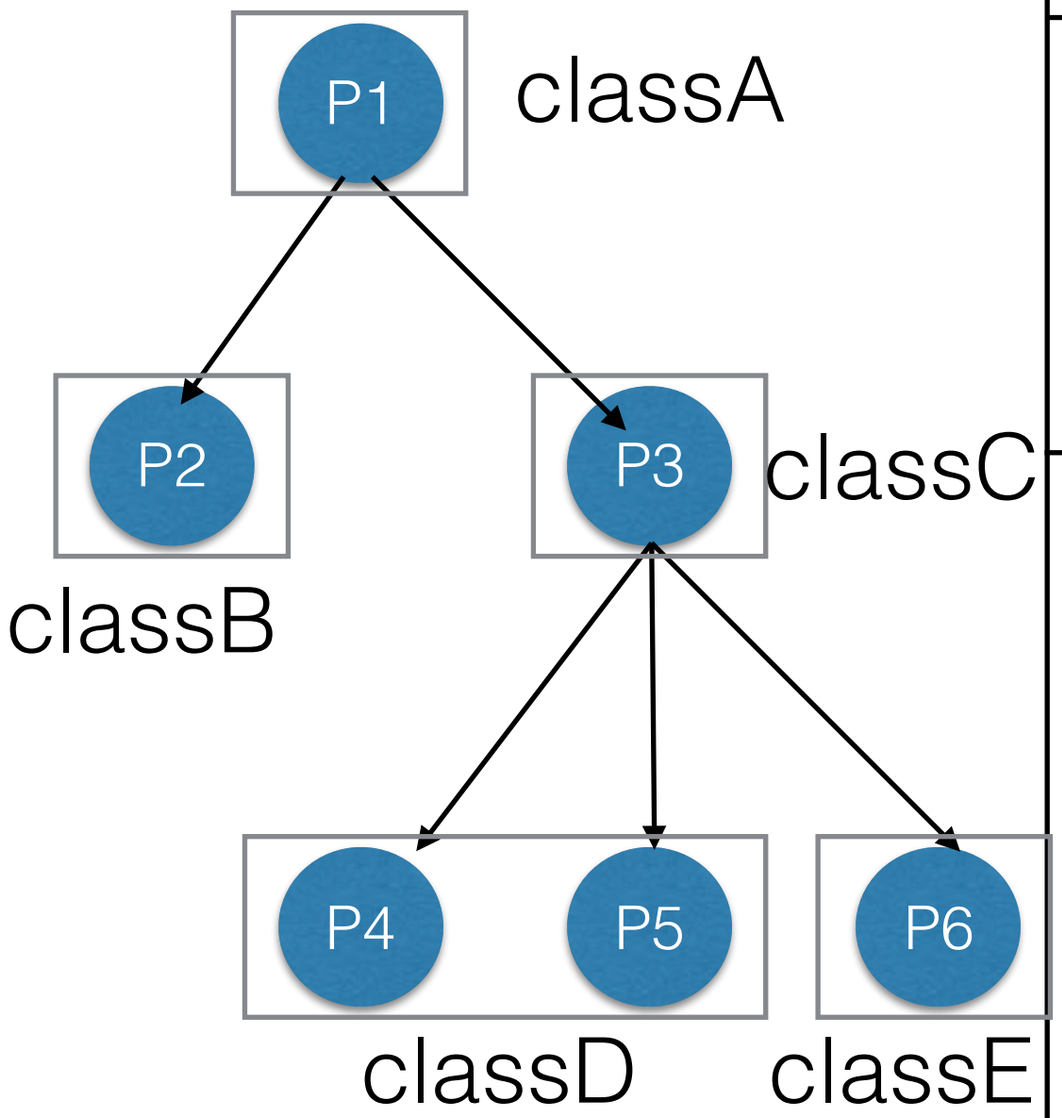
```
interface LivingThing{
    ...
}

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

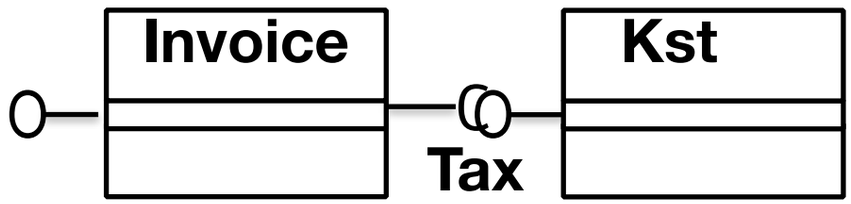

DebugHandler
Ref
DataHandler
WinHandler
JavaHandler
MainframeHandler

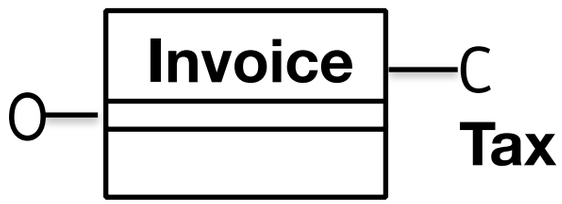
a+b	3 cpu cycles
Fun call	10 cpu cycles
Exception handling	1000 cpu cycles
Create thread	200,000 cpu cycles
Write to file	10,00,000 cpu cycles
Db call	40,00,000 cpu cycles

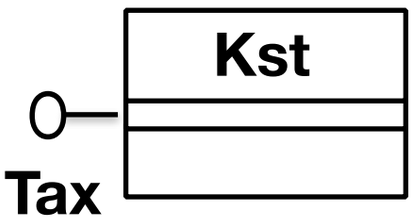
Procedural Prog (tree)

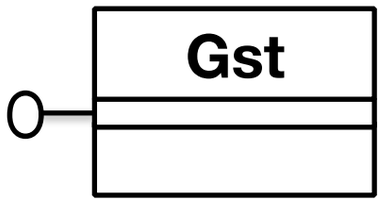


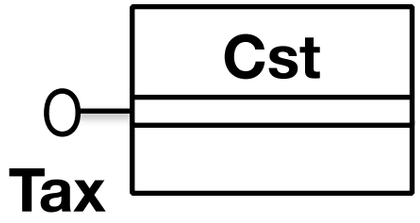
OO Prog (Lego)





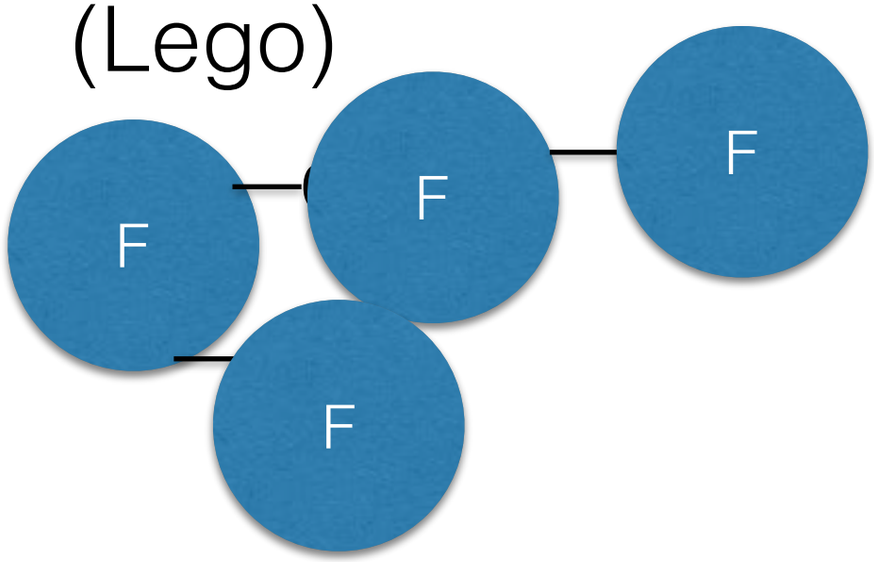


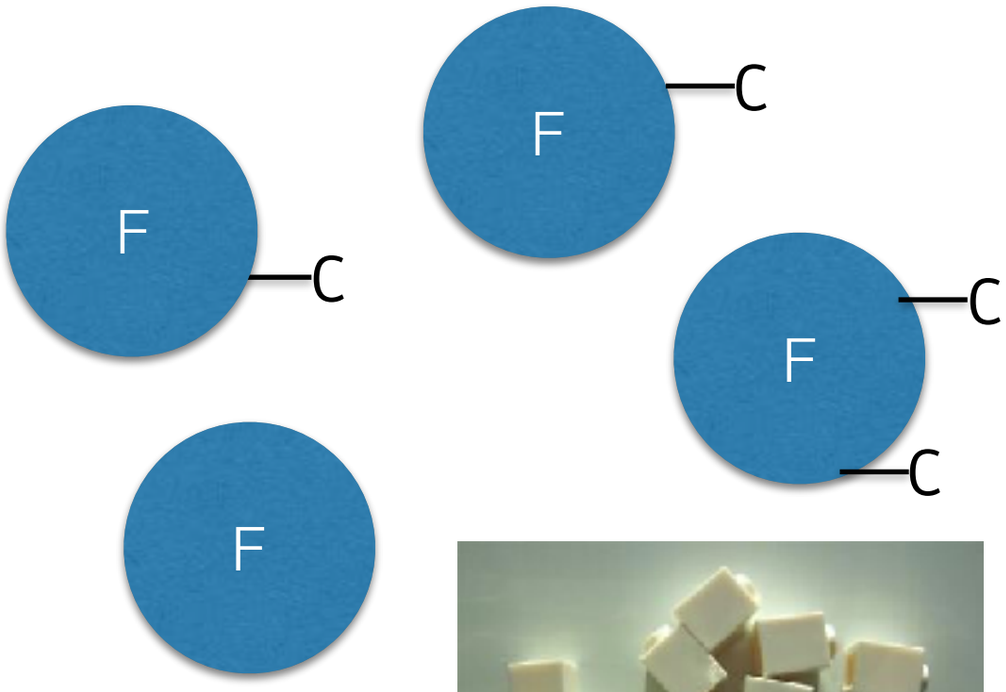




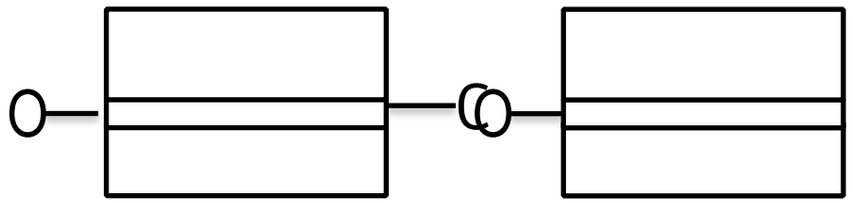
	Proc	**OO**	**Functional**
Performance	n/a	n/a	+ +
Security	n/a	n/a	n/a
Learning Curve	++	- -	-
Development Effort	++	- -	-
Unit test	- -	+ +	+ + +
Less Coupling	- -	+ +	+ +
Manage large code	- -	++	+
Concurrency	- -	- -	+ +

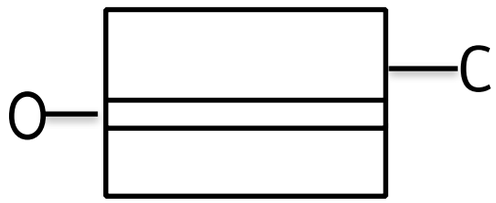
Functional Prog
(Lego)

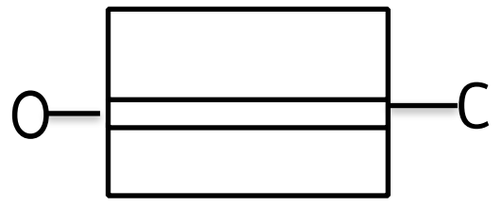


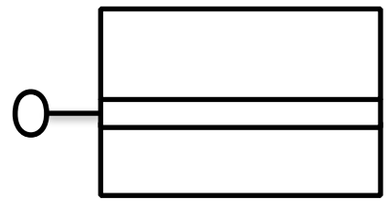


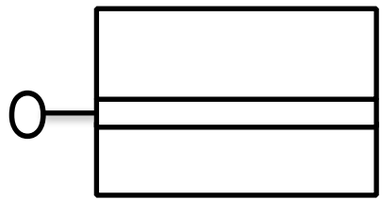
OO Prog
(Lego)











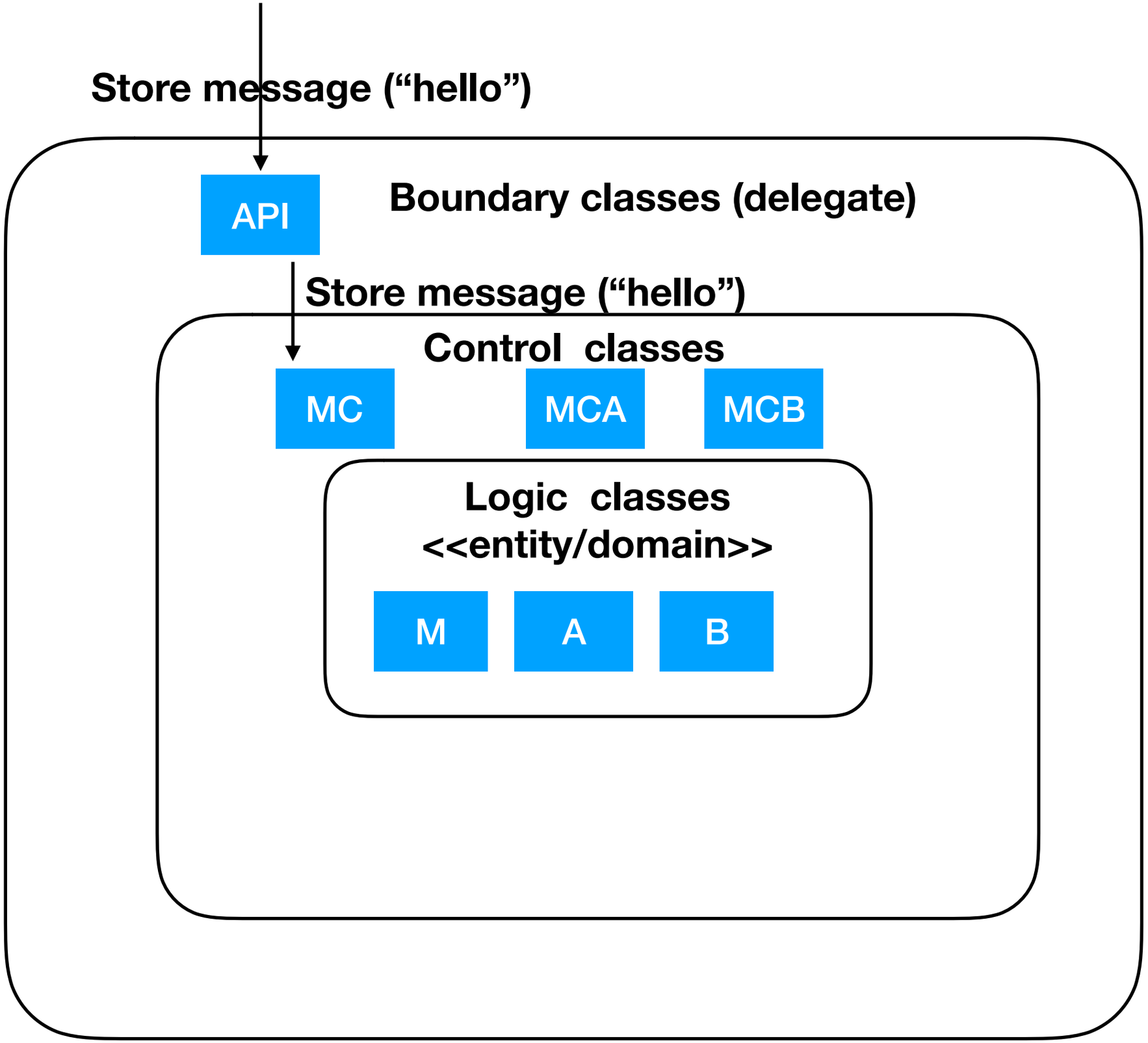
Tight coupling	Interface typing (java, c++, C#) Compiled Languages	Duck typing (py, js) Dynamic Languages

class Parrot
{
  void fly(){
  …
  }
}
 	
do(Parrot obj)
{
  obj.fly();
}
 	
do(new Parrot( ))
	interface Bird{
	  void fly();
	}
	
	class Parrot implements Bird
	{
	  void fly(){
	  …
	  }
	}
	
	do(Bird obj)
	{
	  obj.fly();
	}
	
	do(new Parrot( ))
		class Parrot{
		  void fly(){
		  …
		  }
		}
		
		do(obj)
		{
		  obj.fly();
		}
		
		do(new Parrot( ))

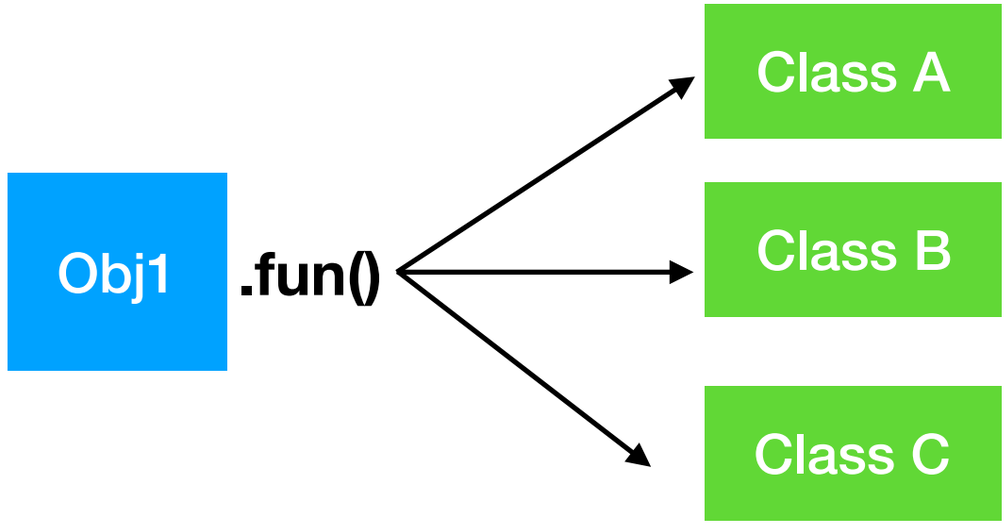
Tight coupling	Interface typing (java, c++)	Duck typing (py, js)	Lamda (py,js, java)	Reflection

class Parrot
{
  void fly(){
  …
  }
}
 	
do(Parrot obj)
{
  obj.fly();
}
	interface Bird{
	  void f1();
	}
	
	class Parrot
	implements Bird
	{
	  void f1(){
	  …
	  }
	}
	
	do(Bird obj)
	{
	  obj.f1();
	}
		class Parrot{
		  void f1(){
		  …
		  }
		}
		
		do(obj)
		{
		  obj.f1();
		}
			class Parrot{
			  void fly(){
			  …
			  }
			}
			
			do(Lamda f1)
			{
			  f1();
			}
				class CA{
				  void 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”)

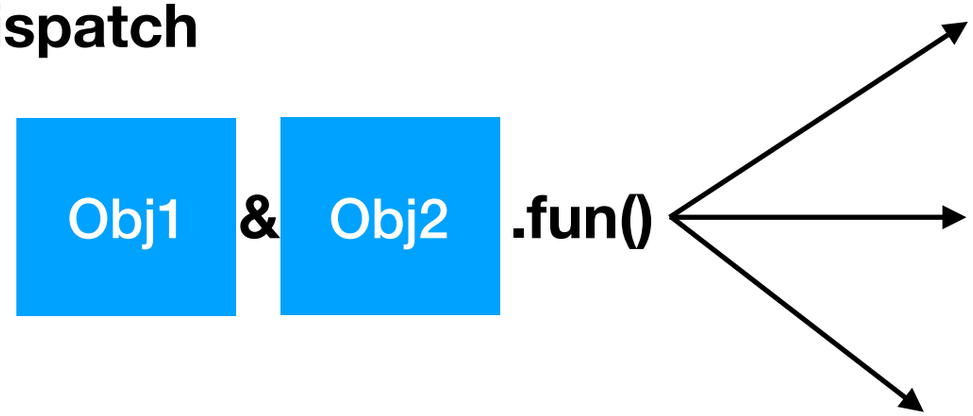

Store message (“hello”)
API
Boundary classes (delegate)
Store message (“hello”)
Control classes
MC
MCA
MCB
Logic classes
<<entity/domain>>
M
A
B

Single dispatch

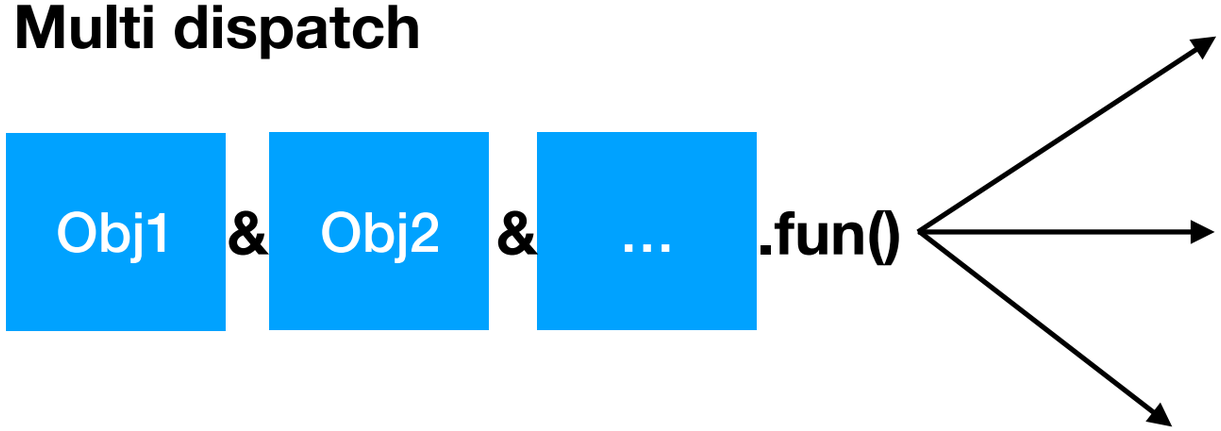


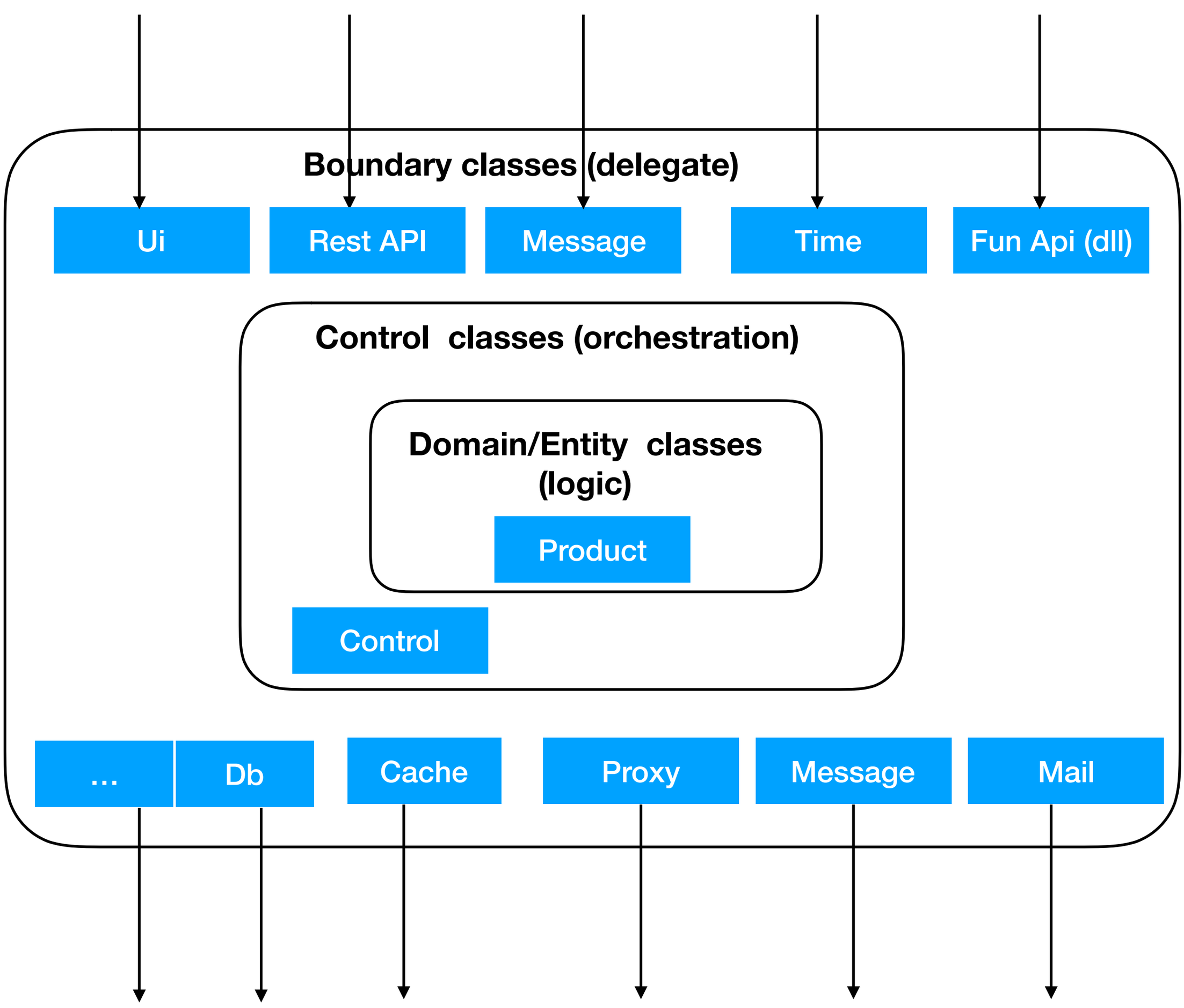
Java, c++, py, ..

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

System

Bounded Context (Inventory)

Boundary classes

Control classes

Workflow classes

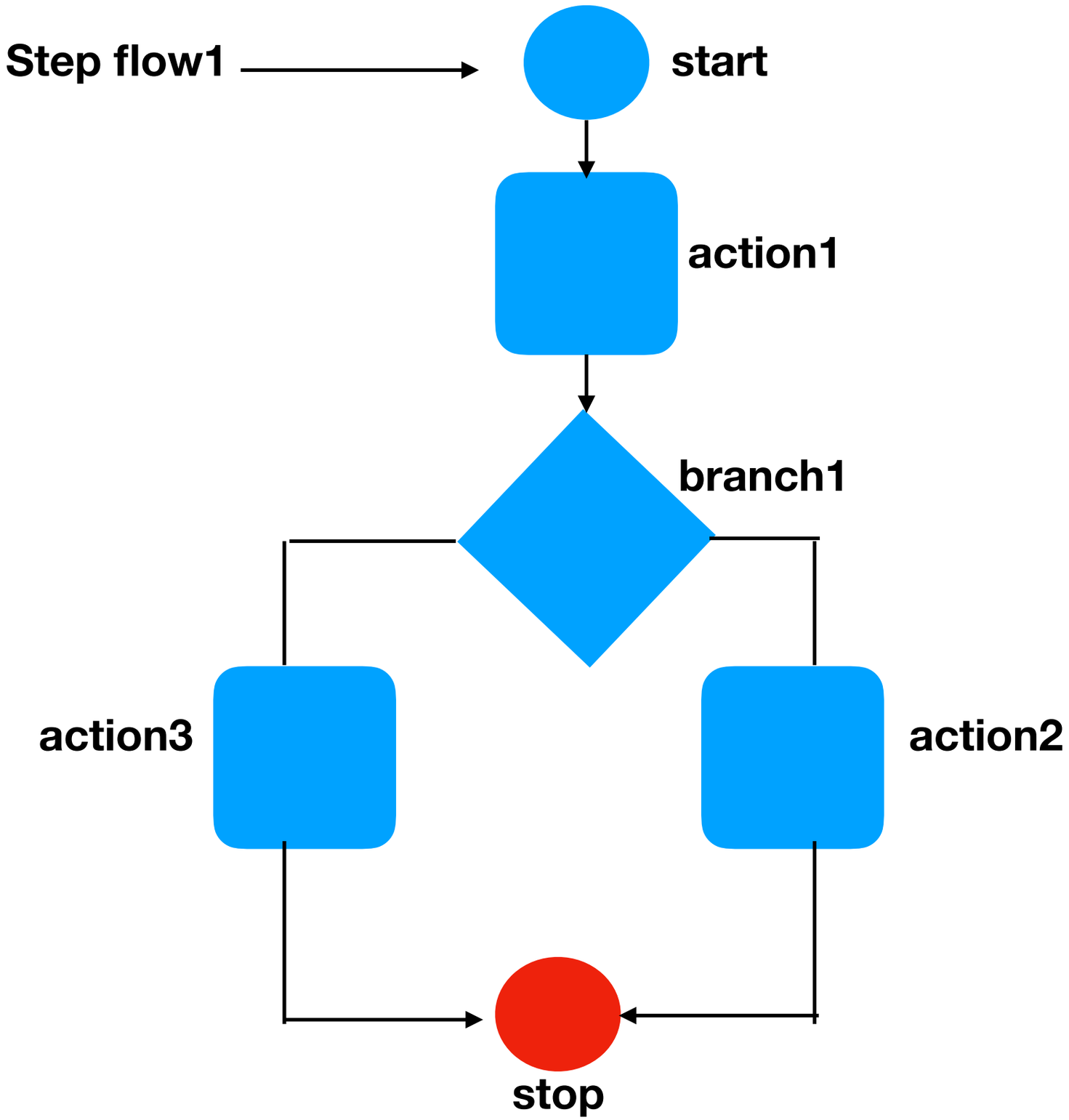
Entity classes

Domain classes

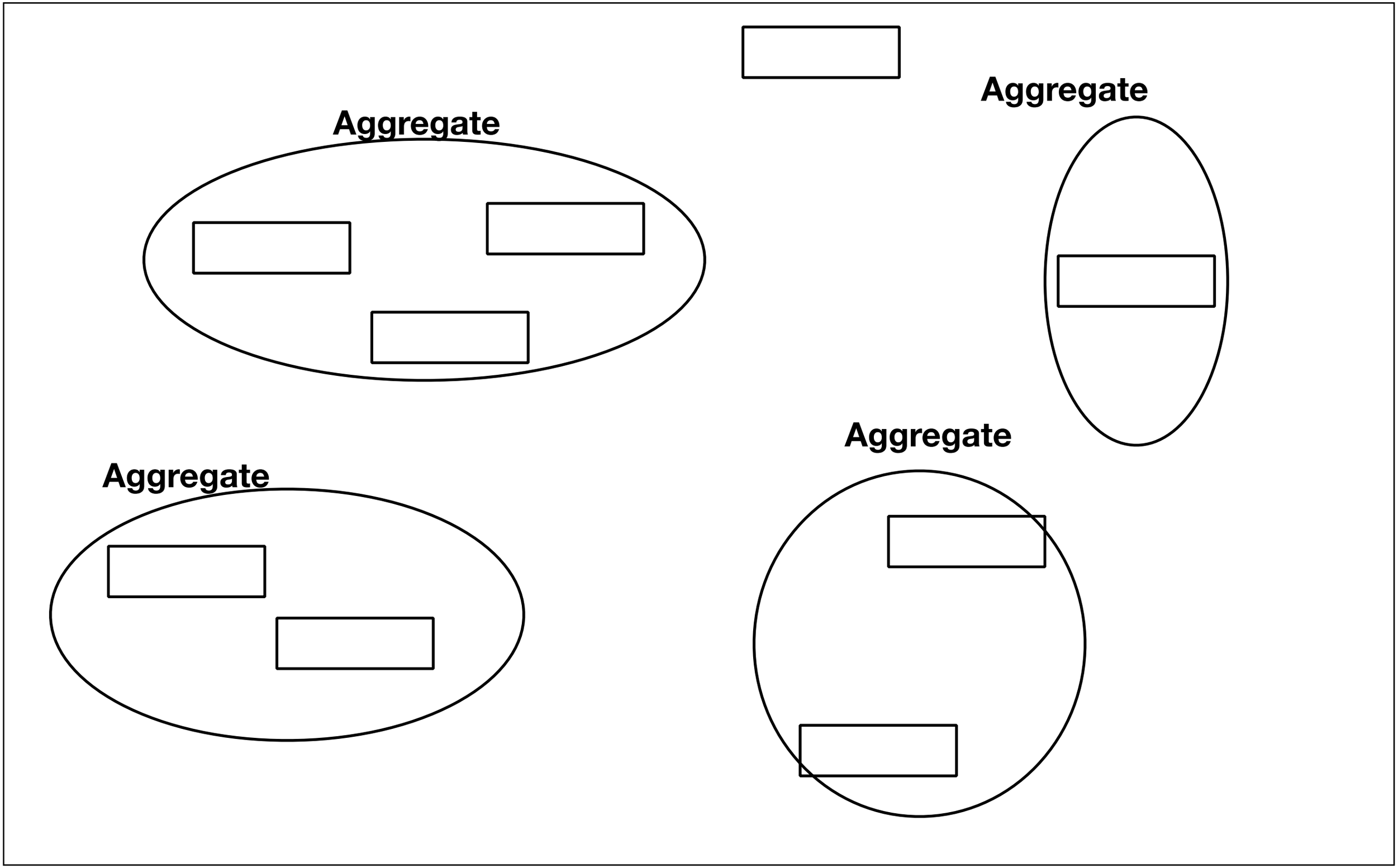
Ag1

Ag2

Bounded Context (Accounting)


Step flow1
start
action1
branch1
action3
action2
stop

DDD



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)

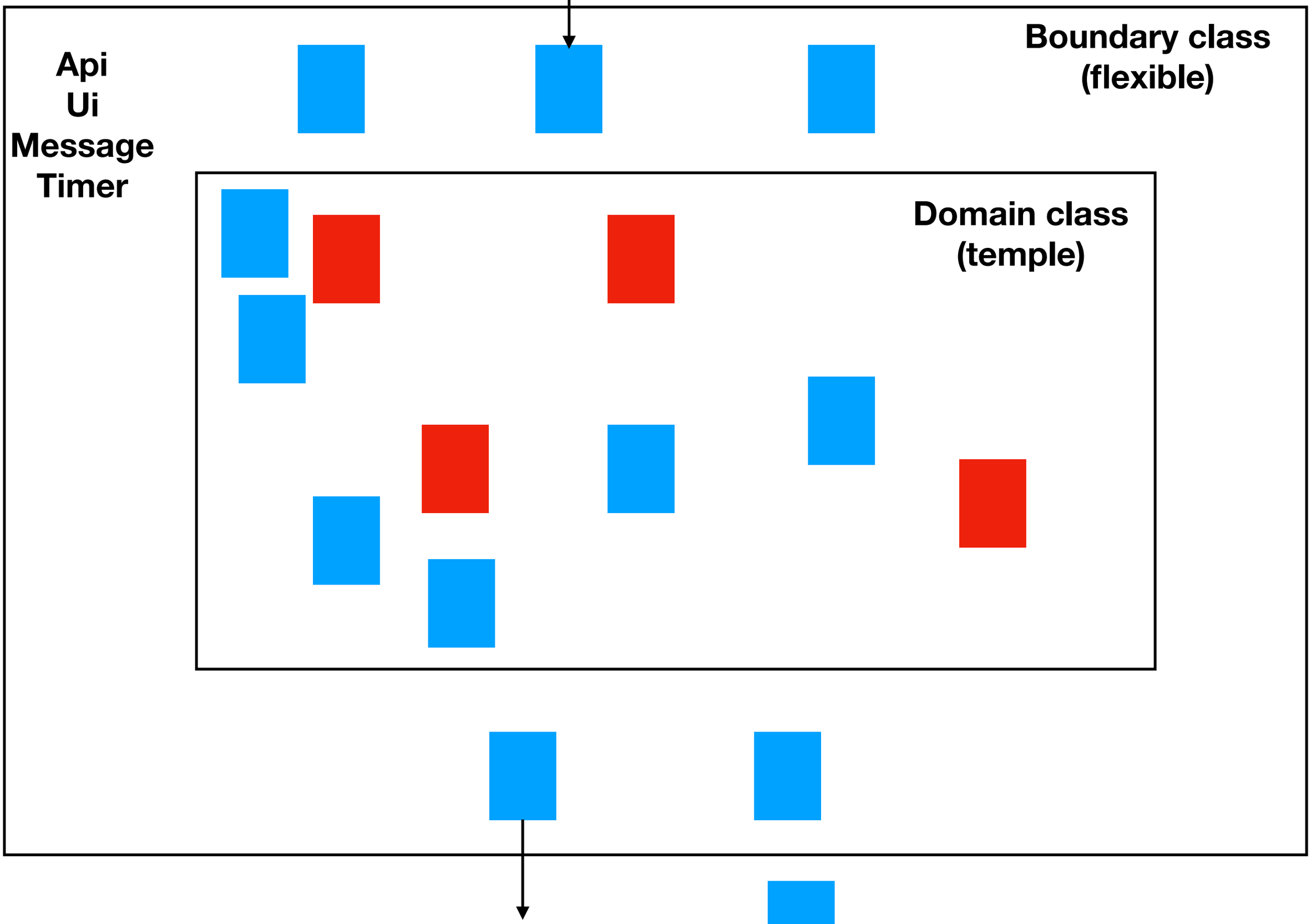
Guid	**IssueId**
Guid	**LabelId**

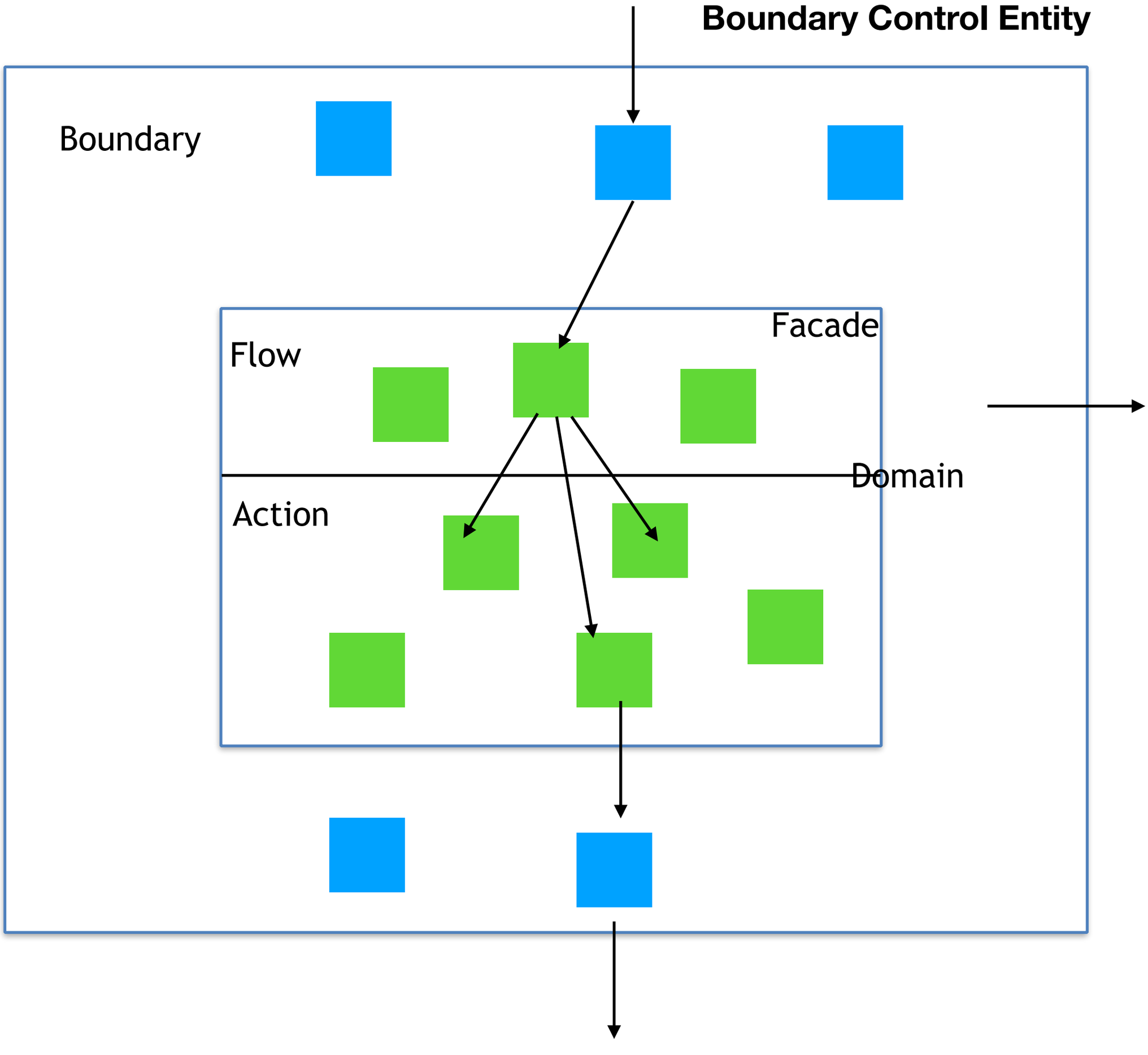
DDD

Boundary class
(flexible)

Api
Ui
Message
Timer

Domain class
(temple)




Boundary Control Entity
Boundary
Facade
Flow
Domain
Action

- Boundary
- Control
- Entity

Agg

Order Item

Order Disc

...

Root

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

```
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]

0

CA vtbl[1]

0

?

Runtime

CA

vptr

account

account.Create()

account.vptr.vtbl[0]

Compile time