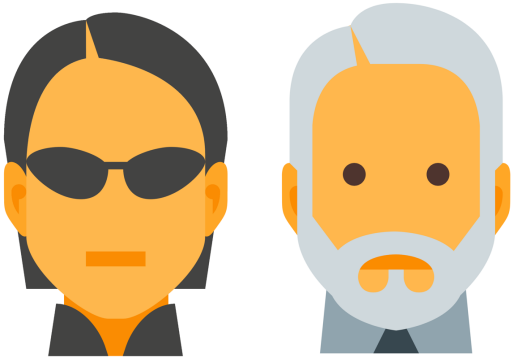
155  PATTERNS

Wilmer Krisp

278 SOFTWARE ARCHITECTURE DESIGN MODELS

COMPLETE CATALOG OF ALL CLASSICAL PATTERNS IN THE ARCHIMATE LANGUAGE

01

Design Patterns

Elements of Reusable Object-Oriented Software

GANG OF FOUR

02

Enterprise patterns

Catalog of Patterns of Enterprise Application Architecture

MARTIN FOWLER

03

Analysis Patterns

Reusable Object Models

MARTIN FOWLER

04

Domain Driven Design

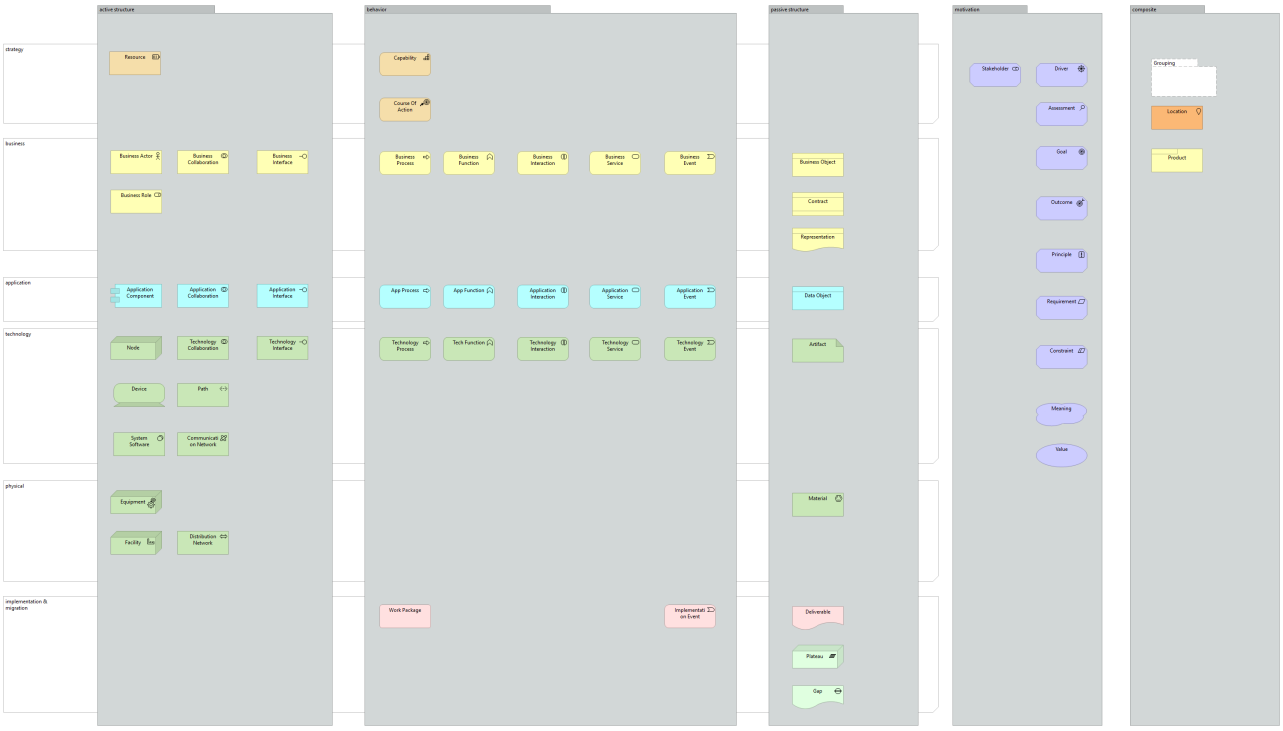
Tackling Complexity in the Heart of Software.

ERIC EVANS

USED NOTATION

ARCHIMATE METAMODEL

The Open Group



DESIGN PATTERNS 11
CREATIONAL PATTERNS 12
ABSTRACT FACTORY 13
BUILDER 16
FACTORY METHOD 18
PROTOTYPE 20
SINGLETON 22
STRUCTURAL PATTERNS 24
ADAPTER OF CLASS 25
ADAPTER OF OBJECT 27
BRIDGE 30
COMPOSITE 32
DECORATOR 34
FACADE 36
FLYWEIGHT 39
FLYWEIGHT + COMPOSITE 41
PROXY 42
BEHAVIORAL PATTERNS 44
CHAIN OF RESPONSIBILITY 45
COMMAND 47
INTERPRETER 49
ITERATOR 51
MEDIATOR 53
MEMENTO 55
OBSERVER 57
STATE 59
STRATEGY 61
TEMPLATE METHOD 63
VISITOR 66
ENTERPRISE PATTERNS 68
BUSINESS LOGIC 69
DOMAIN MODEL 70
SERVICE LAYER 71
TRANSACTION SCRIPT 72
TABLE MODULE 73
DATA SOURCES 74
ACTIVE RECORD 75
DATA MAPPER 76
ROW DATA GATEWAY 77
TABLE DATA GATEWAY 78
MODELING BEHAVIOR 79

IDENTITY MAP 80
LAZY LOAD 81
UNIT OF WORK. 82
MODELING STRUCTURE HIERARCHY 83
CLASS TABLE INHERITANCE 84
CONCRETE TABLE INHERITANCE 85
INHERITANCE MAPPERS 86
SINGLE TABLE INHERITANCE 87
MODELING STRUCTURE RELATIONS 88
ASSOCIATION TABLE MAPPING 89
DEPENDENT MAPPING 90
EMBEDDED VALUE 91
FOREIGN KEY MAPPING 92
IDENTITY FIELD 93
SERIALIZED LOB 94
METADATA 95
METADATA MAPPING 96
QUERY OBJECT 97
REPOSITORY 98
WEB REPRESENTATION CONTROLLER 99
MODEL VIEW CONTROLLER 100
APPLICATION CONTROLLER 101
FRONT CONTROLLER 102
PAGE CONTROLLER 103
WEB REPRESENTATION VIEW 104
TEMPLATE VIEW 105
TRANSFORM VIEW 106
TWO STEP VIEW 107
DISTRIBUTED PROCESSING 108
DATA TRANSFER OBJECT 109
REMOTE FAÇADE 110
PARALLEL PROCESSING 111
COARSE-GRAINED LOCK 112
IMPLICIT LOCK 113
OPTIMISTIC OFFLINE LOCK 114
PESSIMISTIC OFFLINE LOCK 115
SESSION STATE 116
CLIENT SESSION STATE 117
DATABASE SESSION STATE 118
SERVER SESSION STATE 119
COMMON PATTERNS 120
GATEWAY 121
LAYER SUPERTYPE 122

MAPPER 123
MONEY 124
PLUGIN 125
RECORD SET 126
REGISTRY 127
SEPARATED INTERFACE 128
SERVICE STUB 129
SPECIAL CASE 130
VALUE OBJECT 131
ANALYSIS PATTERNS 132
ACCOUNTABILITY 134
PARTY 135
ACCOUNTABILITY 136
ORGANIZATION HIERARCHIES 138
ORGANIZATION STRUCTURE 139
ACCOUNTABILITY KNOWLEDGE LEVEL 140
PARTY TYPE GENERALIZATIONS 141
HIERARCHIC ACCOUNTABILITY 142
OPERATING SCOPES 143
POST 144
OBSERVATIONS AND MEASUREMENTS 145
QUANTITY 146
CONVERSION RATIO 147
OBSERVATIONS AND MEASUREMENTS 148
COMPOUND UNITS 149
MEASUREMENT 150
OBSERVATION 151
SUBTYPING OBSERVATION CONCEPTS 152
PROTOCOL 153
DUAL TIME RECORD 154
REJECTED OBSERVATION 155
ACTIVE OBSERVATION, HYPOTHESIS, AND PROJECTION 156
ASSOCIATED OBSERVATION 157
PROCESS OF OBSERVATION 158
OBSERVATIONS FOR CORPORATE FINANCE 159
ENTERPRISE SEGMENT 160
MEASUREMENT PROTOCOL 161
RANGE 162
OBSERVATIONS FOR CORPORATE FINANCE 163
PHENOMENON WITH RANGE 166
REFERRING TO OBJECTS 167
NAME 168
IDENTIFICATION SCHEME 169

OBJECT MERGE 170
OBJECT EQUIVALENCE 171
REFERRING TO OBJECTS 172
INVENTORY AND ACCOUNTING 173
ACCOUNT 174
TRANSACTIONS 175
SUMMARY ACCOUNT 176
MEMO ACCOUNT 177
POSTING RULES 178
INVENTORY AND ACCOUNTING 179
INDIVIDUAL INSTANCE METHOD 180
POSTING RULE EXECUTION 181
POSTING RULES FOR MANY ACCOUNTS 182
CHOOSING ENTRIES 183
ACCOUNTING PRACTICE 184
SOURCES OF AN ENTRY 185
BALANCE SHEET AND INCOME STATEMENT 186
CORRESPONDING ACCOUNT 187
SPECIALIZED ACCOUNT MODEL (BILLING EXAMPLE) 188
SPECIALIZED ACCOUNT MODEL (INVENTORY EXAMPLE) 189
BOOKING ENTRIES TO MULTIPLE ACCOUNTS 190
PLANNING 191
PROPOSED AND IMPLEMENTED ACTION 192
COMPLETED AND ABANDONED ACTIONS 193
SUSPENSION 194
PLAN 195
PROTOCOL 196
RESOURCE ALLOCATION 197
PLANNING 198
PLANNING (NO OUTCOME) 199
OUTCOME AND START FUNCTIONS 200
TRADING 201
CONTRACT 202
PORTFOLIO 203
QUOTE 204
SCENARIO 205
TRADING 206
DERIVATIVE CONTRACTS 207
FORWARD CONTRACTS 208
OPTIONS 209
PRODUCT 210
SUBTYPE STATE MACHINES 211
PARALLEL APPLICATION AND DOMAIN HIERARCHIES 212

DERIVATIVE CONTRACTS 213
TRADING PACKAGES 214
MULTIPLE ACCESS LEVELS TO A PACKAGE 215
MUTUAL VISIBILITY 216
TRADING PACKAGES 217
LAYERED ARCHITECTURE FOR INFORMATION SYSTEMS 218
TWO-TIER ARCHITECTURE 219
THREE-TIER ARCHITECTURE 220
PRESENTATION AND APPLICATION LOGIC 221
DATABASE INTERACTION 222
TYPE MODEL DESIGN 223
IMPLEMENTING ASSOCIATIONS 224
IMPLEMENTING GENERALIZATION 225
OBJECT CREATION 226
OBJECT DESTRUCTION 227
ENTRY POINT. 228
IMPLEMENTING CONSTRAINTS 229
DOMAIN DRIVEN DESIGN 230
MODEL AND STRUCTURAL ELEMENTS 231
MODEL-DRIVEN DESIGN 232
LAYERED ARCHITECTURE (ASYMMETRIC) 235
HEXAGONAL ARCHITECTURE (SYMMETRIC) 237
238
COMPOSITE UI 241
ENTITIES 242
VALUE-OBJECTS 244
DOMAIN SERVICES 246
MODULES 251
AGGREGATES 252
AGGREGATE ROOT 254
BEHAVIOR-FOCUSED AGGREGATE ROOT 255
MODIFY AND COMMIT ONLY ONE AGGREGATE INSTANCE IN ONE TRANSACTION 256
PROTECT BUSINESS INVARIANTS INSIDE AGGREGATE BOUNDARIES 257
REFERENCE OTHER AGGREGATES BY IDENTITY ONLY 258
FACTORIES 259
REPOSITORIES 261
SUPPLE DESIGN 265
UBIQUITOUS LANGUAGE 266
INTENTION-REVEALING INTERFACES 267
SIDE-EFFECT FREE FUNCTIONS 268
ASSERTIONS 269
CONCEPTUAL CONTOURS 270
STANDALONE CLASSES 271

CLOSURE OF OPERATIONS 272
MODEL INTEGRITY AND CONTEXT 273
BOUNDED CONTEXT 274
CONTINUOUS INTEGRATION 275
STRATEGIC CONTEXT MAP 276
CONTEXTUAL MAP 277
SHARED KERNEL 278
CUSTOMER-SUPPLIER TEAMS 279
CONFORMIST 280
ANTICORRUPTION LAYER 281
SEPARATE WAYS 282
OPEN HOST SERVICE 283
PUBLISHED LANGUAGE 284
DISTILLATION 285
CORE DOMAIN 286
GENERIC SUBDOMAINS 287
DOMAIN VISION STATEMENT 288
HIGHLIGHTED CORE 289
COHESIVE MECHANISMS 290
SEGREGATED CORE 291
ABSTRACT CORE 292
LARGE-SCALE STRUCTURE 293
EVOLVING ORDER 294
SYSTEM METAPHOR 295
RESPONSIBILITY LAYERS 296
KNOWLEDGE LEVEL 300
PLUGGABLE COMPONENT FRAMEWORK 301
ADDITIONAL PATTERNS 302
TYPES OF CONSISTENCY 303
EVENT SOURCING 304
EVENT PROCESSOR 305
EVENT DISPATCHER 306
INTERNAL DOMAIN EVENTS 307
EXTERNAL DOMAIN EVENTS, TRANSFER BETWEEN CONTEXTS 308
STATIC DOMAIN EVENTS CLASS 309
ONE SUBDOMAIN PER BOUNDED CONTEXT 310
THE APPLICATION LAYER COORDINATES THE WORK BETWEEN CONTEXTS 311
THE SAME PHYSICAL ENTITY IN DIFFERENT CONTEXTS 312
INTEGRATION OF BOUDED CONTEXTS THROUGH DATABASE 313
INTEGRATION OF BOUDED CONTEXTS THROUGH FLAT FILES 314
INTEGRATION OF BOUDED CONTEXTS THROUGH ENTERPRISE SERVICE BUS 315
INTEGRATION OF BOUDED CONTEXTS THROUGH MESSAGE QUEUE 316
DEPENDENCY INJECTION 317

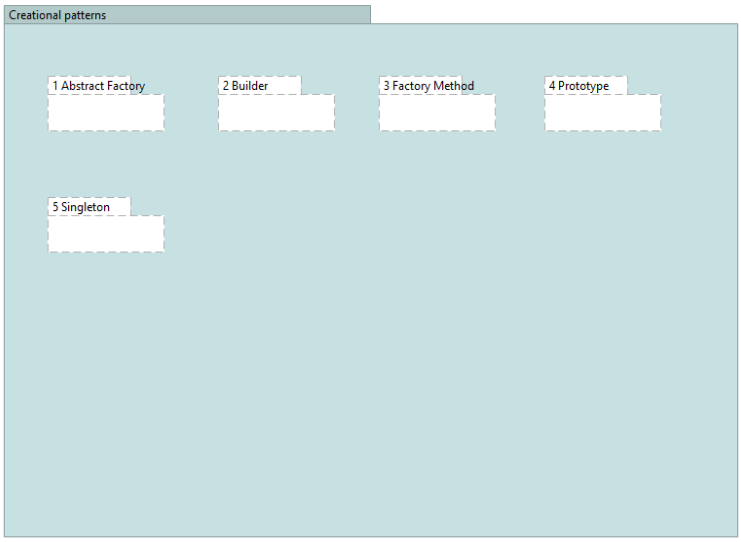
318
DEPENDENCY INVERSION 319
INVERSION OF CONTROL 320
SERVICE LOCATOR 321
CQRS 322
CQS 323
WRAP LOW-LEVEL EXCEPTIONS 324
EXTRACT DEPENDENCY FROM INTERFACE TO COSNTRUCTOR 325
INTERFACE SEGREGATION 326
CLEAN ARCHITECTURE 327

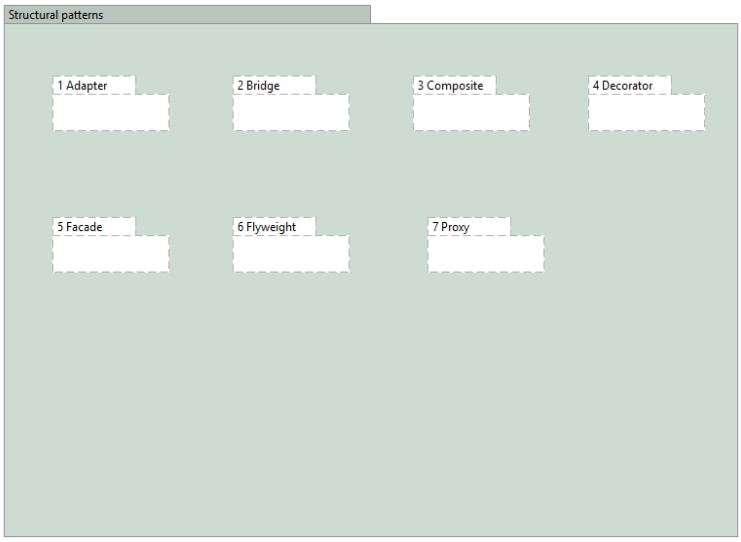
01

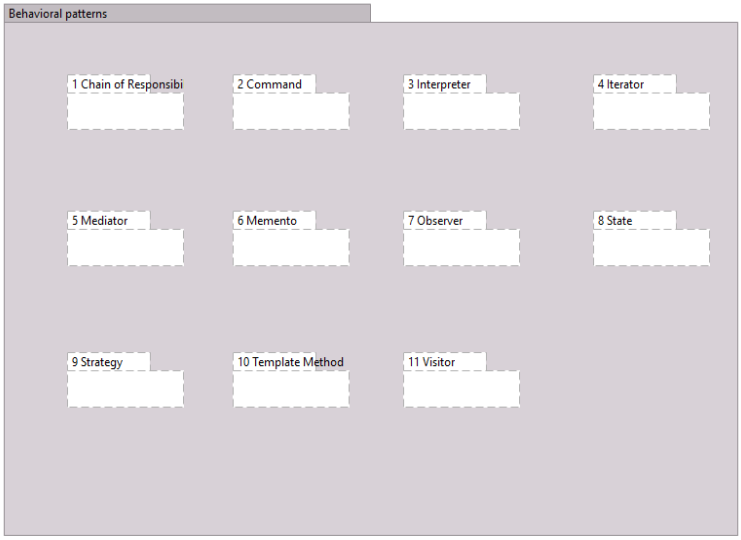
Design Patterns

Elements of Reusable Object-Oriented Software

GANG OF FOUR

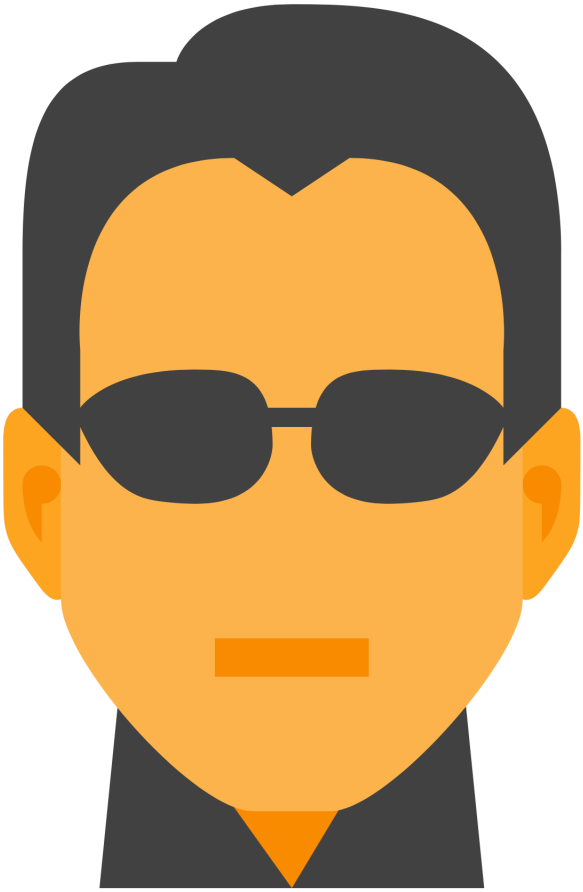




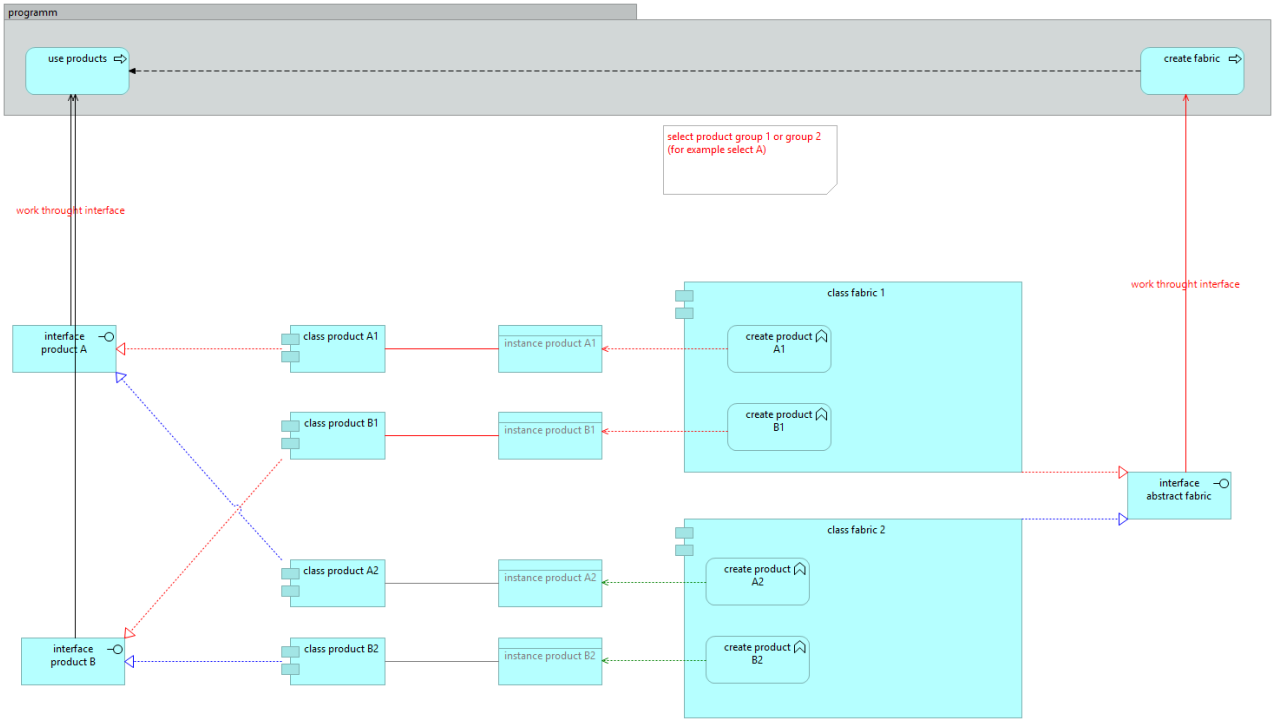


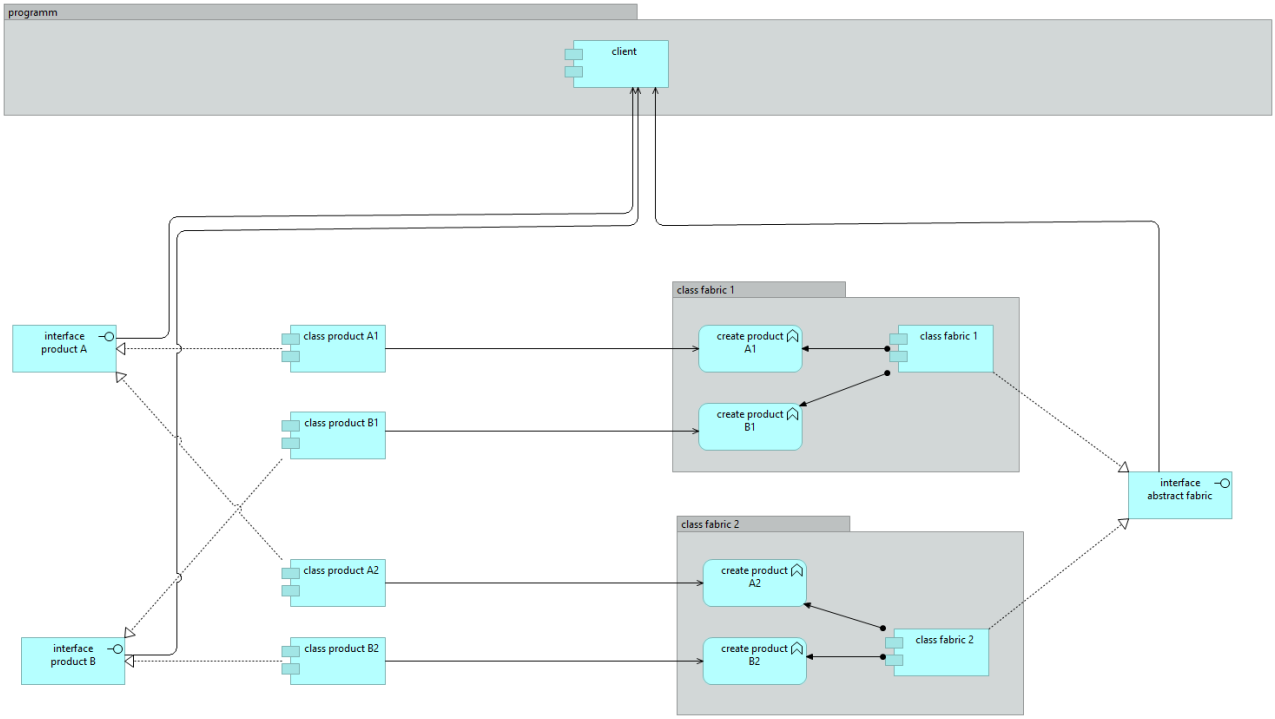
CREATIONAL PATTERNS

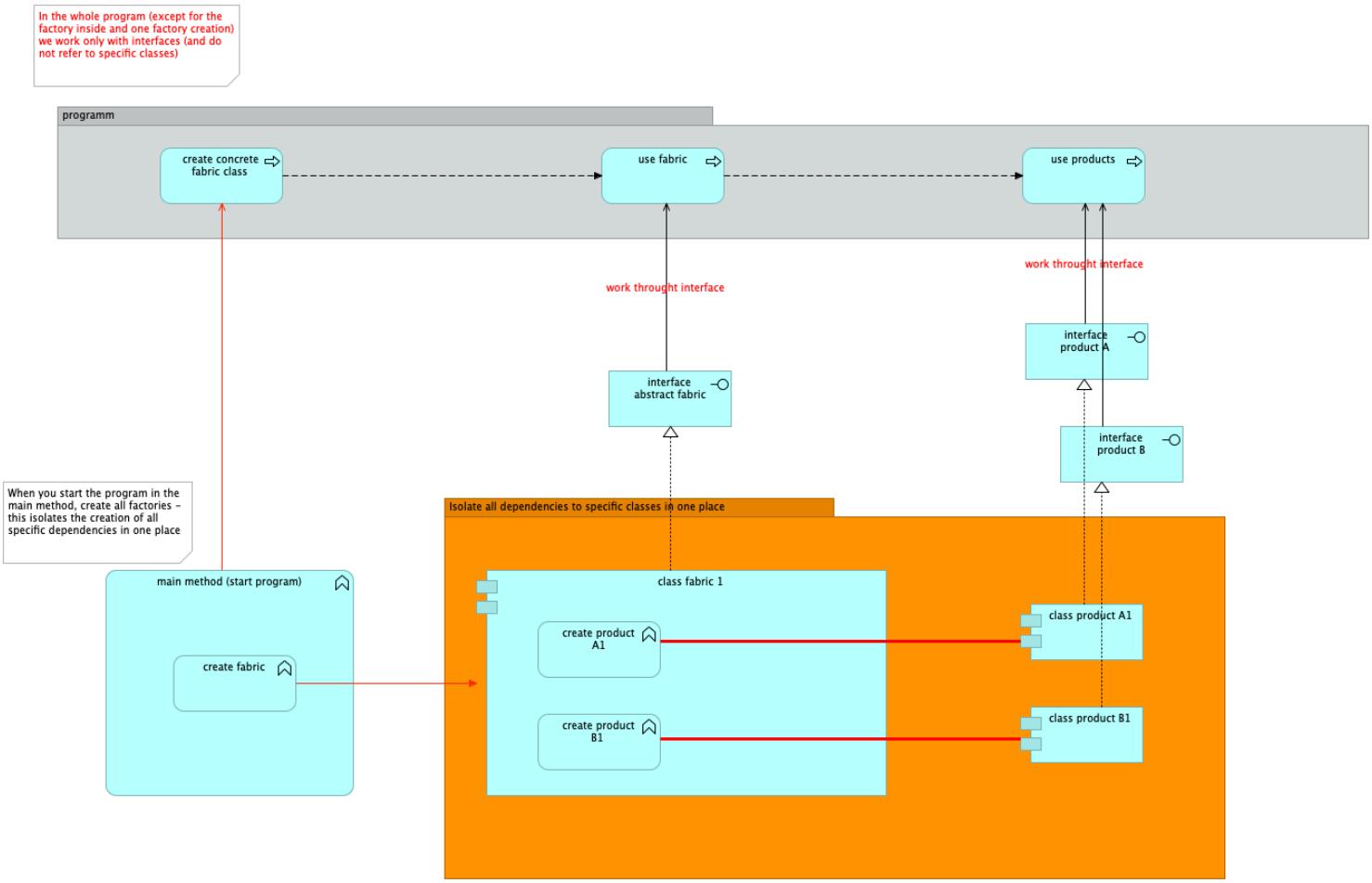
DESIGN PATTERNS | GoF



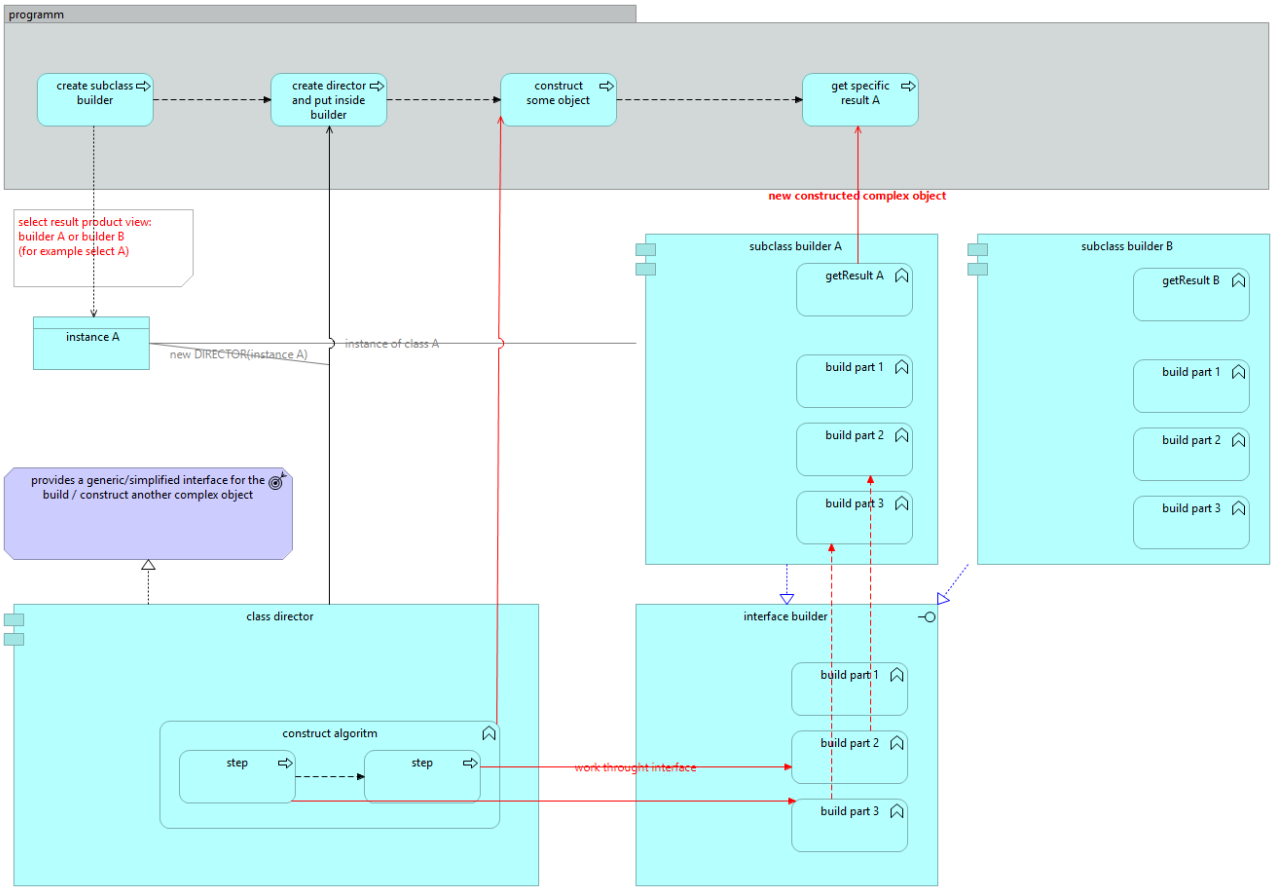
ABSTRACT FACTORY

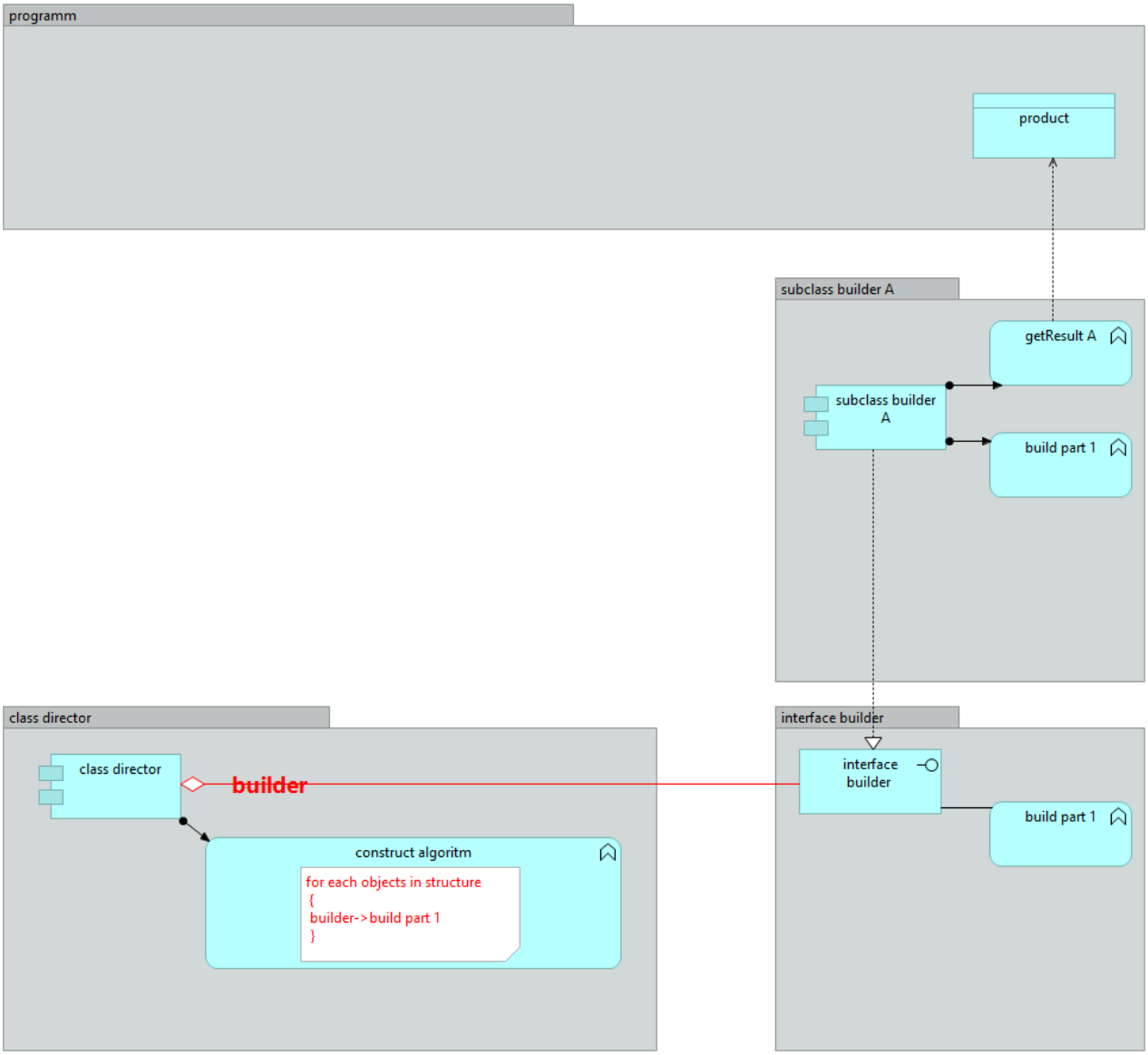



programm
client
class fabric 1
interface product A
class product A1
create product A1
class fabric 1
class product B1
create product B1
interface abstract fabric
class fabric 2
class product A2
create product A2
class fabric 2
interface product B
class product B2
create product B2

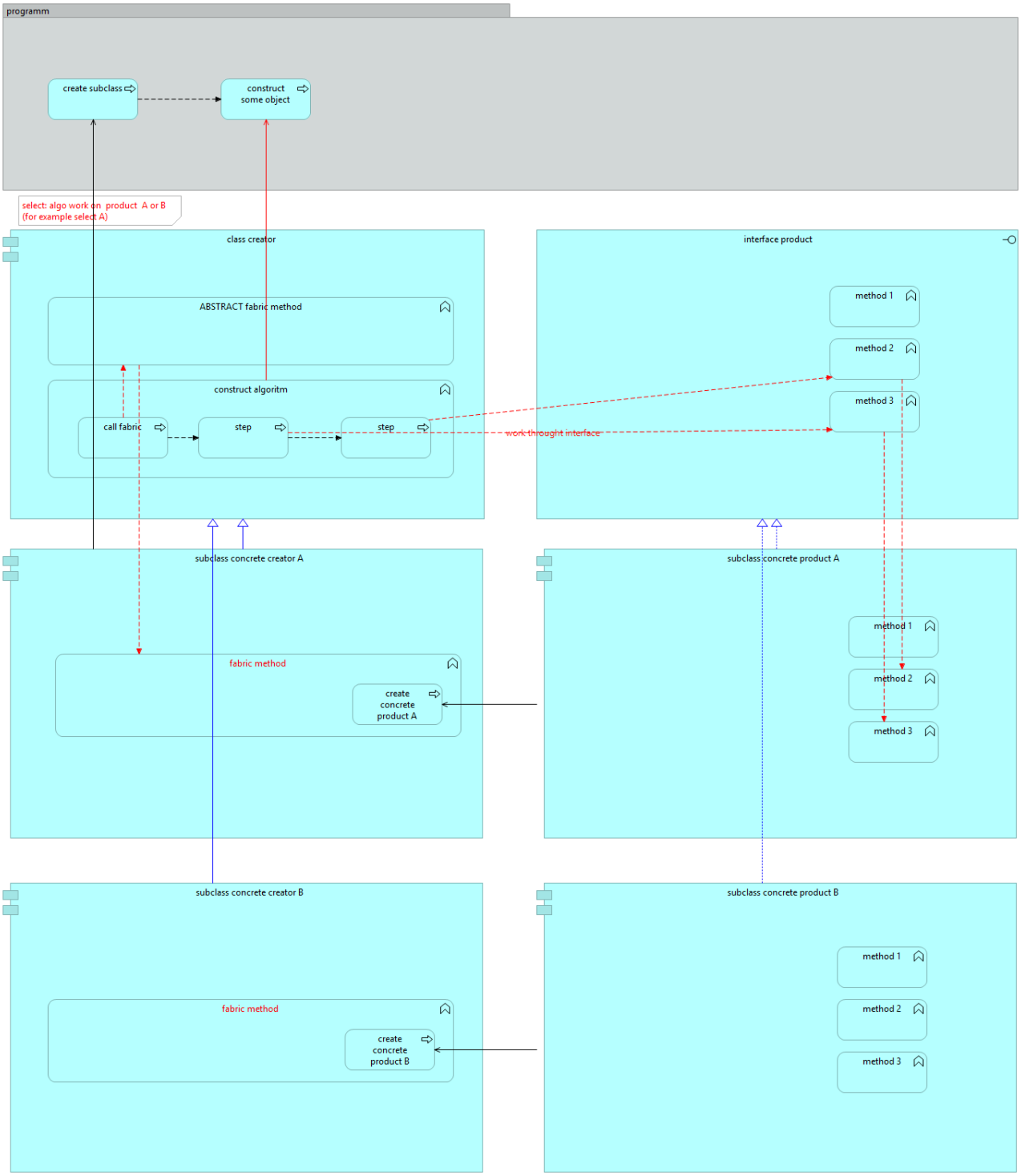

In the whole program (except for the factory inside and one factory creation) we work only with interfaces (and do not refer to specific classes)
programm
create concrete fabric class
use fabric
use products
work throught interface
work throught interface
interface product A
interface abstract fabric
interface product B
When you start the program in the main method, create all factories – this isolates the creation of all specific dependencies in one place
Isolate all dependencies to specific classes in one place
main method (start program)
create fabric
class fabric 1
create product A1
create product B1
class product A1
class product B1

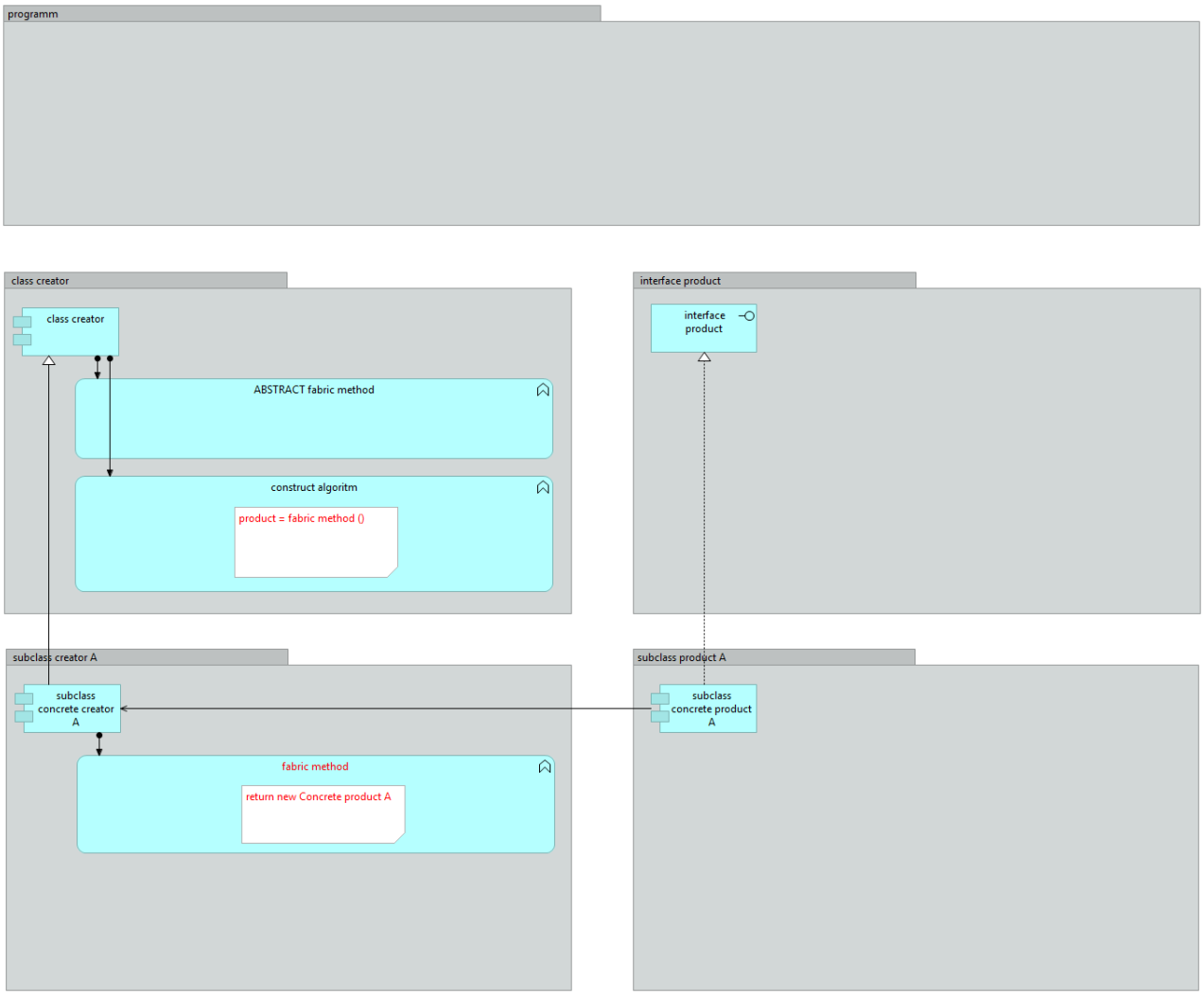
BUILDER


programm
create subclass builder
create director and put inside builder
construct some object
get specific result A
new constructed complex object
select result product view:
builder A or builder B
(for example select A)
instance A
new DIRECTOR(instance A)
instance of class A
subclass builder A
getResult A
build part 1
build part 2
build part 3
subclass builder B
getResult B
build part 1
build part 2
build part 3
provides a generic/simplified interface for the build / construct another complex object
class director
construct algoritm
step
step
work throught interface
interface builder
build part 1
build part 2
build part 3

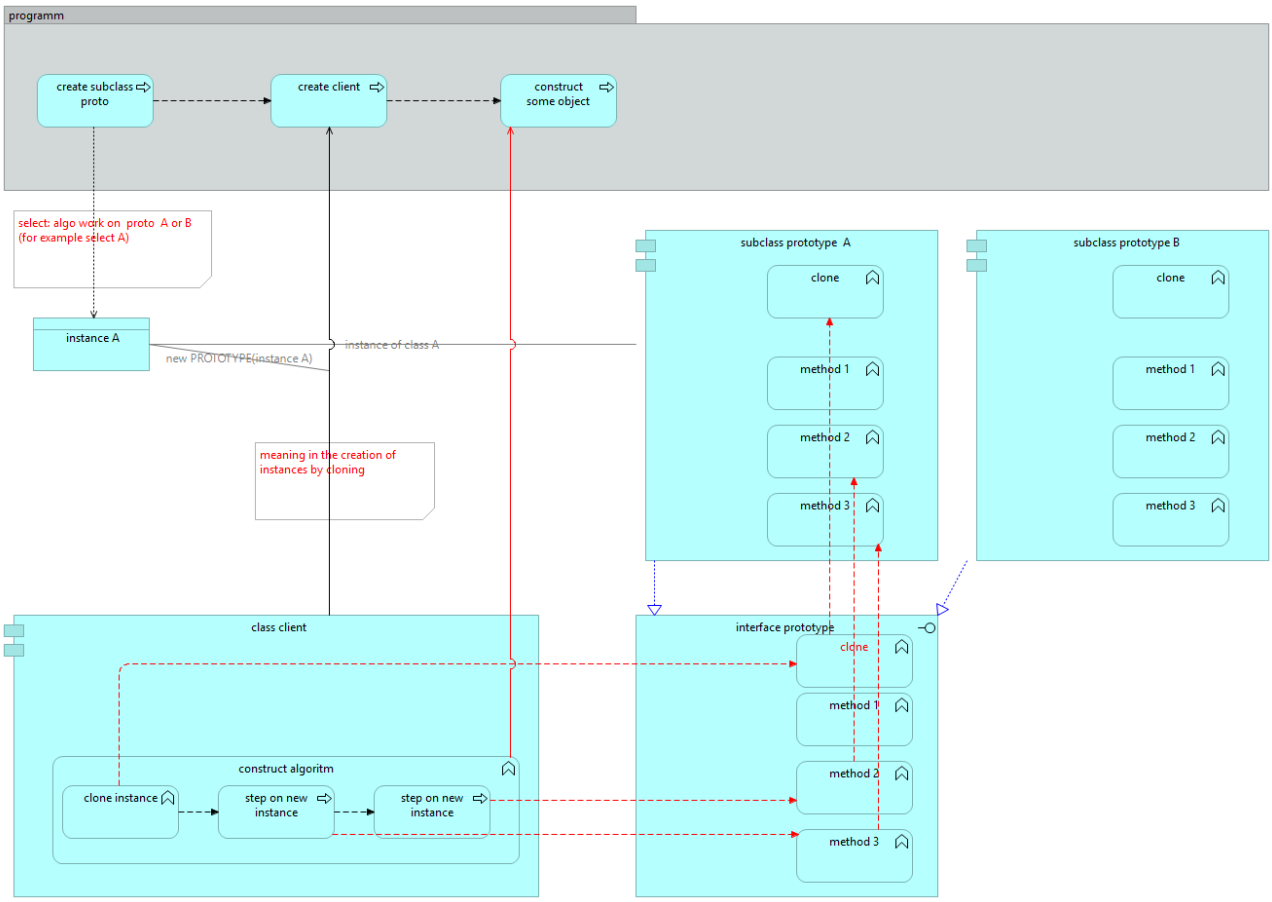

programm
product
subclass builder A
getResult A
subclass builder A
build part 1
class director
class director
builder
construct algoritm
for each objects in structure
{
builder->build part 1
}
interface builder
interface builder
build part 1

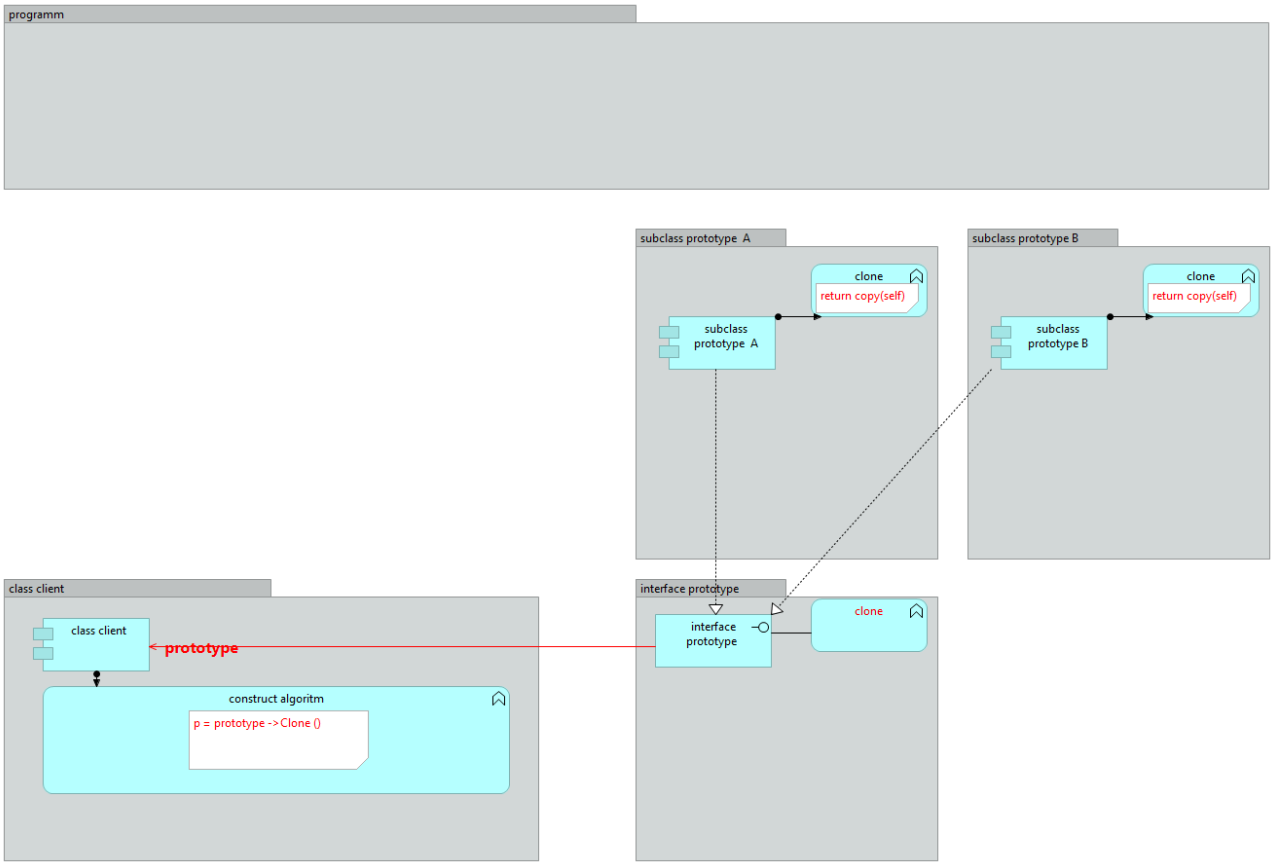
FACTORY METHOD



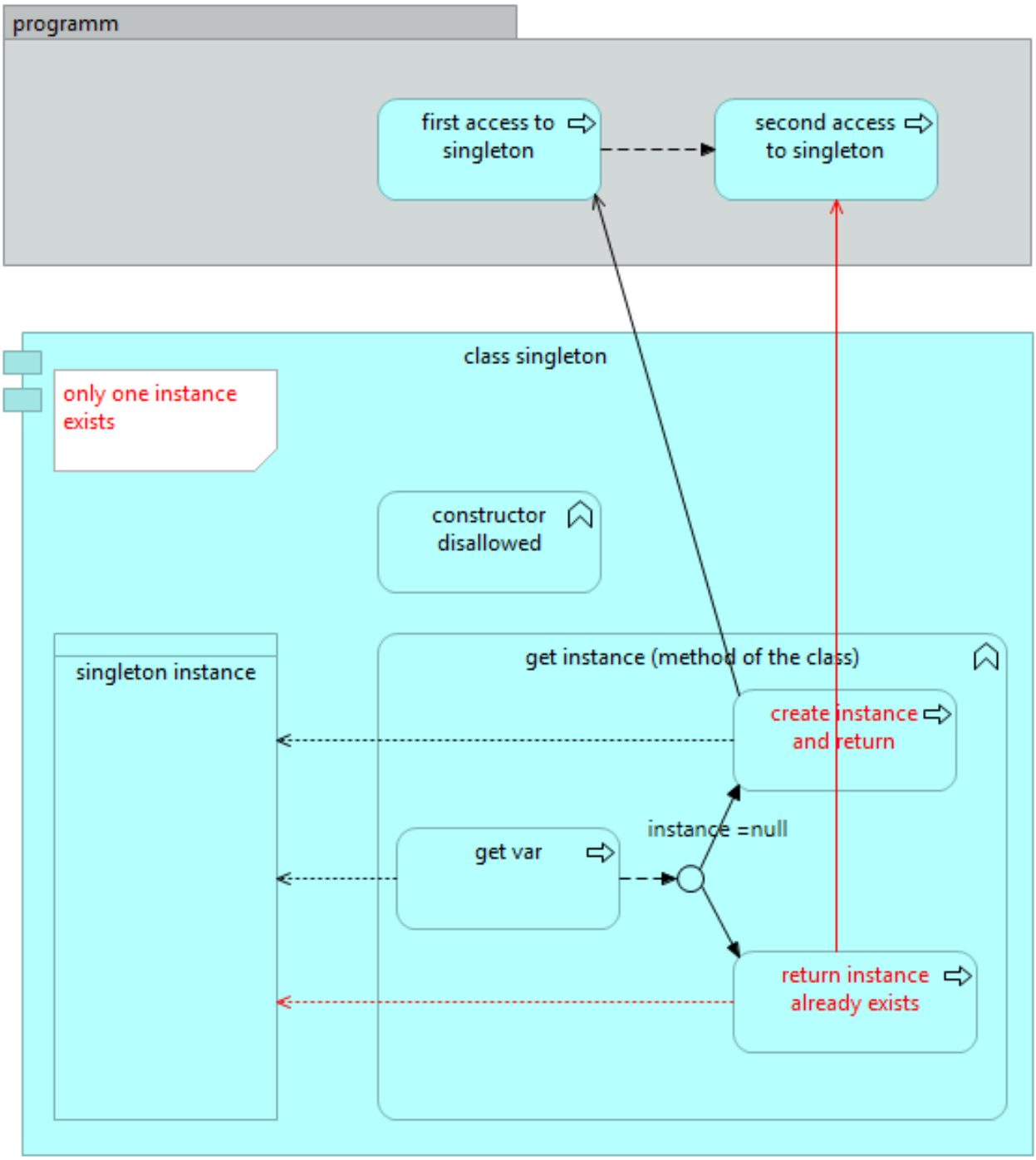

programm
class creator
class creator
ABSTRACT fabric method
construct algoritm
product = fabric method ()
interface product
interface
product
subclass creator A
subclass
concrete creator
A
fabric method
return new Concrete product A
subclass product A
subclass
concrete product
A

PROTOTYPE

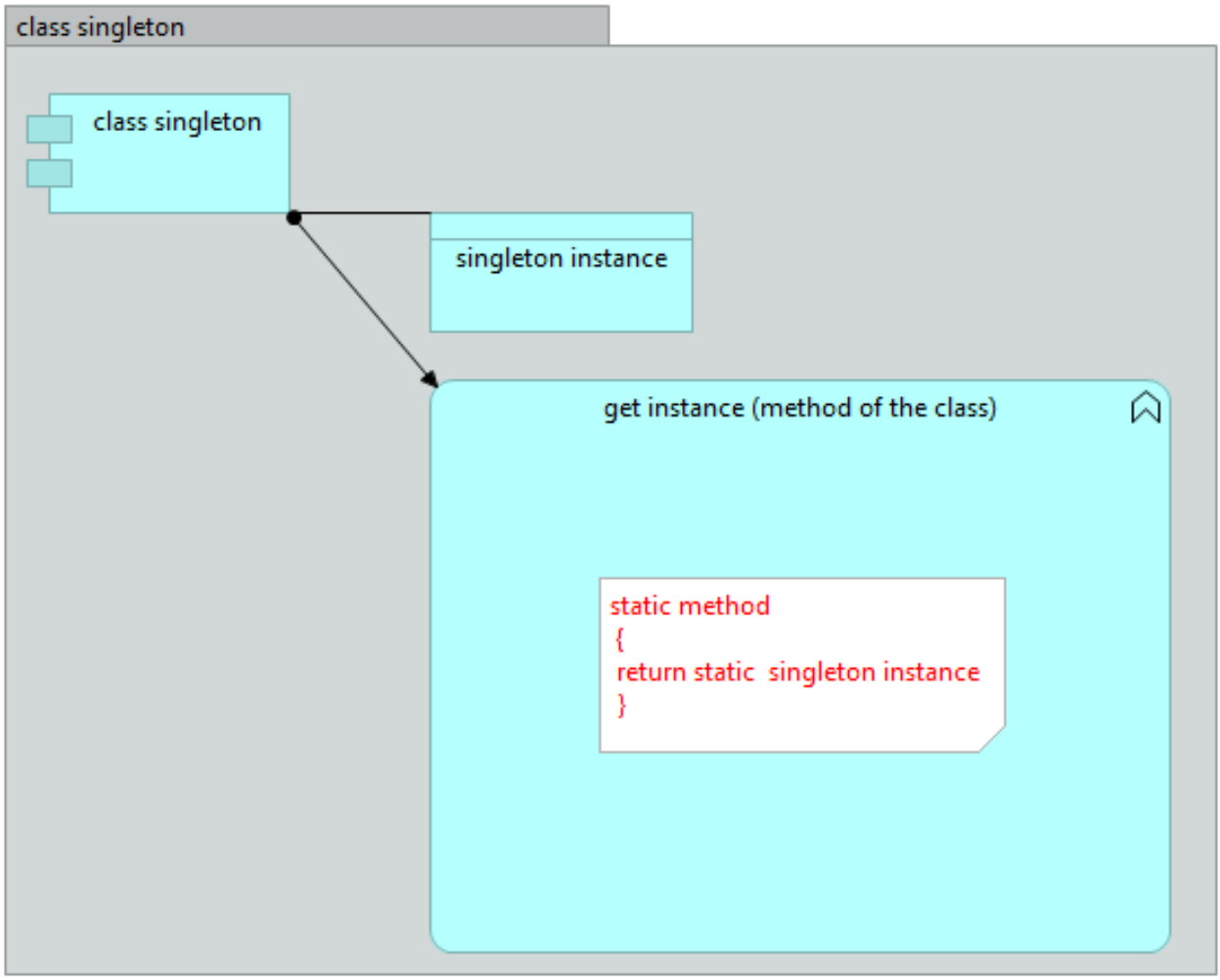

programm
create subclass proto
create client
construct some object
select: algo work on proto A or B (for example select A)
instance A
new PROTOTYPE(instance A)
instance of class A
meaning in the creation of instances by cloning
subclass prototype A
clone
method 1
method 2
method 3
subclass prototype B
clone
method 1
method 2
method 3
class client
construct algoritm
clone instance
step on new instance
step on new instance
interface prototype
clone
method 1
method 2
method 3


programm
subclass prototype A
clone
return copy(self)
subclass
prototype A
subclass prototype B
clone
return copy(self)
subclass
prototype B
class client
class client
prototype
construct algoritm
p = prototype -> Clone ()
interface prototype
interface
prototype
clone

SINGLETON

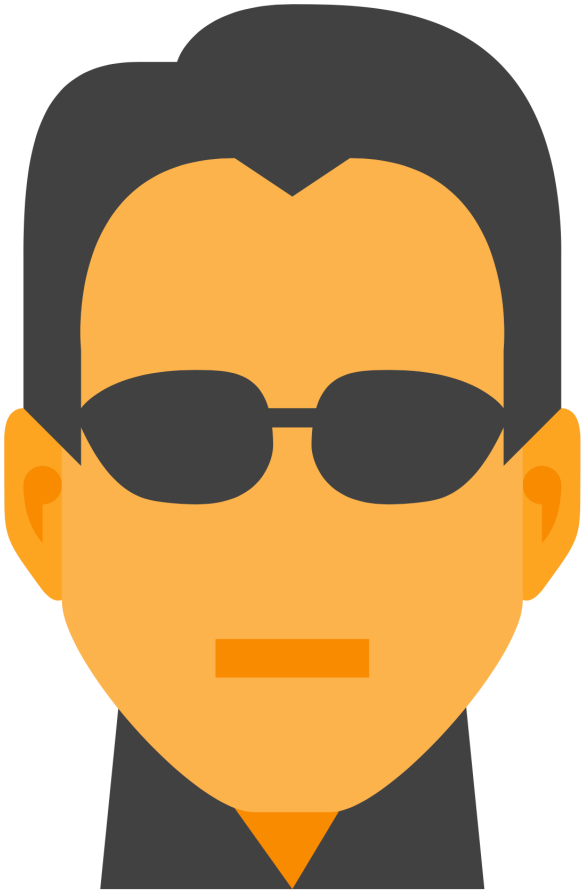

programm
first access to singleton
second access to singleton
class singleton
only one instance exists
constructor disallowed
singleton instance
get instance (method of the class)
create instance and return
instance =null
get var
return instance already exists

programm

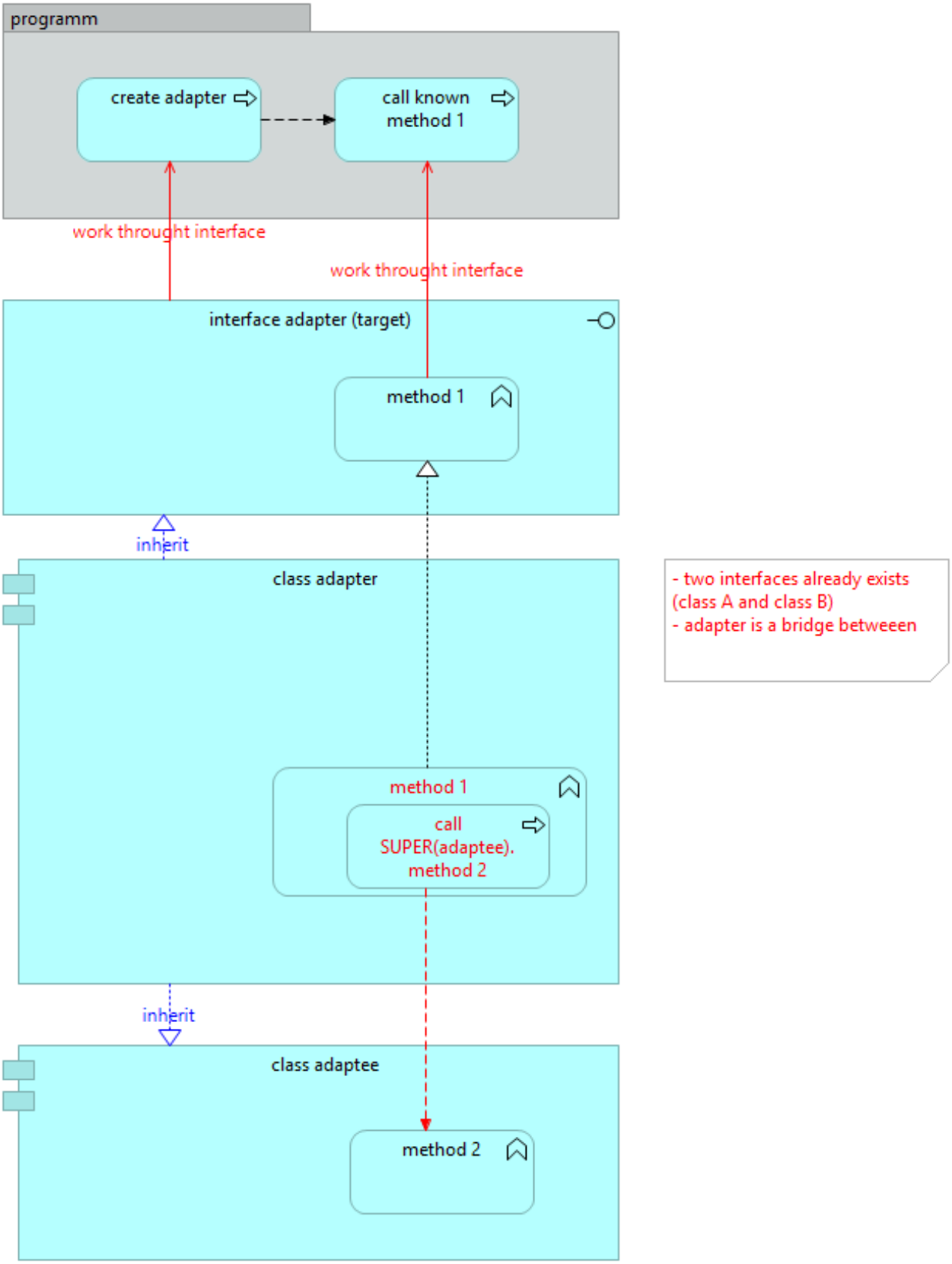

class singleton
class singleton
singleton instance
get instance (method of the class)
static method
{
return static singleton instance
}

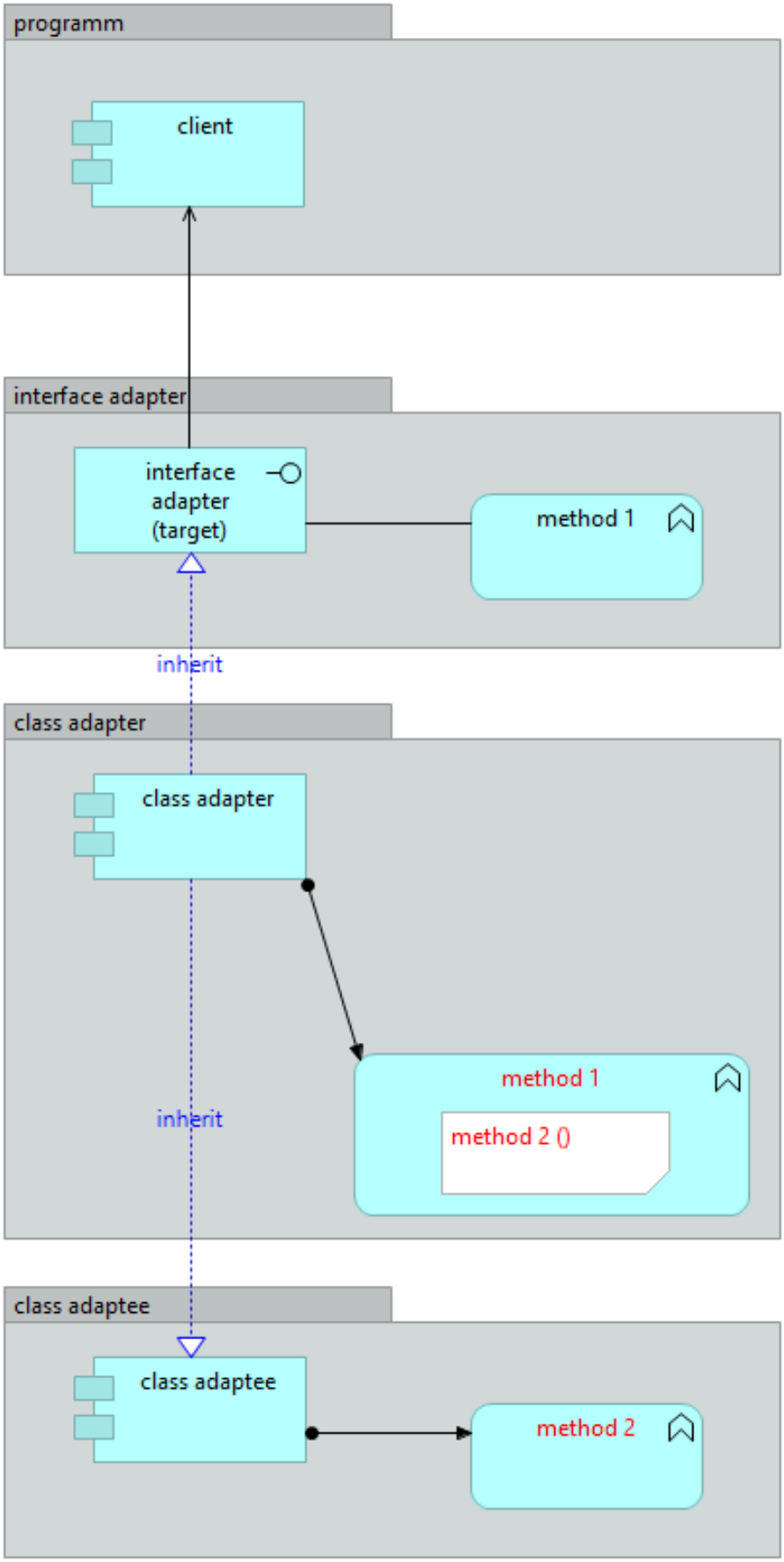
STRUCTURAL PATTERNS

DESIGN PATTERNS | GoF

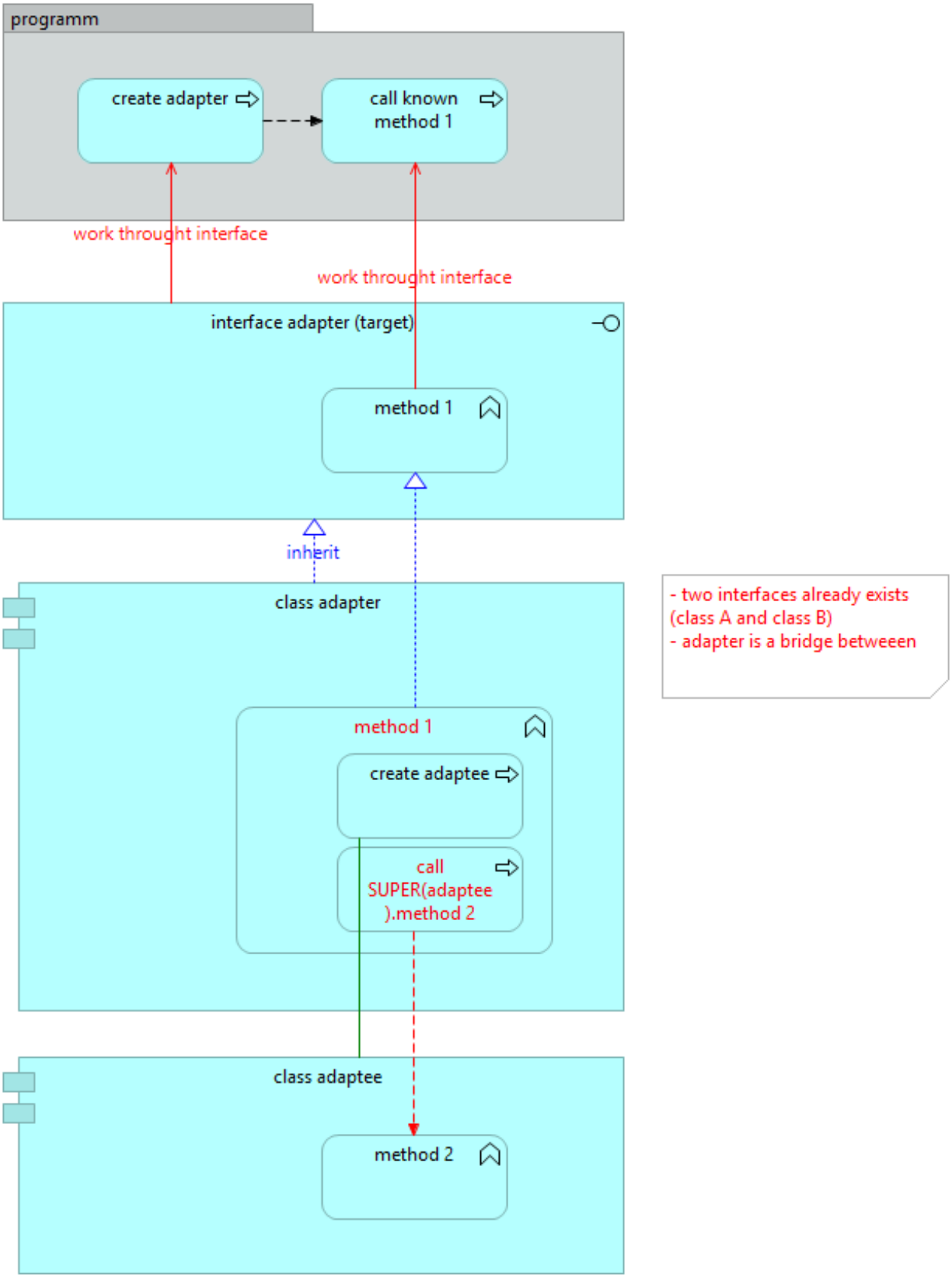


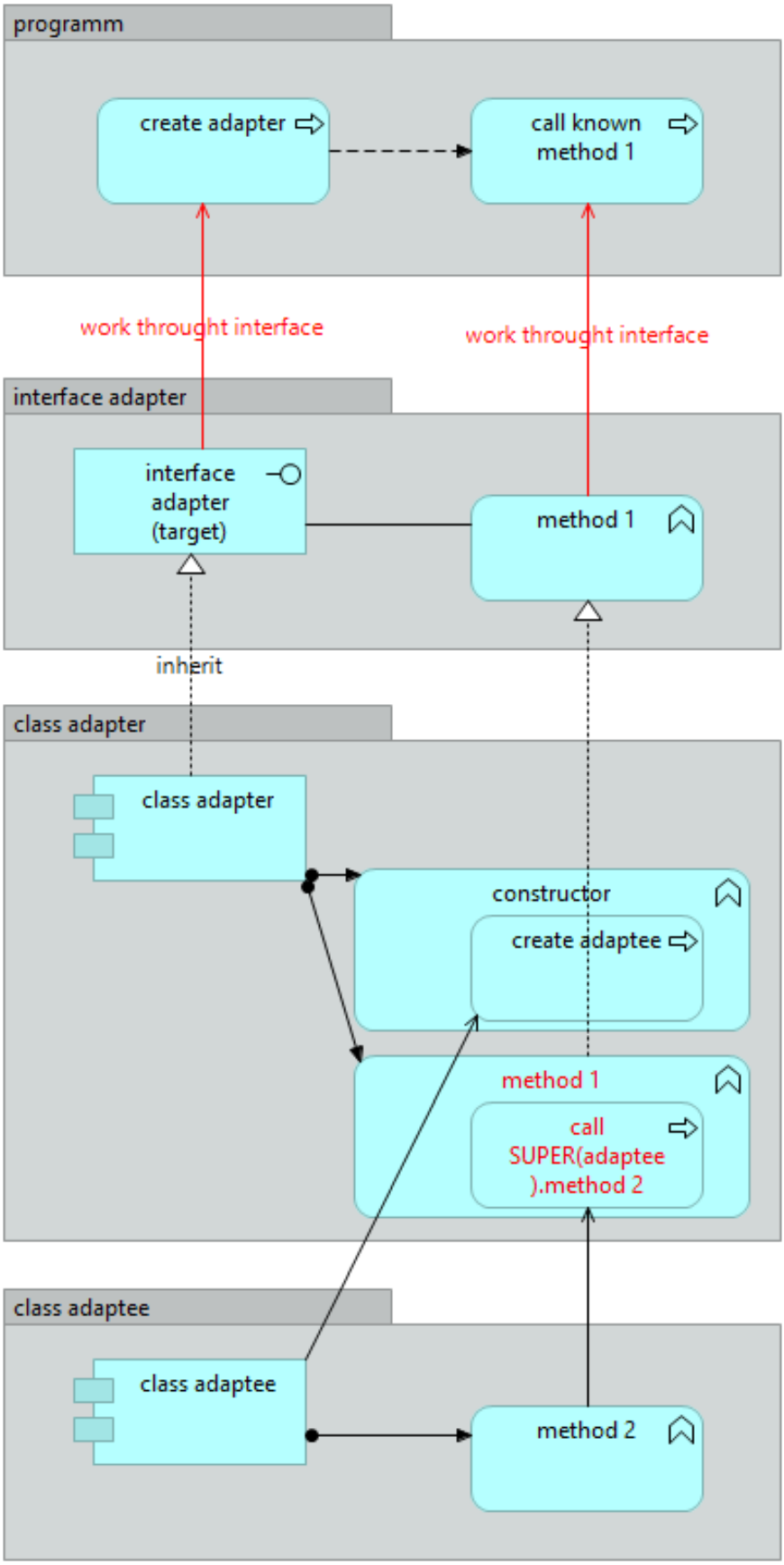
ADAPTER OF CLASS

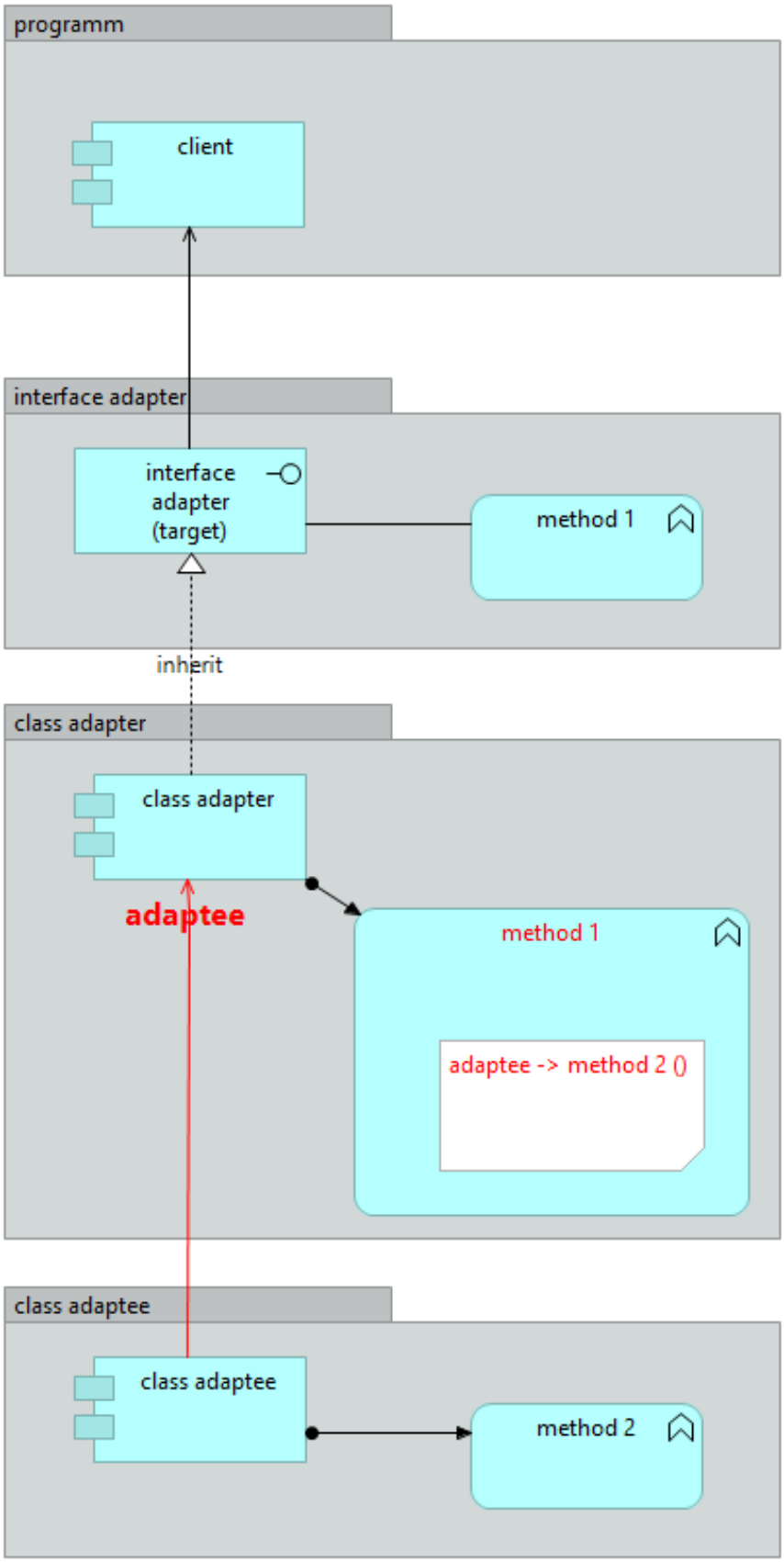



programm
client
interface adapter
interface
adapter
(target)
method 1
inherit
class adapter
class adapter
method 1
method 2 ()
inherit
class adaptee
class adaptee
method 2

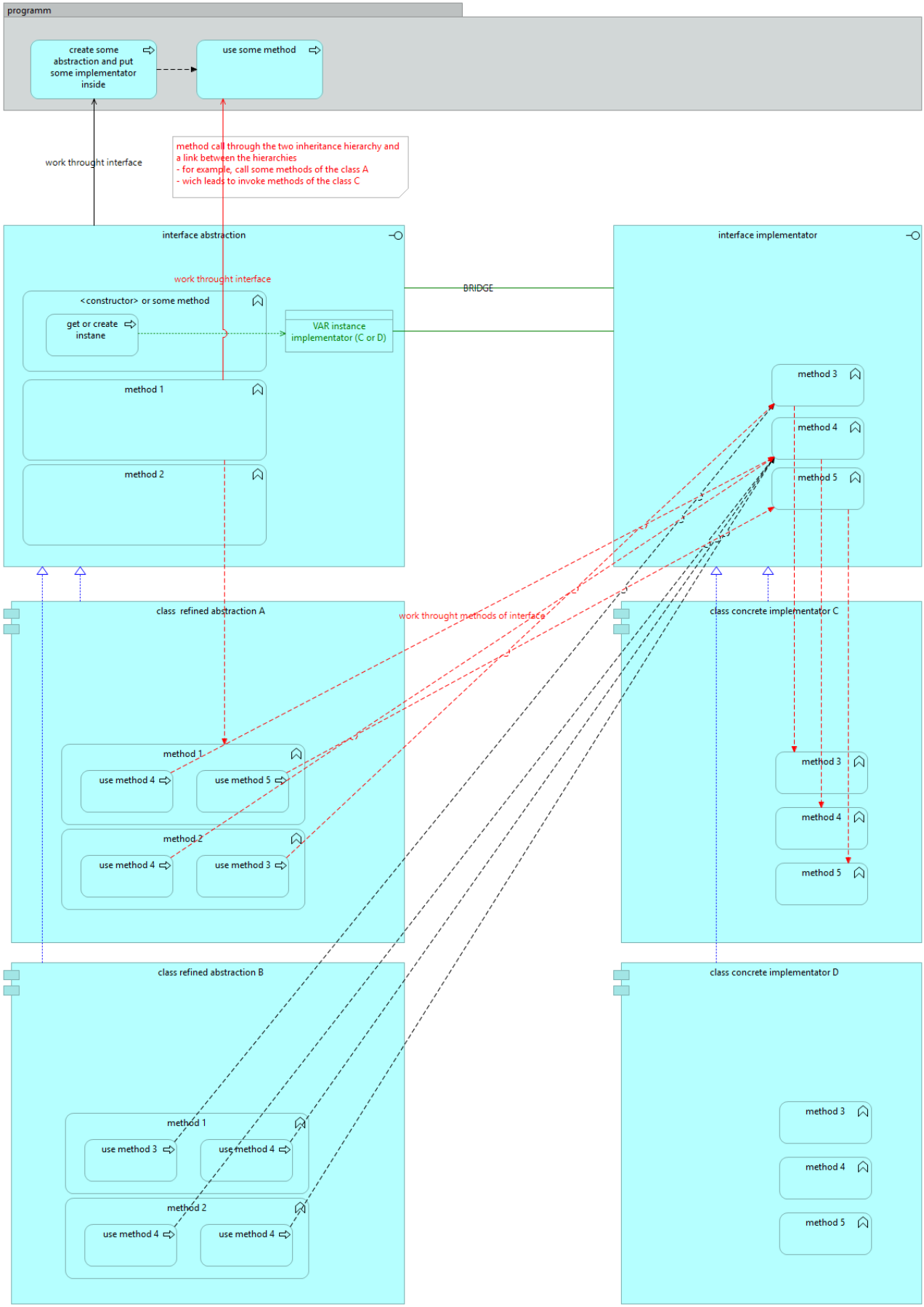
ADAPTER OF OBJECT

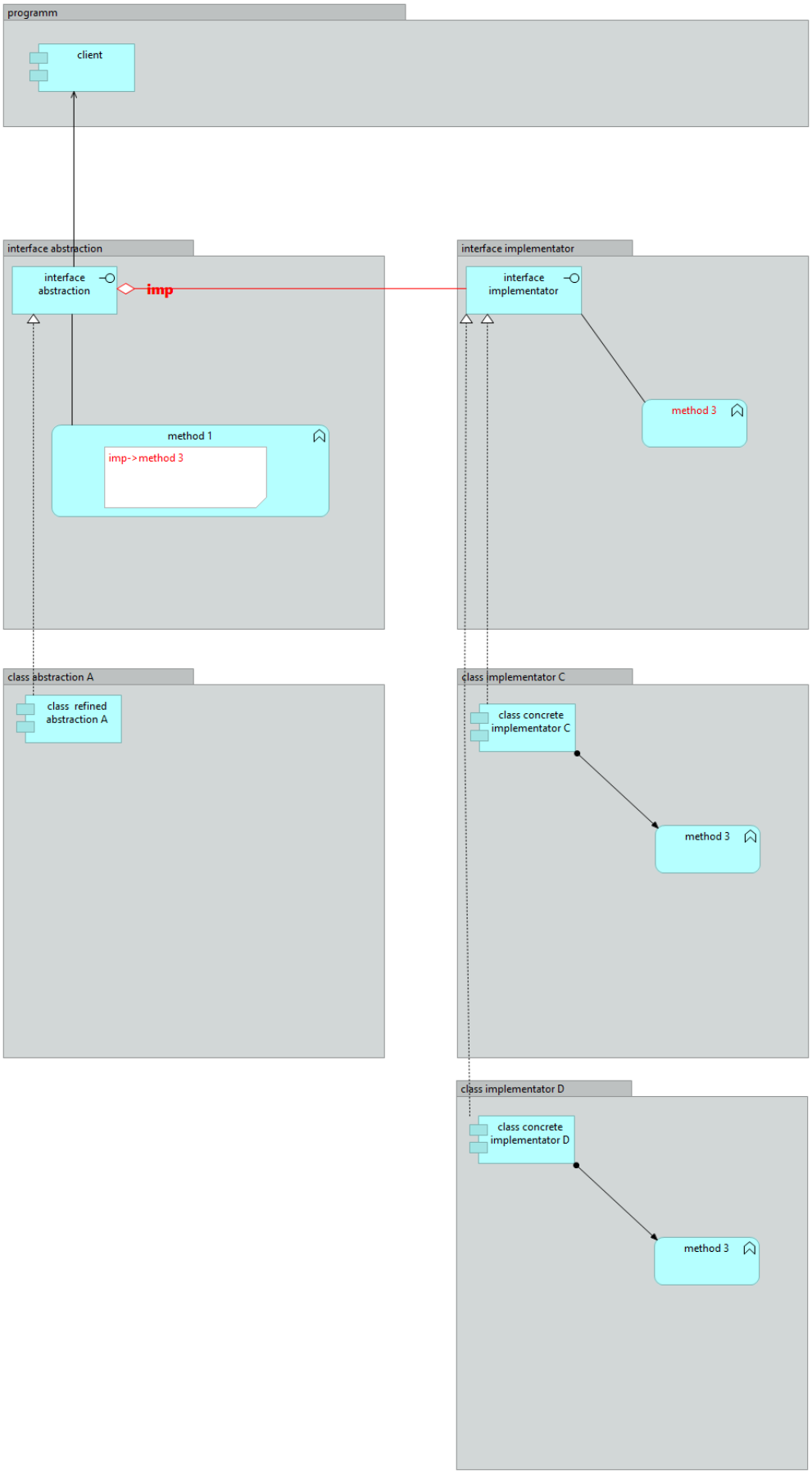



programm
create adapter
call known method 1
work throught interface
work throught interface
interface adapter
interface adapter (target)
method 1
inherit
class adapter
class adapter
constructor
create adaptee
method 1
call SUPER(adaptee).method 2
class adaptee
class adaptee
method 2

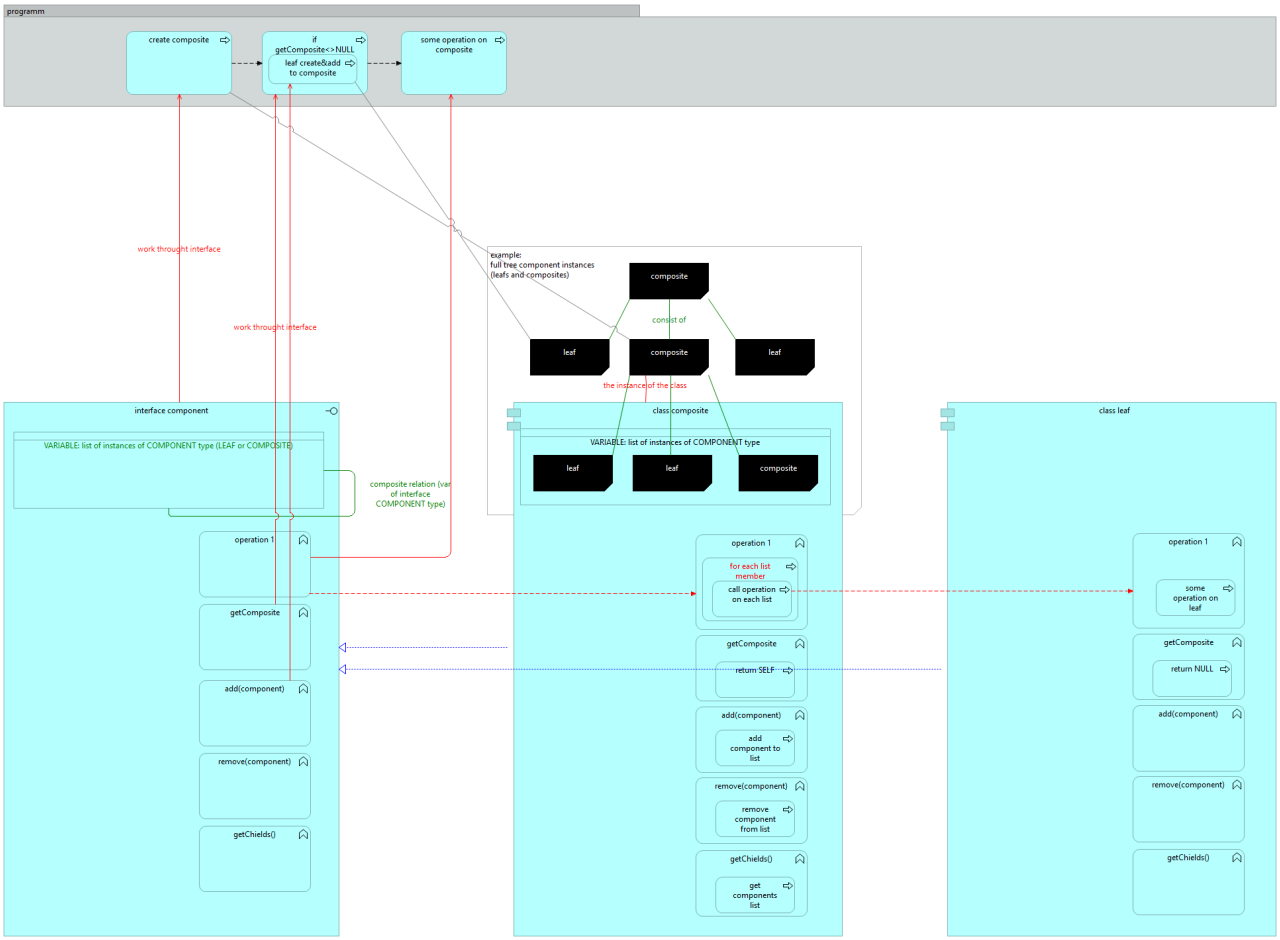

programm
client
interface adapter
interface
adapter
(target)
method 1
inherit
class adapter
class adapter
adaptee
method 1
adaptee -> method 2 ()
class adaptee
class adaptee
method 2

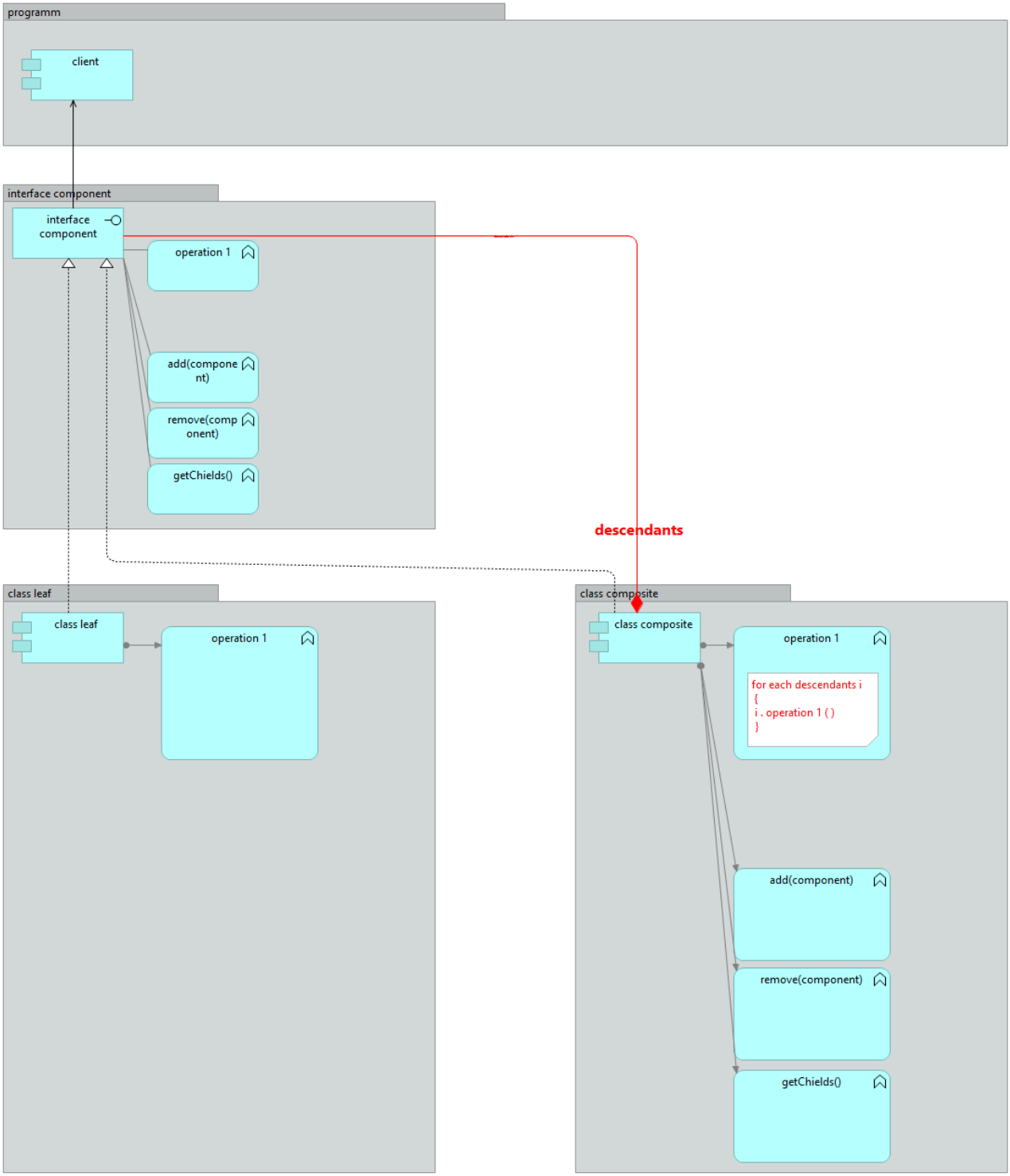
BRIDGE


programm
create some abstraction and put some implementator inside
use some method
work throught interface
method call through the two inheritance hierarchy and a link between the hierarchies
- for example, call some methods of the class A
- wich leads to invoke methods of the class C
interface abstraction
work throught interface
<constructor> or some method
get or create instane
VAR instance implementator (C or D)
method 1
method 2
BRIDGE
interface implementator
method 3
method 4
method 5
class refined abstraction A
work throught methods of interface
method 1
use method 4
use method 5
method 2
use method 4
use method 3
class concrete implementator C
method 3
method 4
method 5
class refined abstraction B
method 1
use method 3
use method 4
method 2
use method 4
use method 4
class concrete implementator D
method 3
method 4
method 5

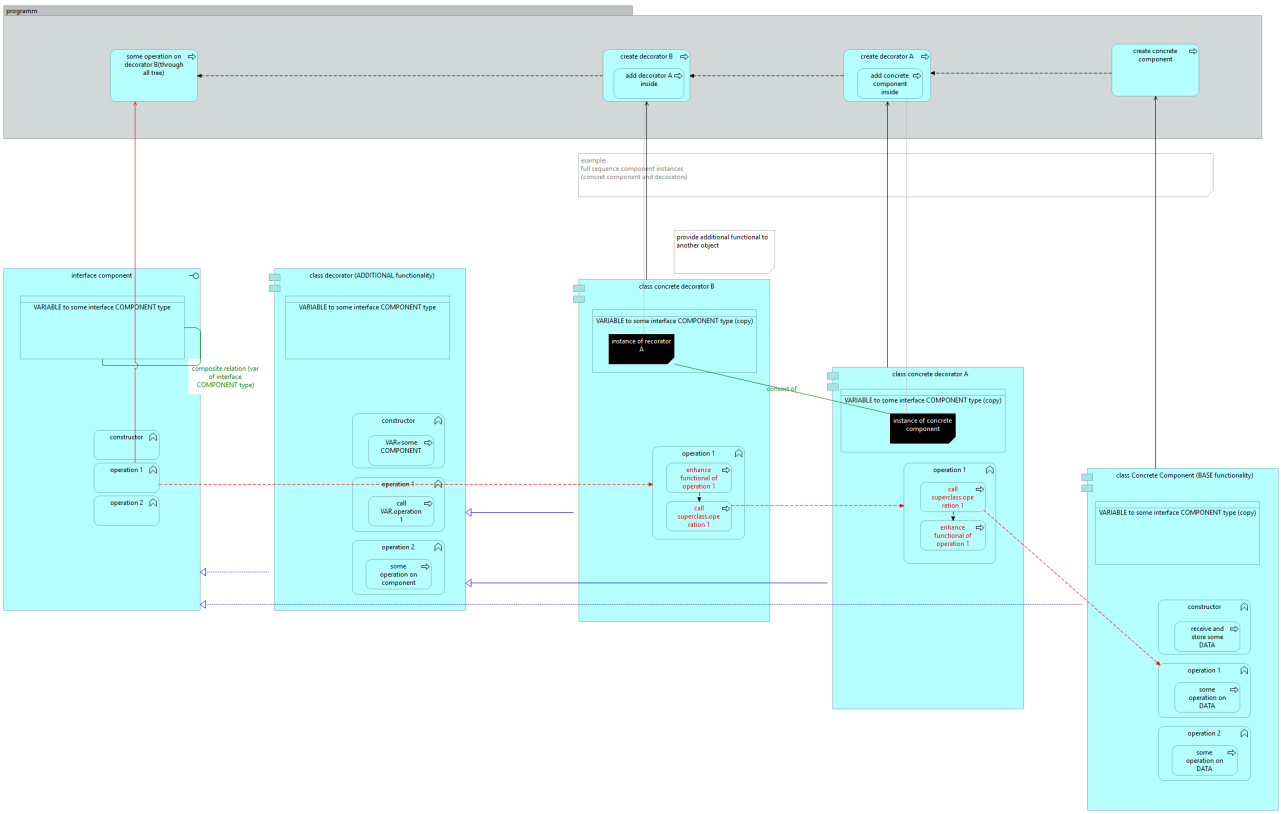

programm
client
interface abstraction
interface abstraction
imp
method 1
imp-> method 3
interface implementator
interface implementator
method 3
class abstraction A
class refined abstraction A
class implementator C
class concrete implementator C
method 3
class implementator D
class concrete implementator D
method 3

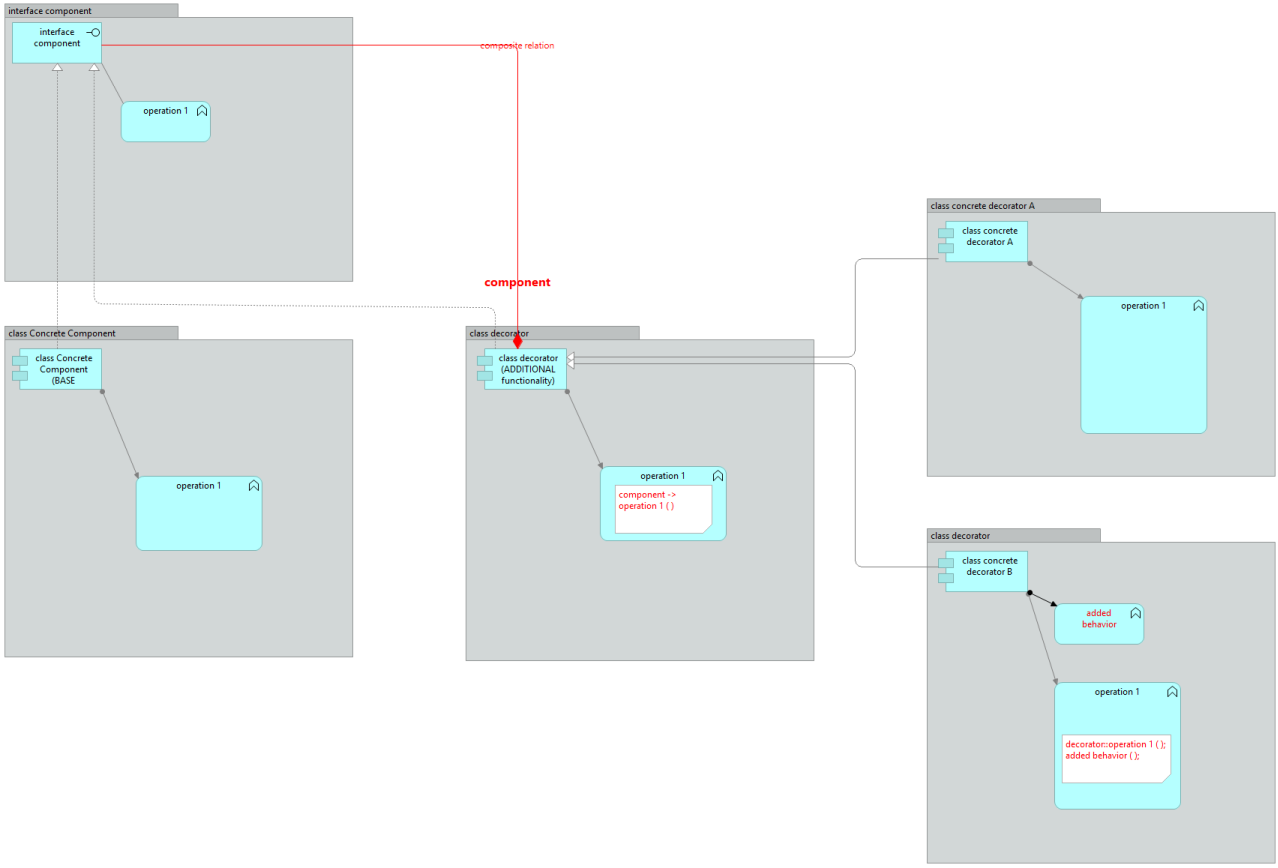
COMPOSITE


programm
create composite
if
getComposite<>NULL
leaf create&add
to composite
some operation on
composite
work throught interface
work throught interface
example:
full tree component instances
(leafs and composites)
composite
consist of
leaf
composite
leaf
the instance of the class
interface component
VARIABLE: list of instances of COMPONENT type (LEAF or COMPOSITE)
composite relation (var
of interface
COMPONENT type)
operation 1
getComposite
add(component)
remove(component)
getChields()
class composite
VARIABLE: list of instances of COMPONENT type
leaf
leaf
composite
operation 1
for each list
member
call operation
on each list
getComposite
return SELF
add(component)
add
component to
list
remove(component)
remove
component
from list
getChields()
get
components
list
class leaf
operation 1
some
operation on
leaf
getComposite
return NULL
add(component)
remove(component)
getChields()

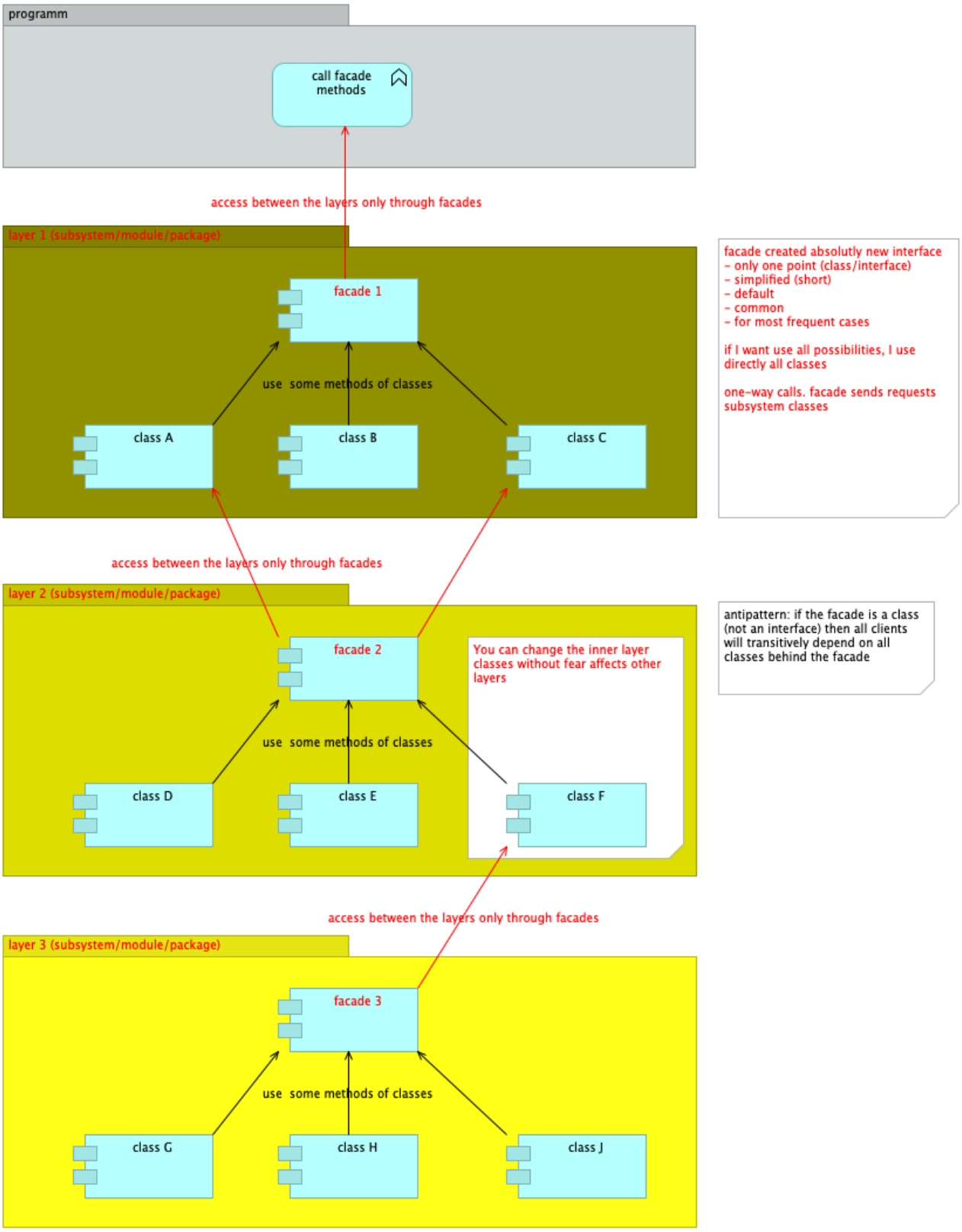

programm
client
interface component
interface component
operation 1
add(component)
remove(component)
getChields()
descendants
class leaf
class leaf
operation 1
class composite
class composite
operation 1
for each descendants i
{
i . operation 1 ( )
}
add(component)
remove(component)
getChields()

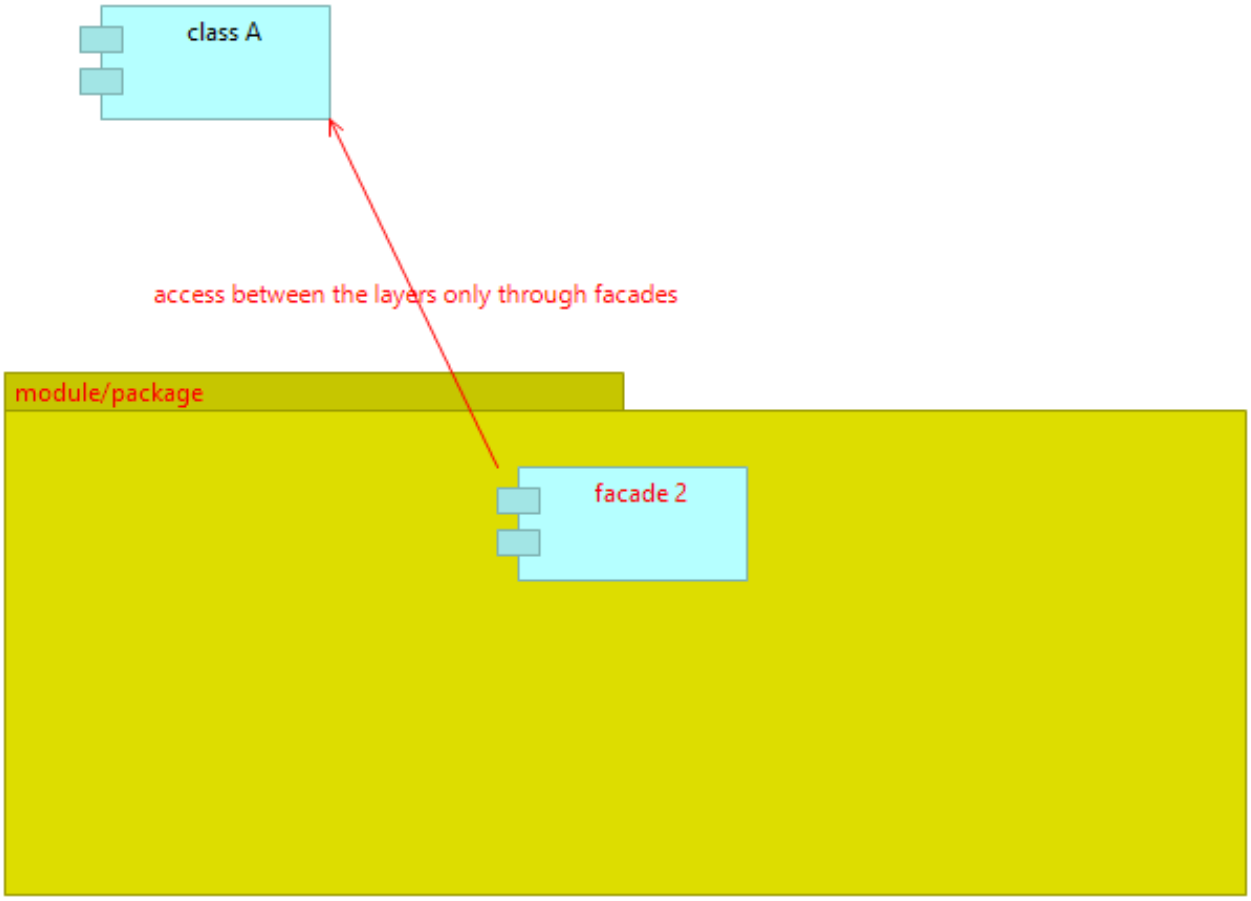
DECORATOR

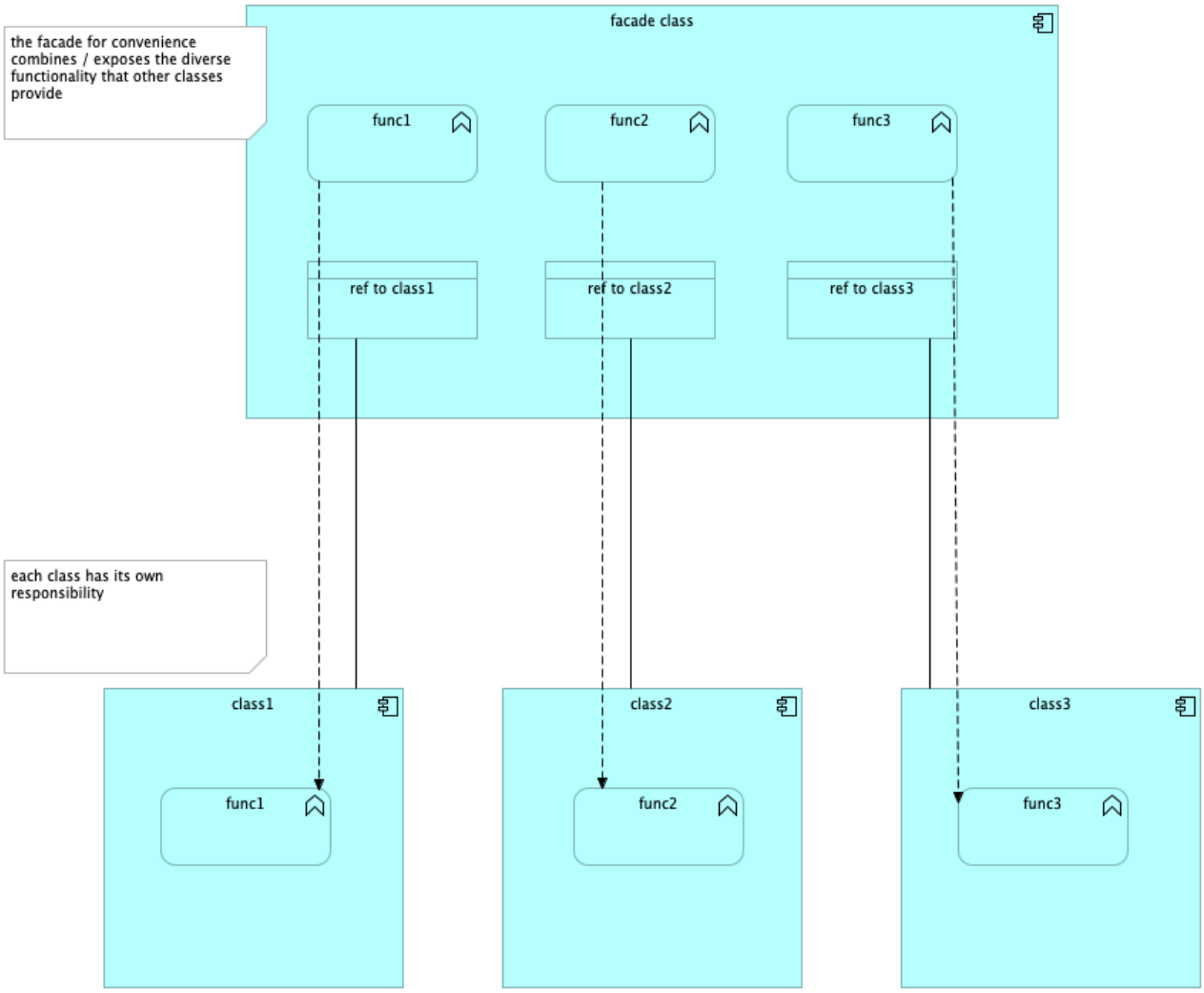

programm
some operation on decorator B(through all tree)
create decorator B
add decorator A inside
create decorator A
add concrete component inside
create concrete component
example:
full sequence component instances
(concret component and decorators)
provide additional functional to another object
interface component
VARIABLE to some interface COMPONENT type
composite relation (var of interface COMPONENT type)
constructor
operation 1
operation 2
class decorator (ADDITIONAL functionality)
VARIABLE to some interface COMPONENT type
constructor
VAR=some COMPONENT
operation 1
call VAR.operation 1
operation 2
some operation on component
class concrete decorator B
VARIABLE to some interface COMPONENT type (copy)
instance of recorator A
operation 1
enhance functional of operation 1
call superclass.operation 1
connect of
class concrete decorator A
VARIABLE to some interface COMPONENT type (copy)
instance of concrete component
operation 1
call superclass.operation 1
enhance functional of operation 1
class Concrete Component (BASE functionality)
VARIABLE to some interface COMPONENT type (copy)
constructor
receive and store some DATA
operation 1
some operation on DATA
operation 2
some operation on DATA


interface component
interface component
operation 1
composite relation
component
class Concrete Component
class Concrete Component (BASE
operation 1
class decorator
class decorator (ADDITIONAL functionality)
operation 1
component -> operation 1 ( )
class concrete decorator A
class concrete decorator A
operation 1
class decorator
class concrete decorator B
added behavior
operation 1
decorator::operation 1 ( ); added behavior ( );

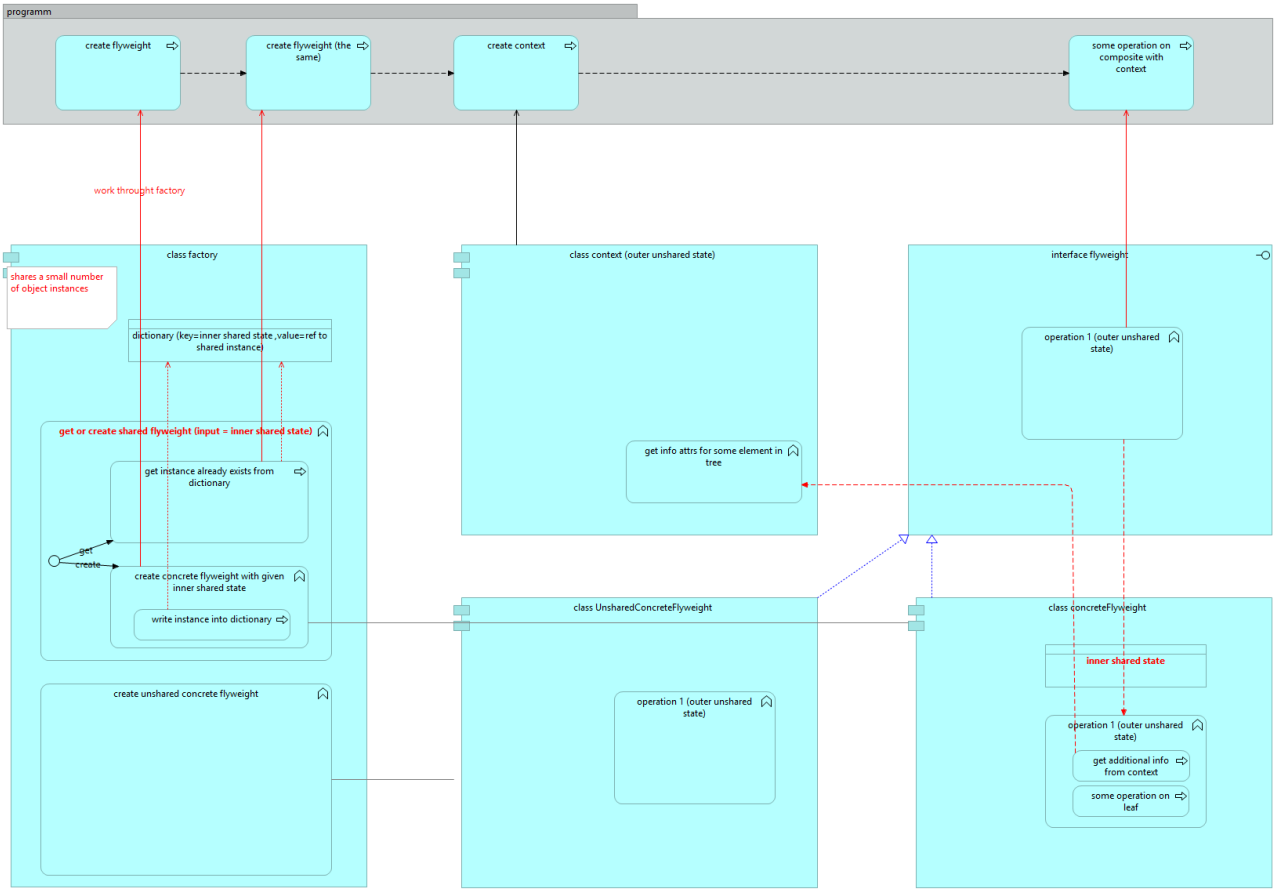
FACADE

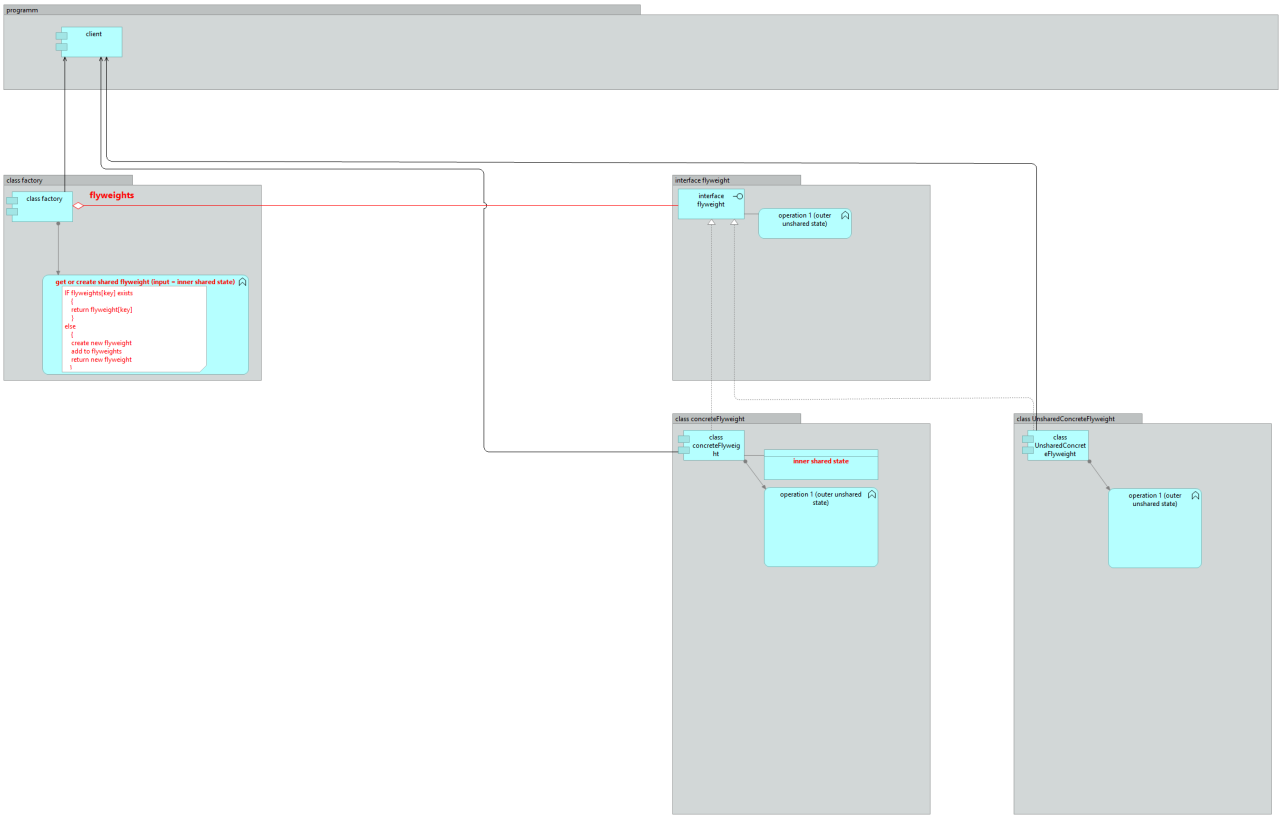

programm
call facade methods
access between the layers only through facades
layer 1 (subsystem/module/package)
facade 1
use some methods of classes
class A
class B
class C
facade created absolutly new interface
- only one point (class/interface)
- simplified (short)
- default
- common
- for most frequent cases
if I want use all possibilities, I use directly all classes
one-way calls. facade sends requests subsystem classes
access between the layers only through facades
layer 2 (subsystem/module/package)
facade 2
You can change the inner layer classes without fear affects other layers
use some methods of classes
class D
class E
class F
antipattern: if the facade is a class (not an interface) then all clients will transitively depend on all classes behind the facade
access between the layers only through facades
layer 3 (subsystem/module/package)
facade 3
use some methods of classes
class G
class H
class J


class A
access between the layers only through facades
module/package
facade 2

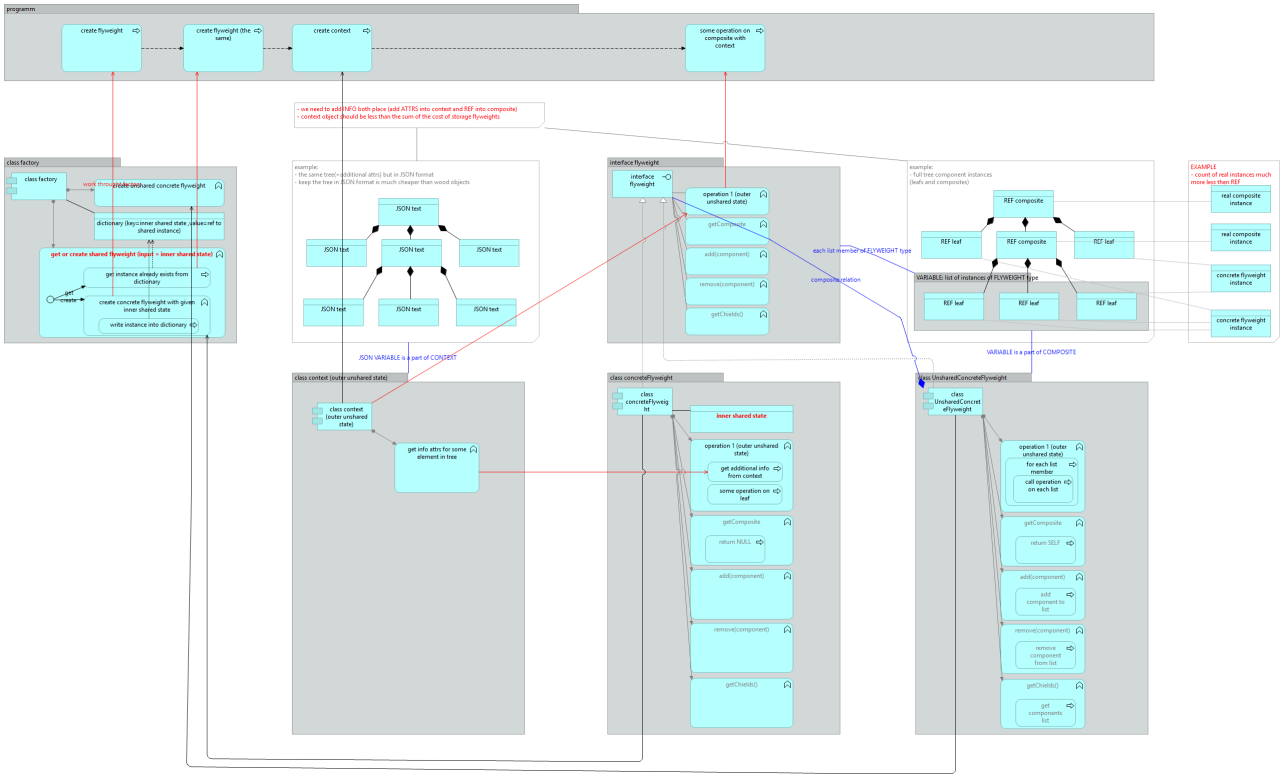

facade class
the facade for convenience combines / exposes the diverse functionality that other classes provide
func1
func2
func3
ref to class1
ref to class2
ref to class3
each class has its own responsibility
class1
func1
class2
func2
class3
func3

FLYWEIGHT

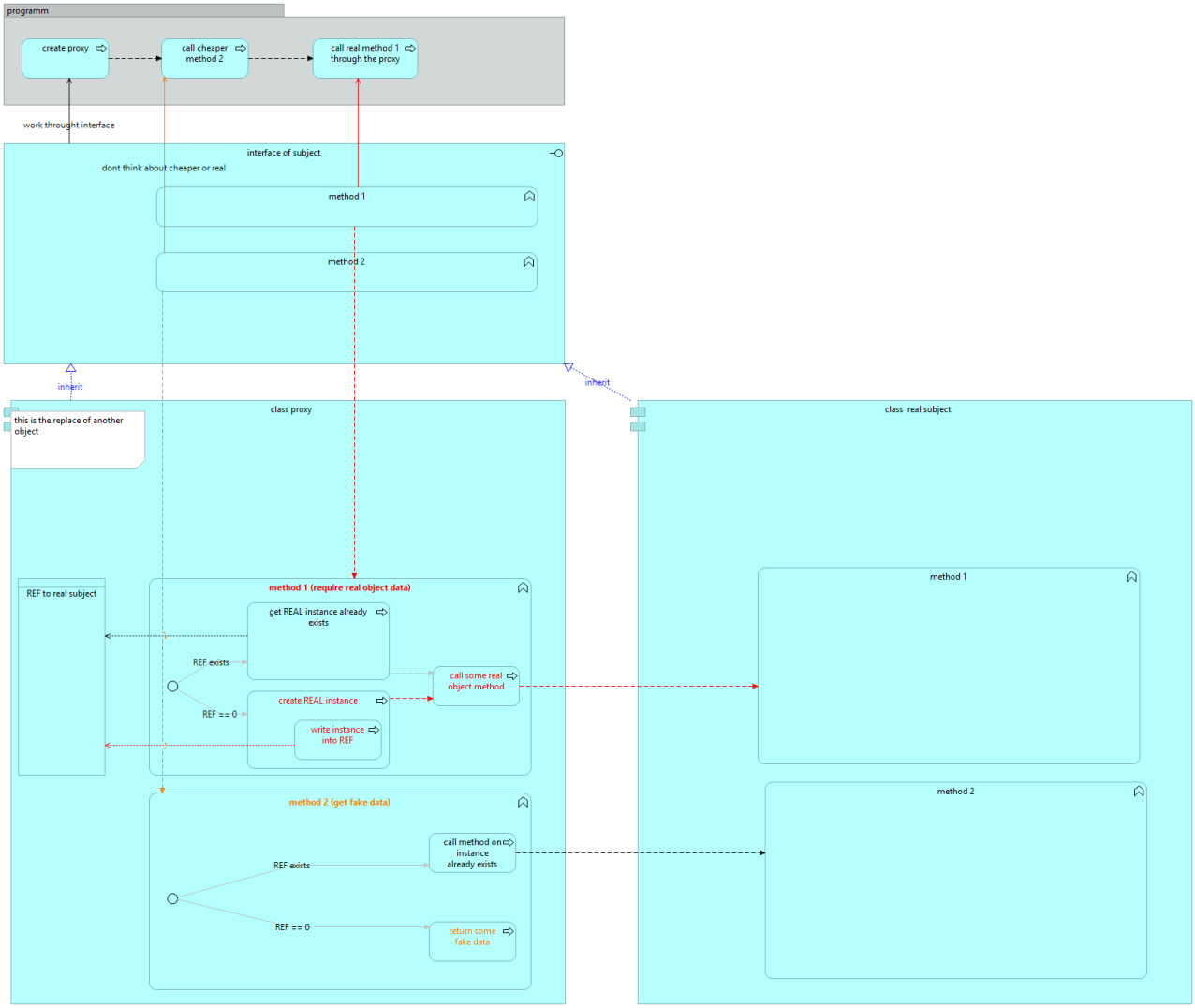

programm
create flyweight
create flyweight (the same)
create context
some operation on composite with context
work throught factory
class factory
shares a small number of object instances
dictionary (key=inner shared state ,value=ref to shared instance)
get or create shared flyweight (input = inner shared state)
get instance already exists from dictionary
get
create
create concrete flyweight with given inner shared state
write instance into dictionary
create unshared concrete flyweight
class context (outer unshared state)
get info attrs for some element in tree
interface flyweight
operation 1 (outer unshared state)
class UnsharedConcreteFlyweight
operation 1 (outer unshared state)
class concreteFlyweight
inner shared state
operation 1 (outer unshared state)
get additional info from context
some operation on leaf

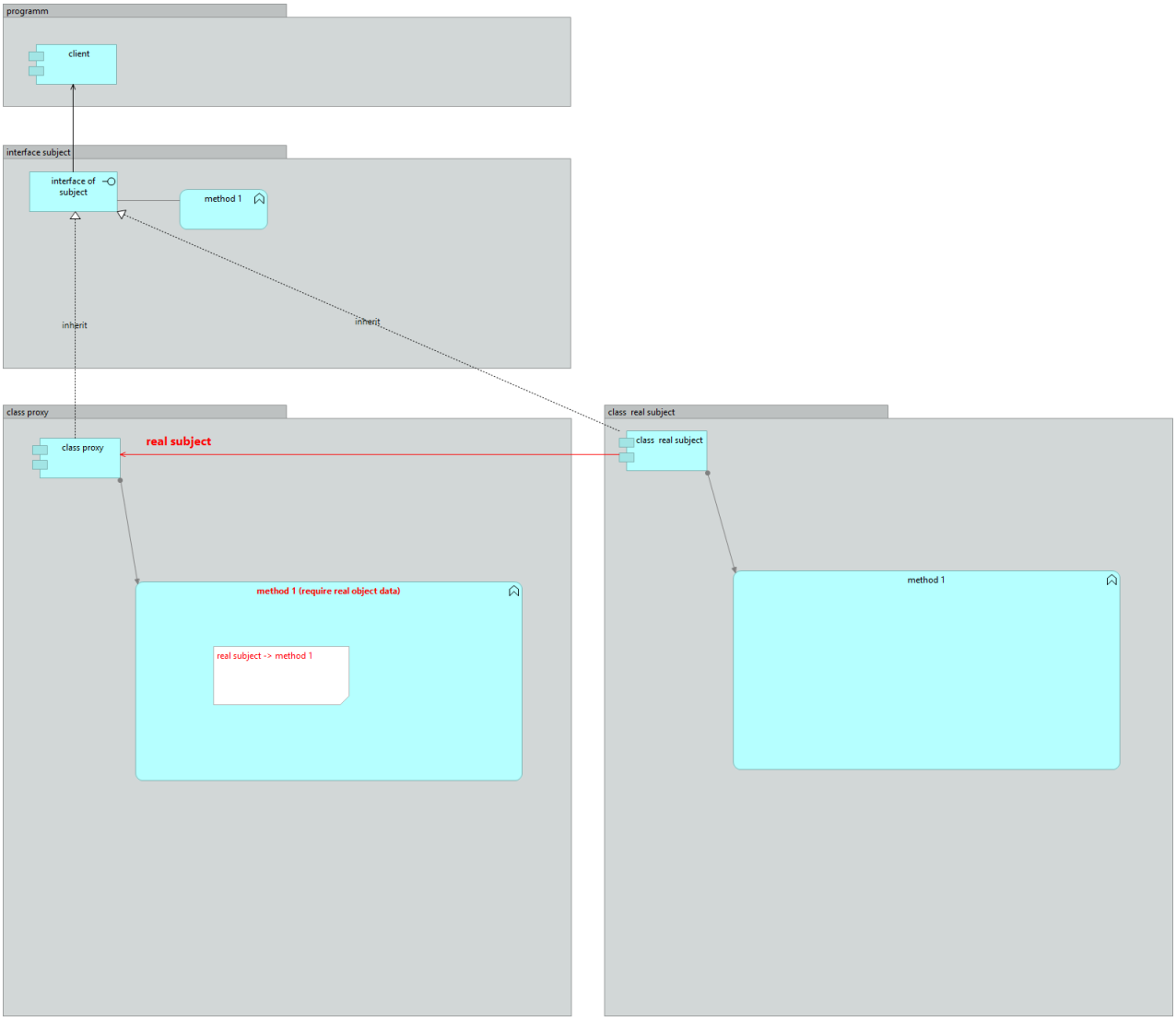

programm
client
class factory
class factory
flyweights
get or create shared flyweight (Input = inner shared state)
IF flyweights[key] exists
{
return flyweight[key]
}
else
{
create new flyweight
add to flyweights
return new flyweight
}
interface flyweight
interface flyweight
operation 1 (outer unshared state)
class concreteFlyweight
class concreteFlyweight
inner shared state
operation 1 (outer unshared state)
class UnsharedConcreteFlyweight
class UnsharedConcreteFlyweight
operation 1 (outer unshared state)

FLYWEIGHT + COMPOSITE



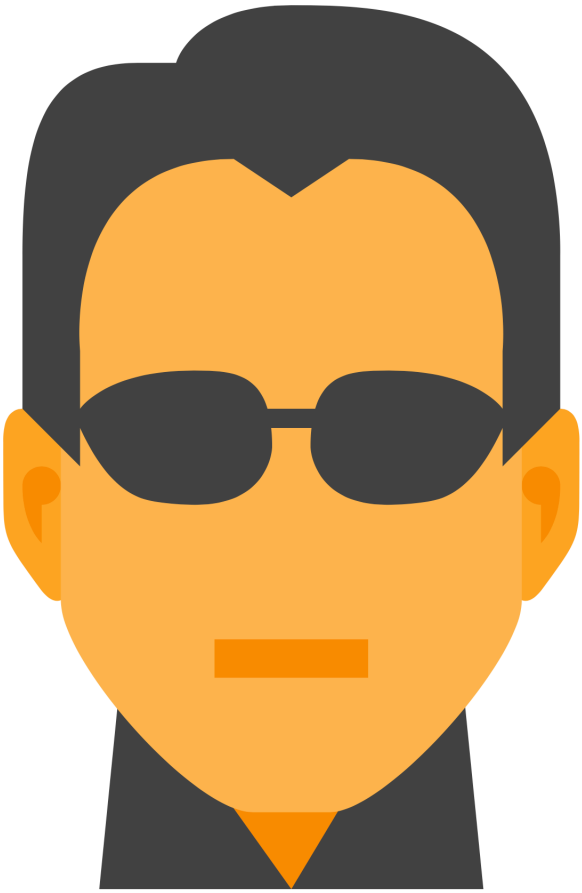
PROXY


programm
create proxy
call cheaper method 2
call real method 1 through the proxy
work throught interface
interface of subject
dont think about cheaper or real
method 1
method 2
inherit
inherit
class proxy
this is the replace of another object
REF to real subject
method 1 (require real object data)
get REAL instance already exists
REF exists
call some real object method
create REAL instance
REF == 0
write instance into REF
method 2 (get fake data)
call method on instance already exists
REF exists
REF == 0
return some fake data
class real subject
method 1
method 2

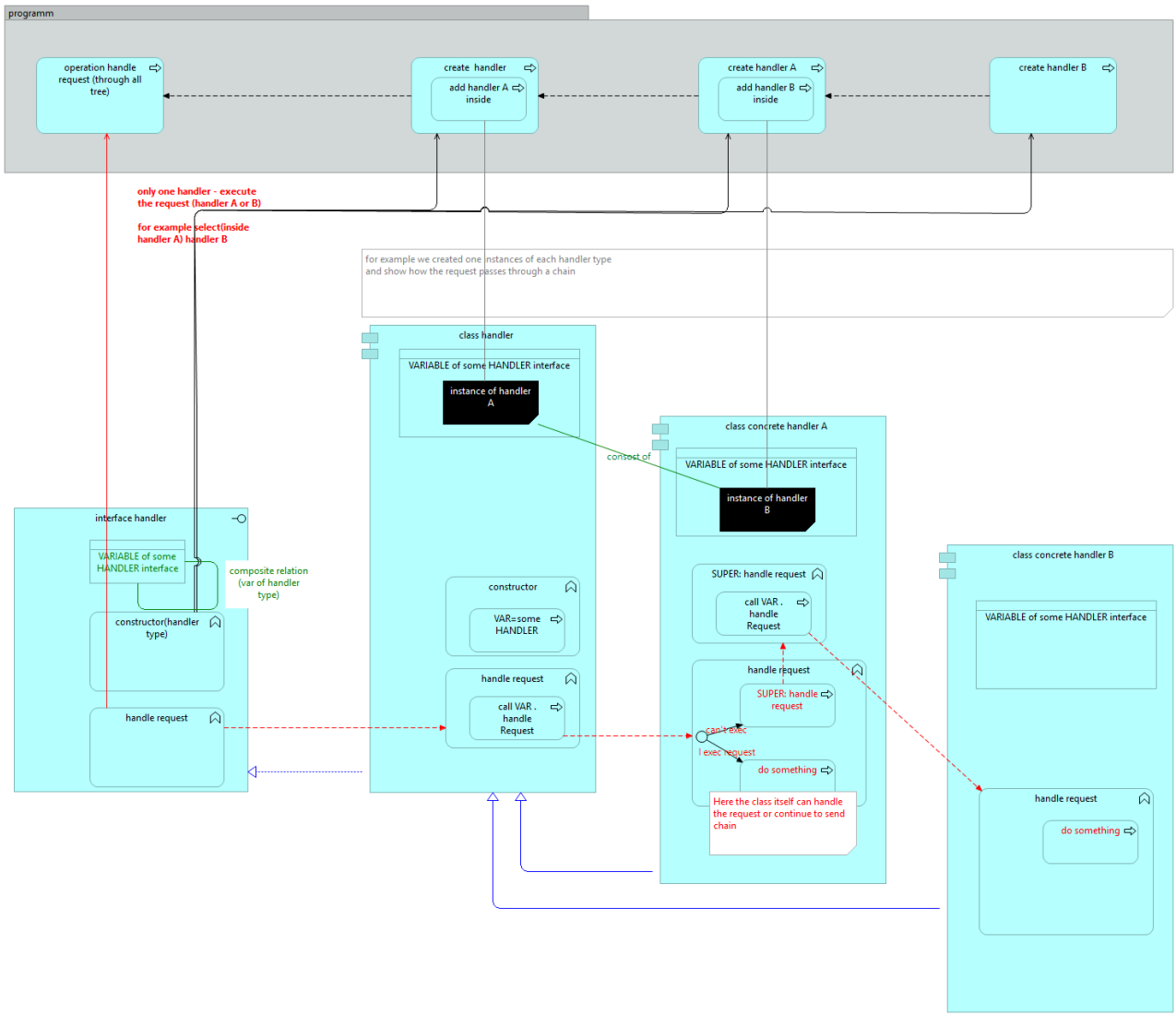

programm
client
interface subject
interface of subject
method 1
inherit
inherit
class proxy
class proxy
real subject
method 1 (require real object data)
real subject -> method 1
class real subject
class real subject
method 1

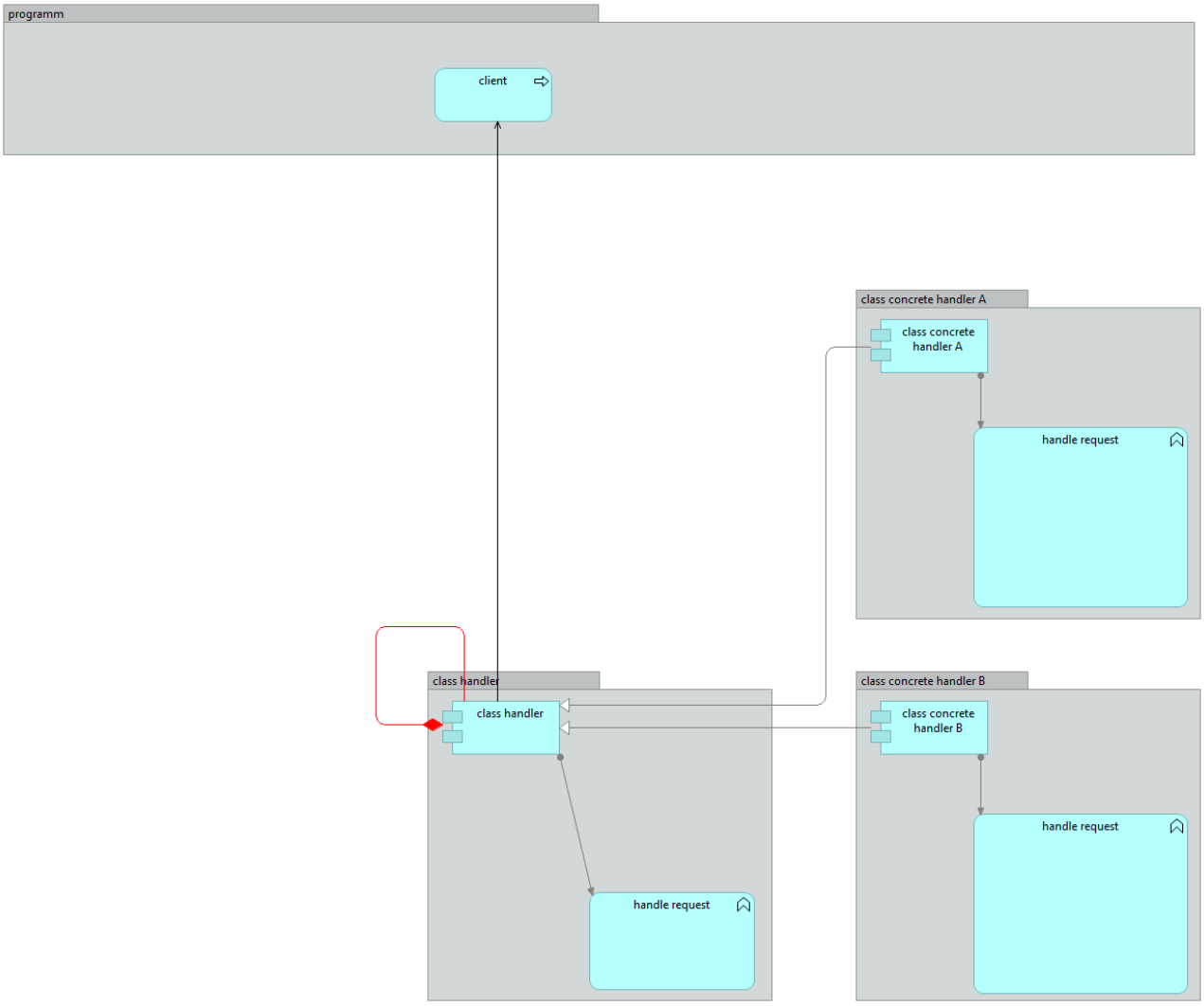
BEHAVIORAL PATTERNS

DESIGN PATTERNS | GoF

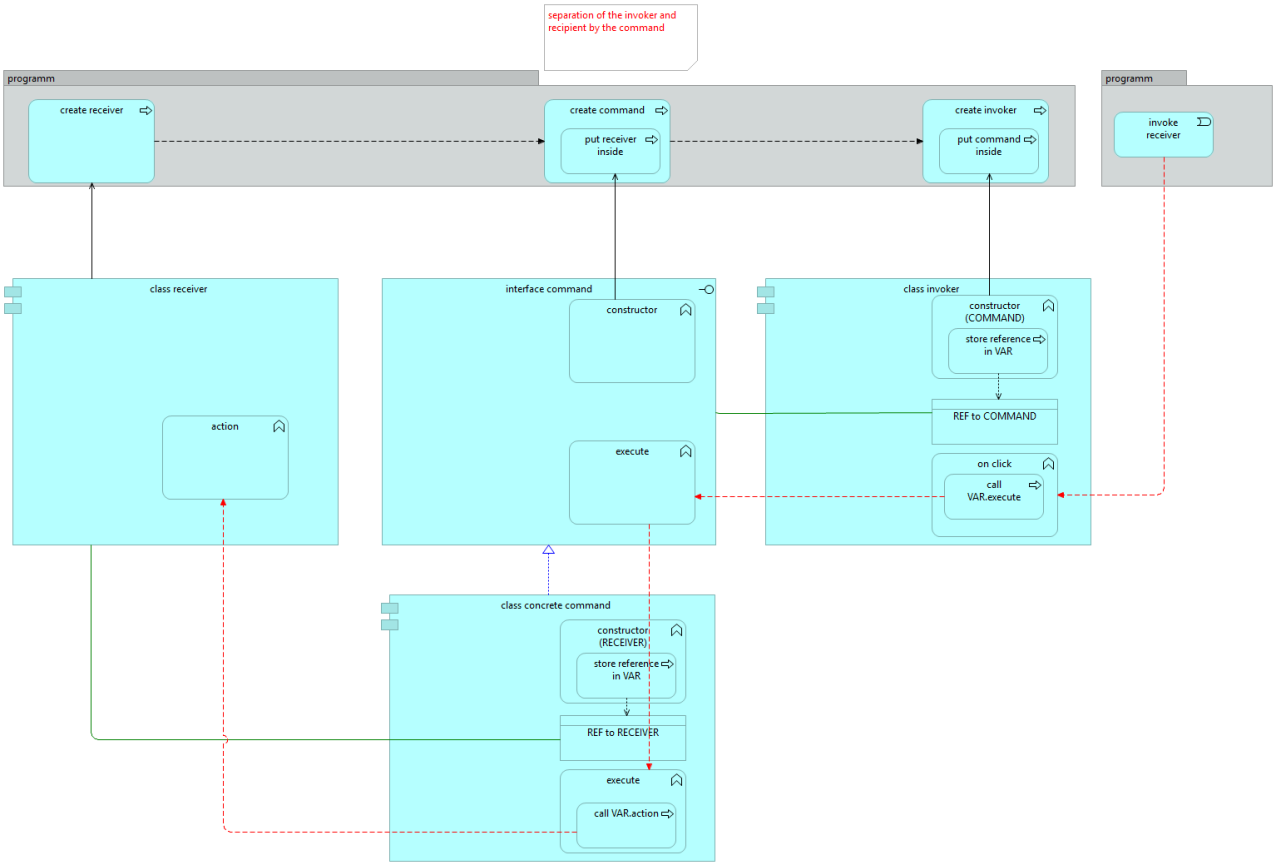


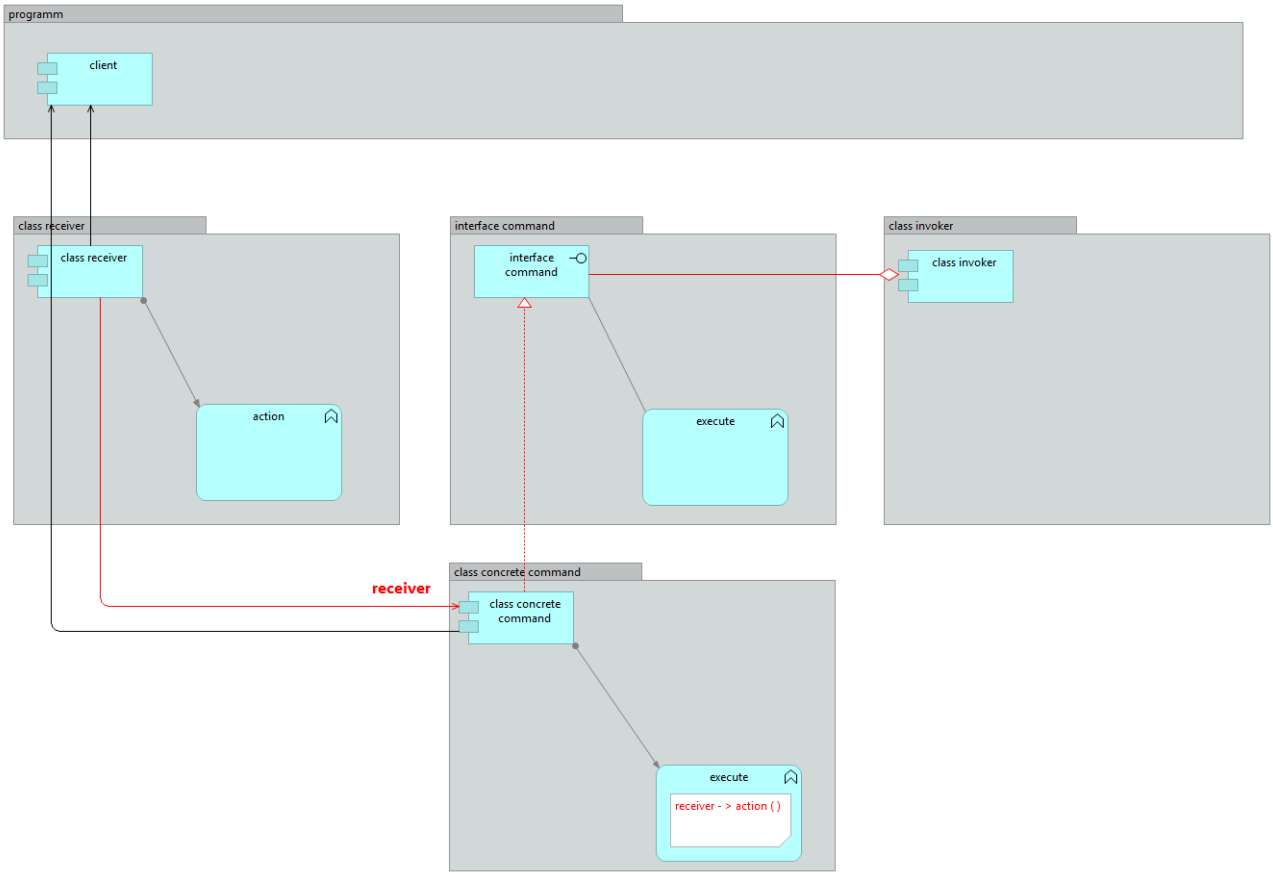
CHAIN OF RESPONSIBILITY



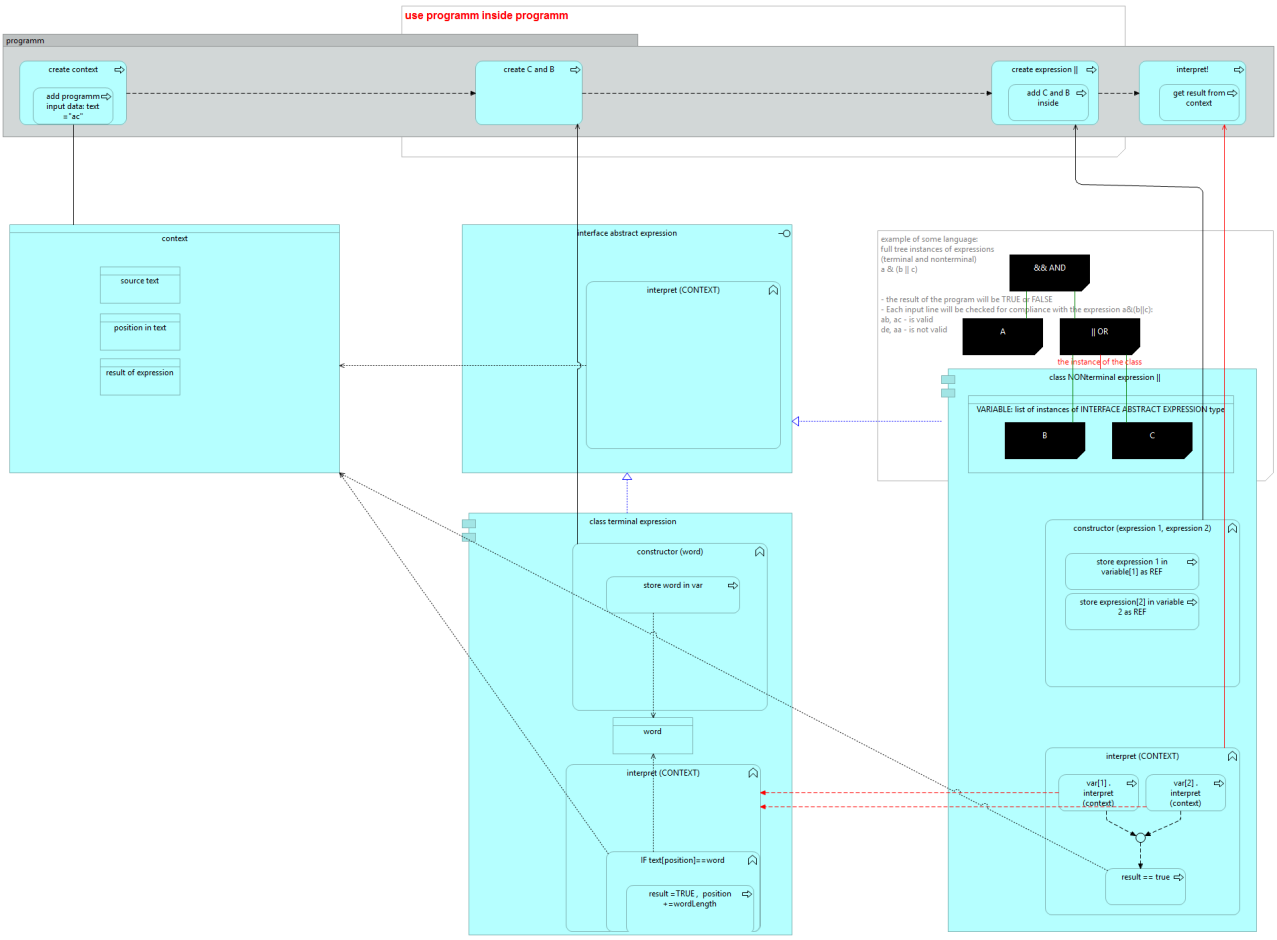

programm
client
class concrete handler A
class concrete handler A
handle request
class handler
class handler
handle request
class concrete handler B
class concrete handler B
handle request

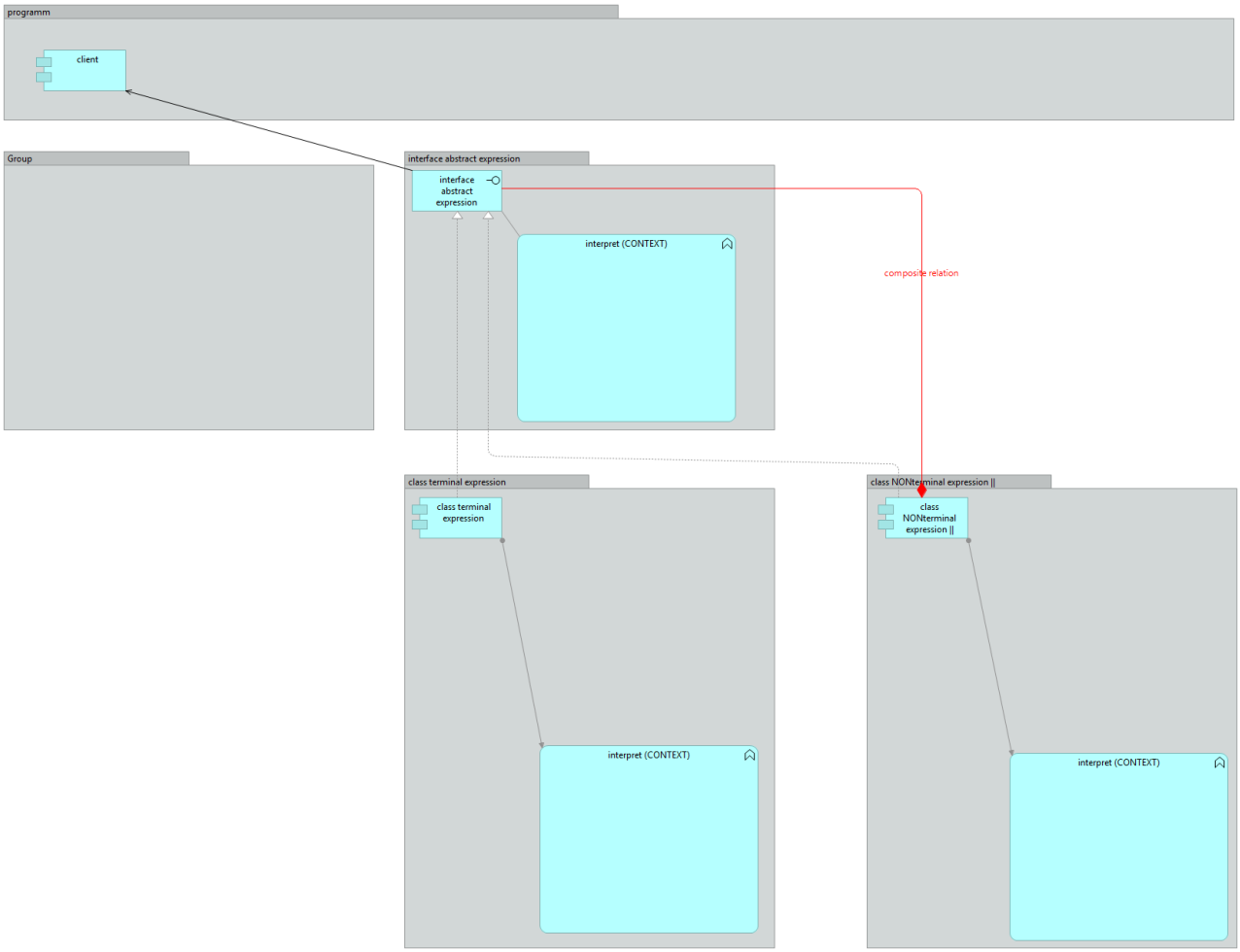
COMMAND


separation of the invoker and recipient by the command
programm
create receiver
create command
put receiver inside
create invoker
put command inside
programm
invoke receiver
class receiver
action
interface command
constructor
execute
class invoker
constructor (COMMAND)
store reference in VAR
REF to COMMAND
on click
call VAR.execute
class concrete command
constructor (RECEIVER)
store reference in VAR
REF to RECEIVER
execute
call VAR.action

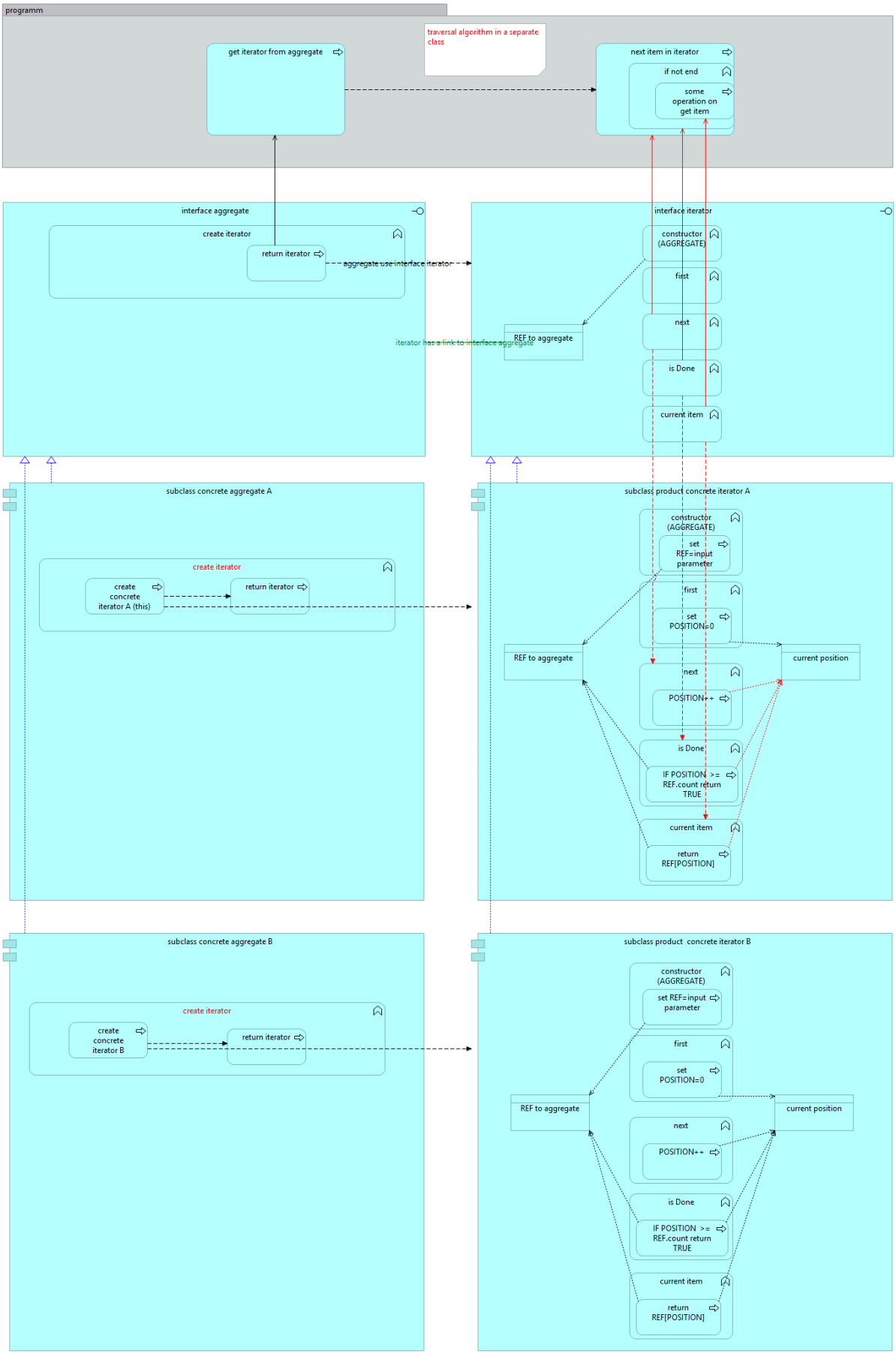

programm
client
class receiver
class receiver
action
interface command
interface command
execute
class invoker
class invoker
receiver
class concrete command
class concrete command
execute
receiver - > action ( )

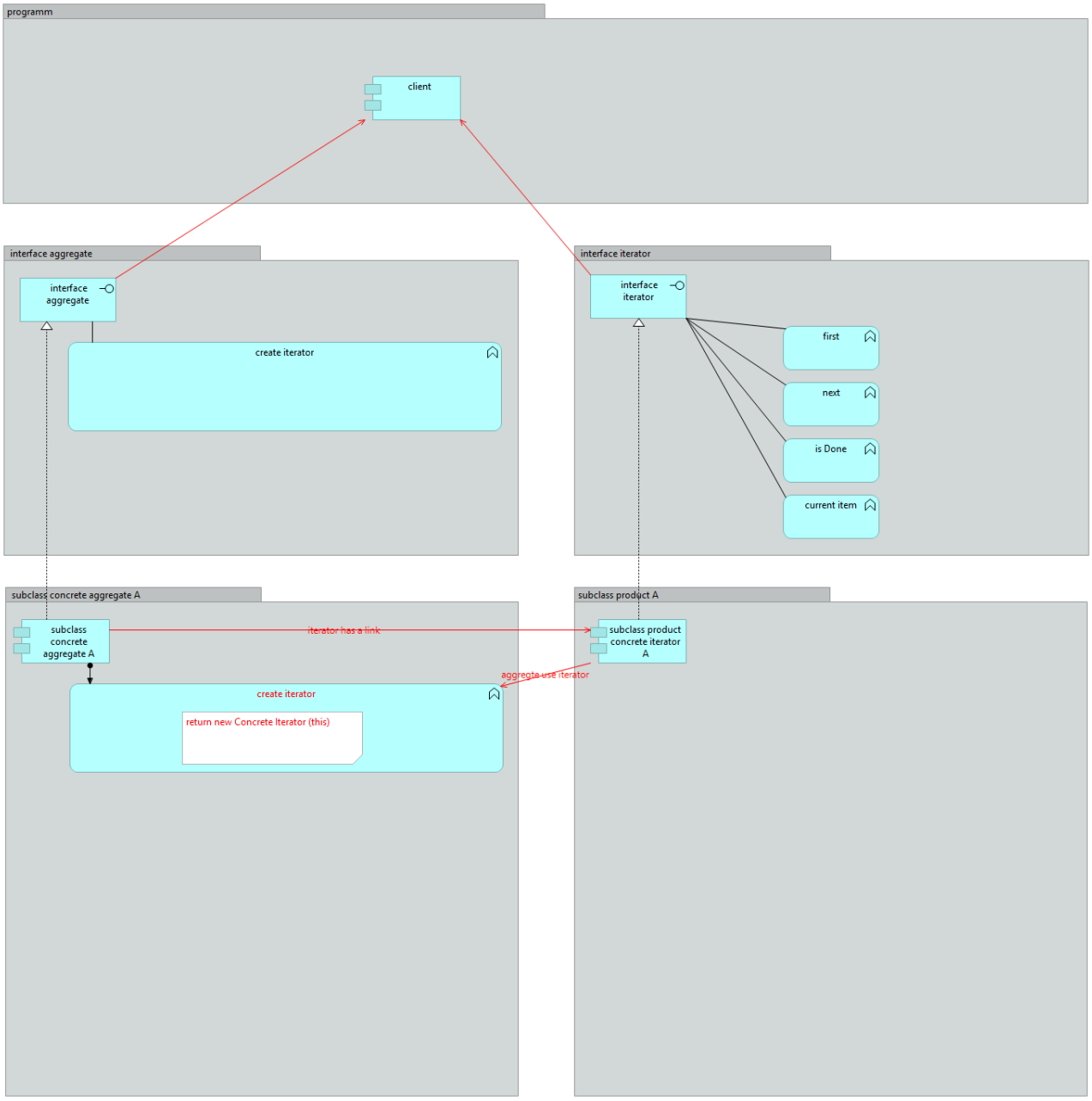
INTERPRETER



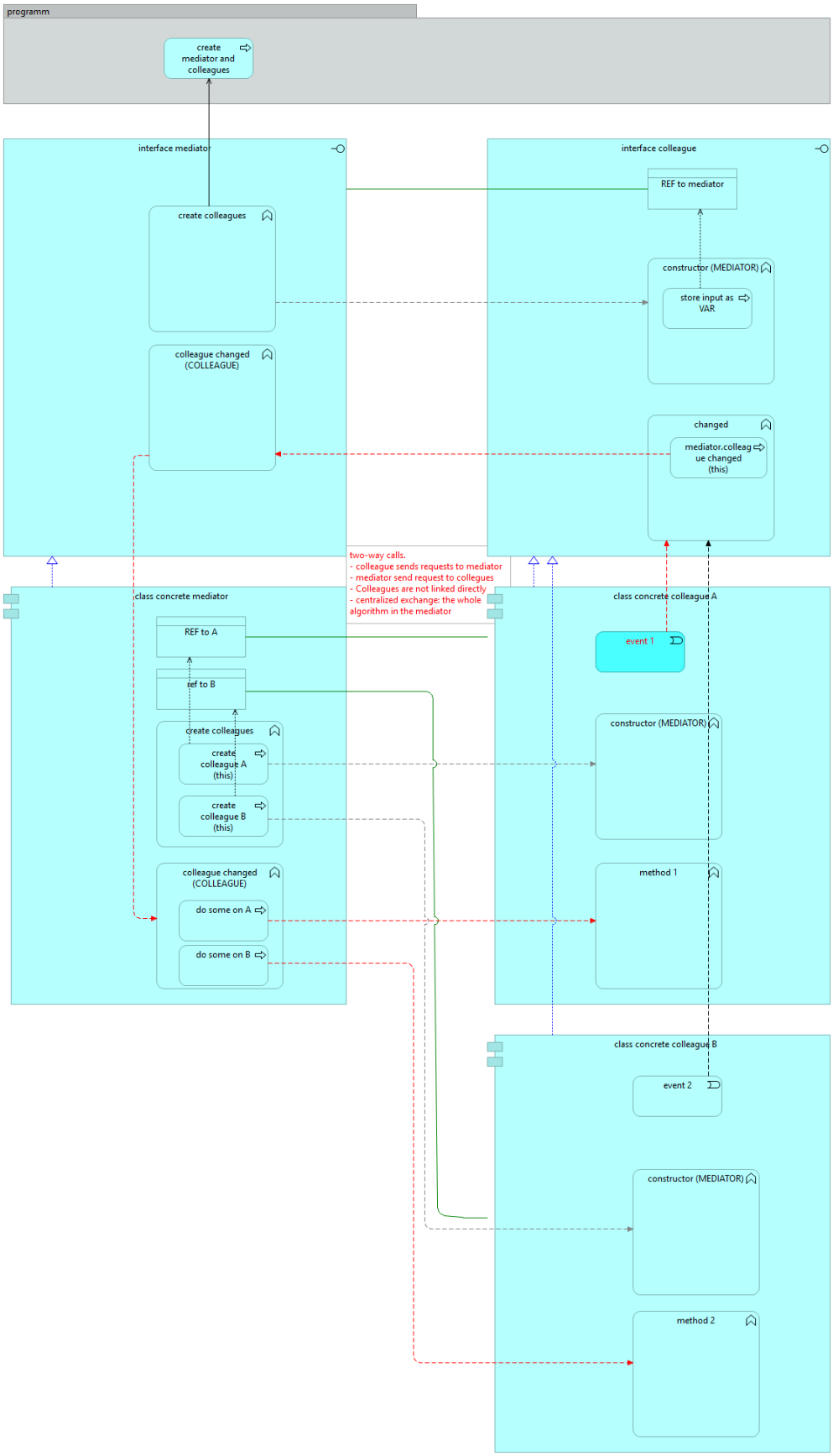

programm
client
Group
interface abstract expression
interface abstract expression
interpret (CONTEXT)
composite relation
class terminal expression
class terminal expression
interpret (CONTEXT)
class NONterminal expression ||
class NONterminal expression ||
interpret (CONTEXT)

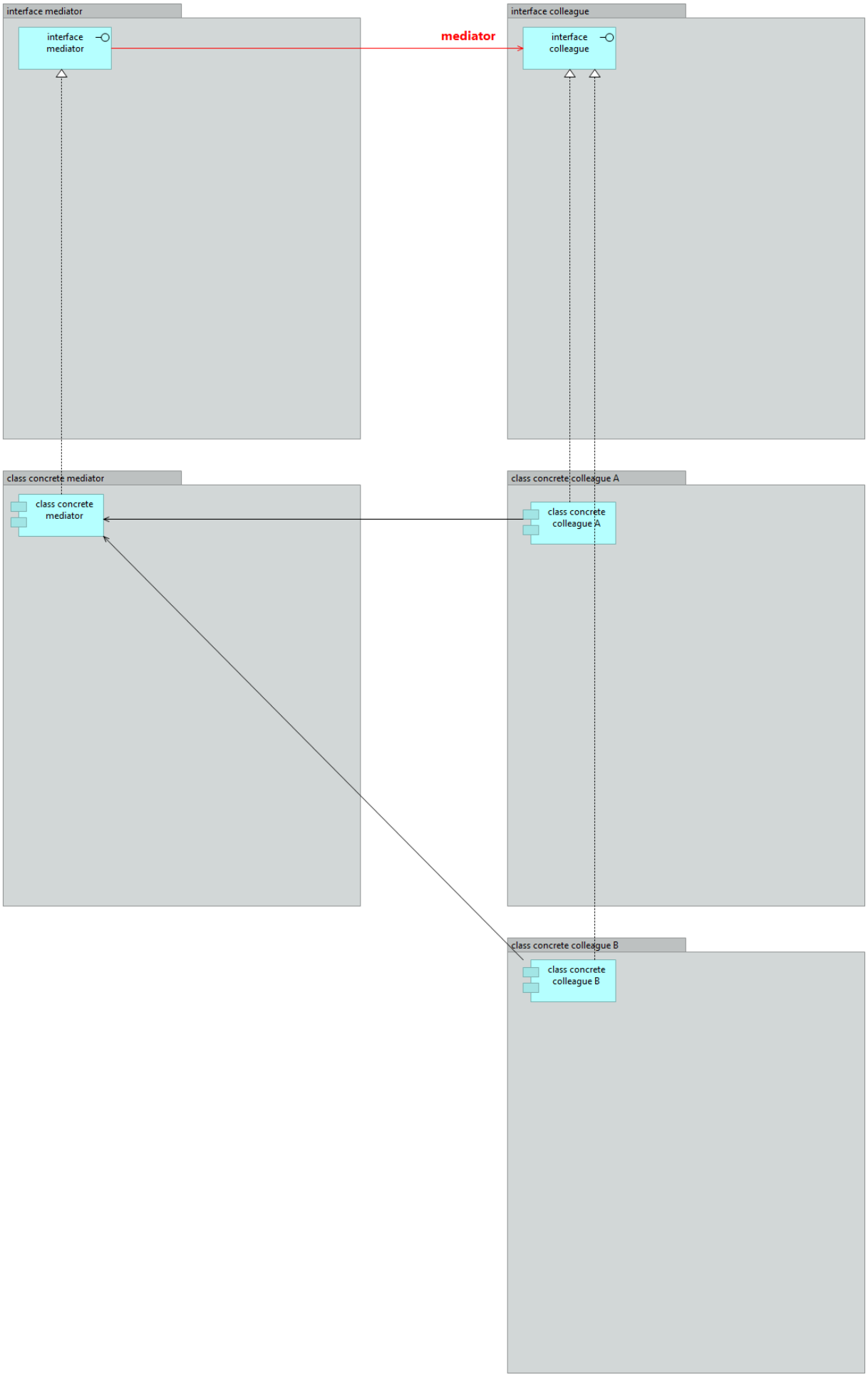
ITERATOR


programm
traversal algorithm in a separate class
get iterator from aggregate
next item in iterator
if not end
some operation on get item
interface aggregate
create iterator
return iterator
aggregate use interface iterator
interface iterator
constructor (AGGREGATE)
first
next
is Done
current item
REF to aggregate
iterator has a link to interface aggregate
subclass concrete aggregate A
create iterator
create concrete iterator A (this)
return iterator
subclass product concrete iterator A
constructor (AGGREGATE)
set REF=input parameter
first
set POSITION=0
REF to aggregate
current position
next
POSITION++
is Done
IF POSITION >= REF.count return TRUE
current item
return REF[POSITION]
subclass concrete aggregate B
create iterator
create concrete iterator B
return iterator
subclass product concrete iterator B
constructor (AGGREGATE)
set REF=input parameter
first
set POSITION=0
REF to aggregate
current position
next
POSITION++
is Done
IF POSITION >= REF.count return TRUE
current item
return REF[POSITION]

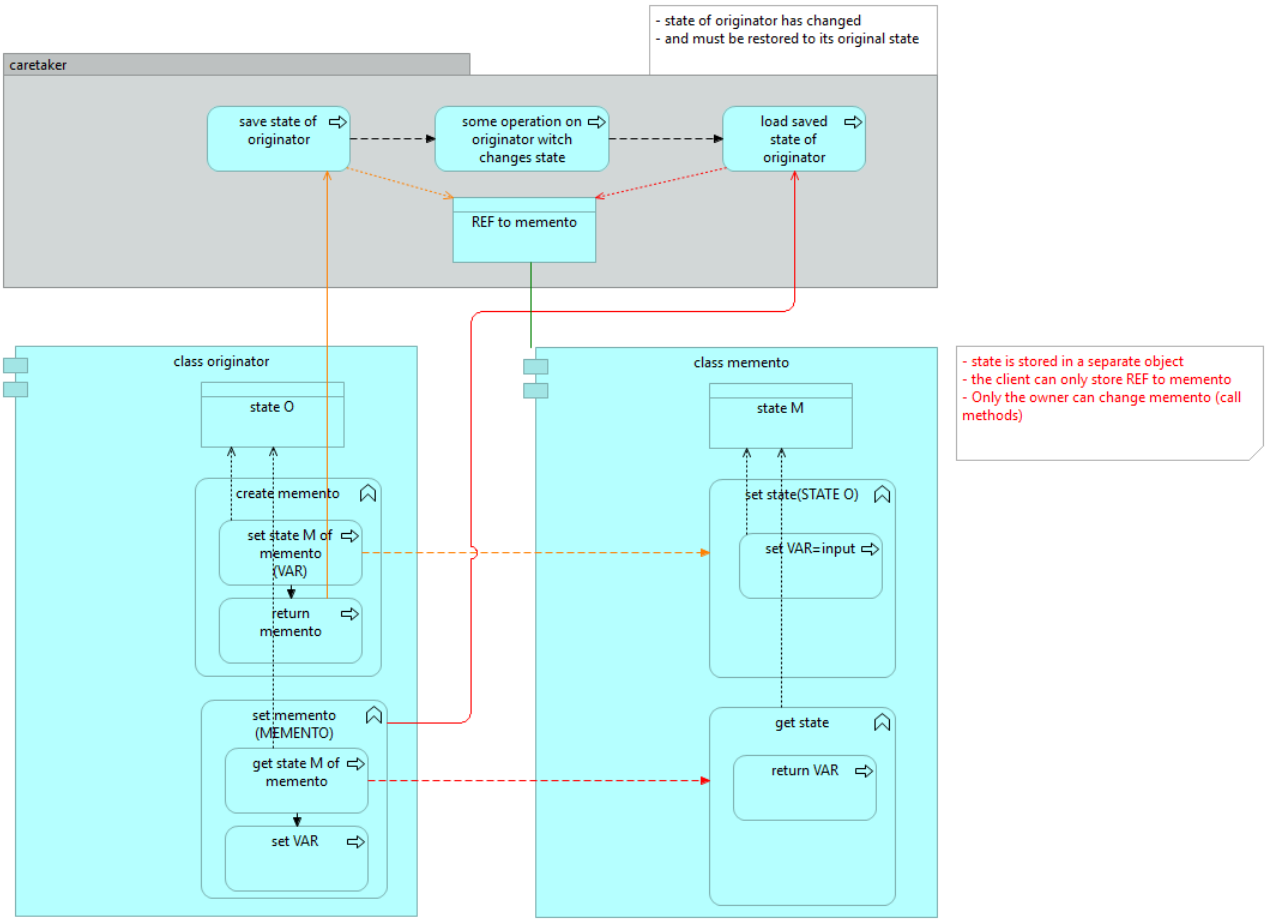

programm
client
interface aggregate
interface aggregate
create iterator
interface iterator
interface iterator
first
next
is Done
current item
subclass concrete aggregate A
subclass concrete aggregate A
iterator has a link
create iterator
return new Concrete Iterator (this)
subclass product A
subclass product concrete iterator A
aggregta use iterator

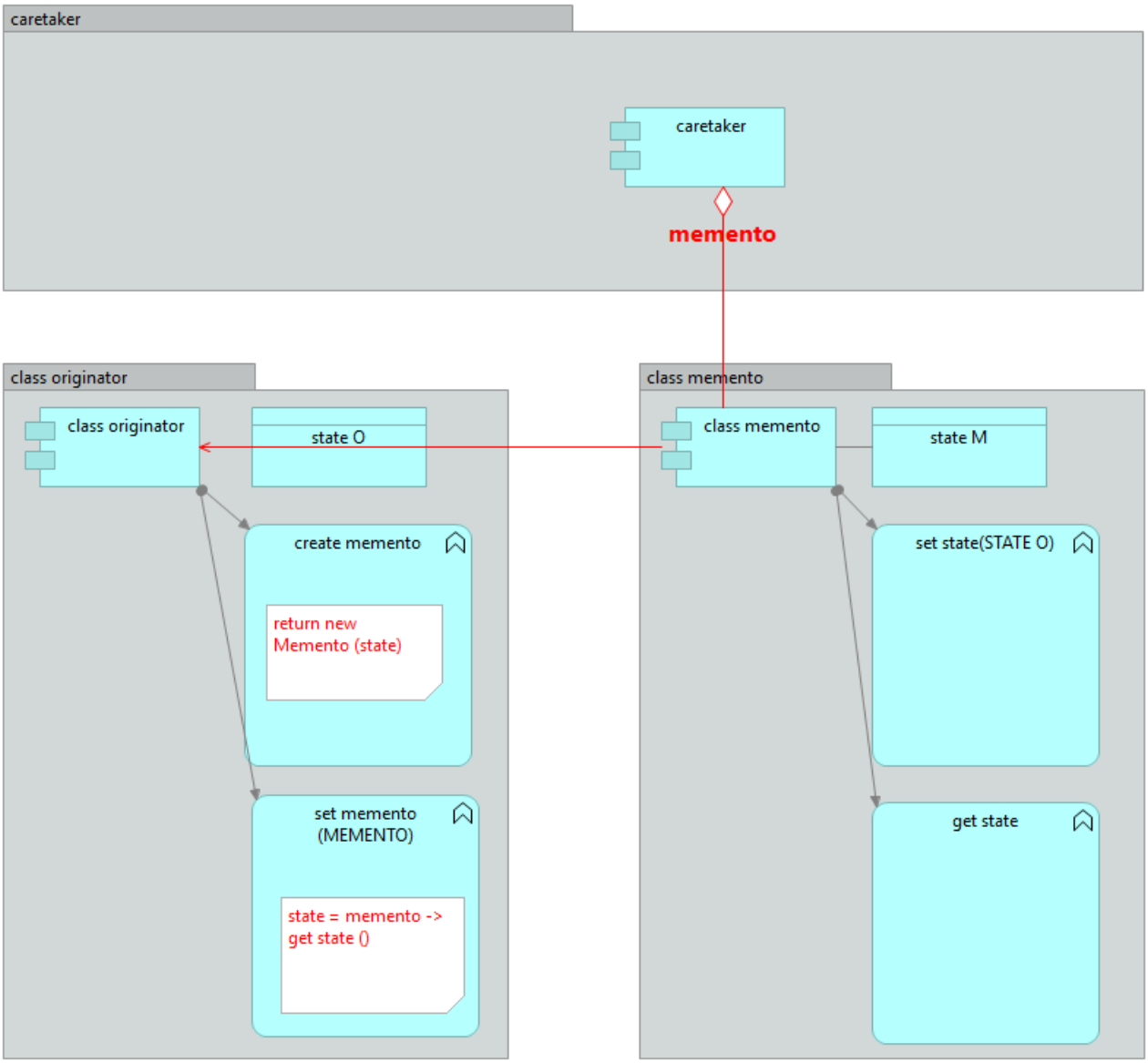
MEDIATOR


programm
create mediator and colleagues
interface mediator
create colleagues
colleague changed (COLLEAGUE)
interface colleague
REF to mediator
constructor (MEDIATOR)
store input as VAR
changed
mediator.colleague changed (this)
two-way calls.
- colleague sends requests to mediator
- mediator send request to collegues
- Colleagues are not linked directly
- centralized exchange: the whole algorithm in the mediator
class concrete mediator
REF to A
ref to B
create colleagues
create colleague A (this)
create colleague B (this)
colleague changed (COLLEAGUE)
do some on A
do some on B
class concrete colleague A
event 1
constructor (MEDIATOR)
method 1
class concrete colleague B
event 2
constructor (MEDIATOR)
method 2

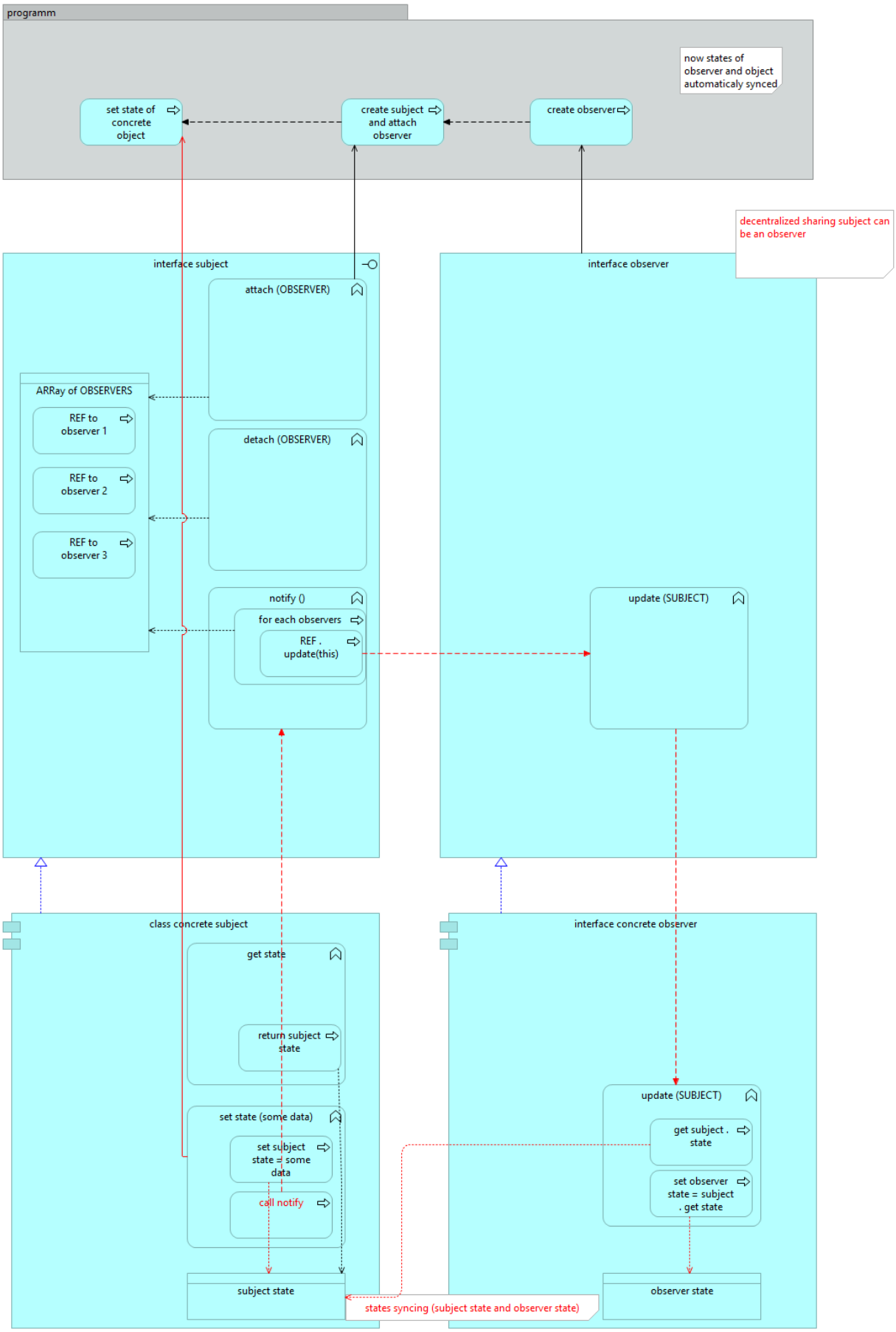

interface mediator
interface mediator
mediator
interface colleague
interface colleague
class concrete mediator
class concrete mediator
class concrete colleague A
class concrete colleague A
class concrete colleague B
class concrete colleague B

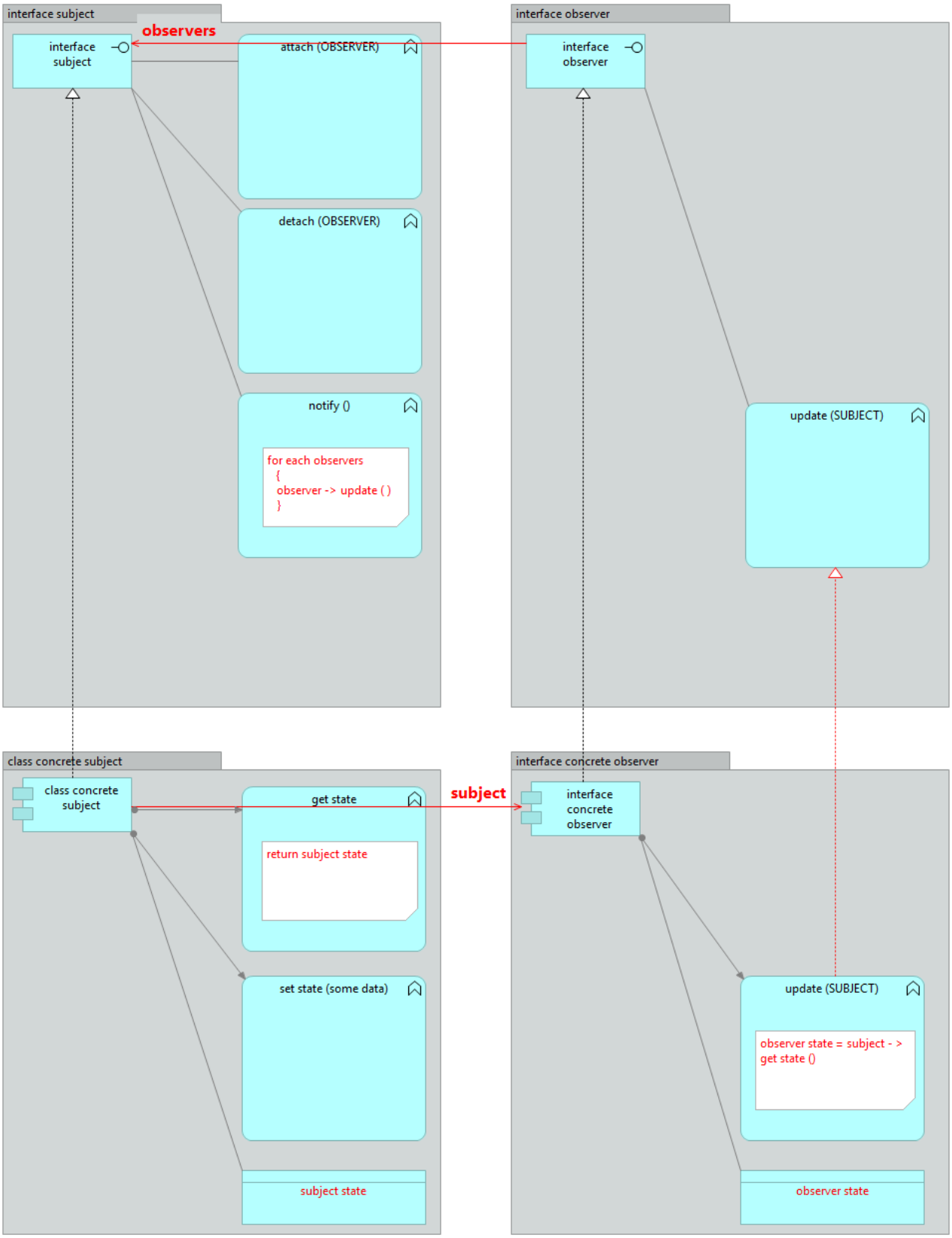
MEMENTO


- state of originator has changed
- and must be restored to its original state
caretaker
save state of originator
some operation on originator witch changes state
load saved state of originator
REF to memento
class originator
state O
create memento
set state M of memento (VAR)
return memento
set memento (MEMENTO)
get state M of memento
set VAR
class memento
state M
set state(STATE O)
set VAR=input
get state
return VAR
- state is stored in a separate object
- the client can only store REF to memento
- Only the owner can change memento (call methods)

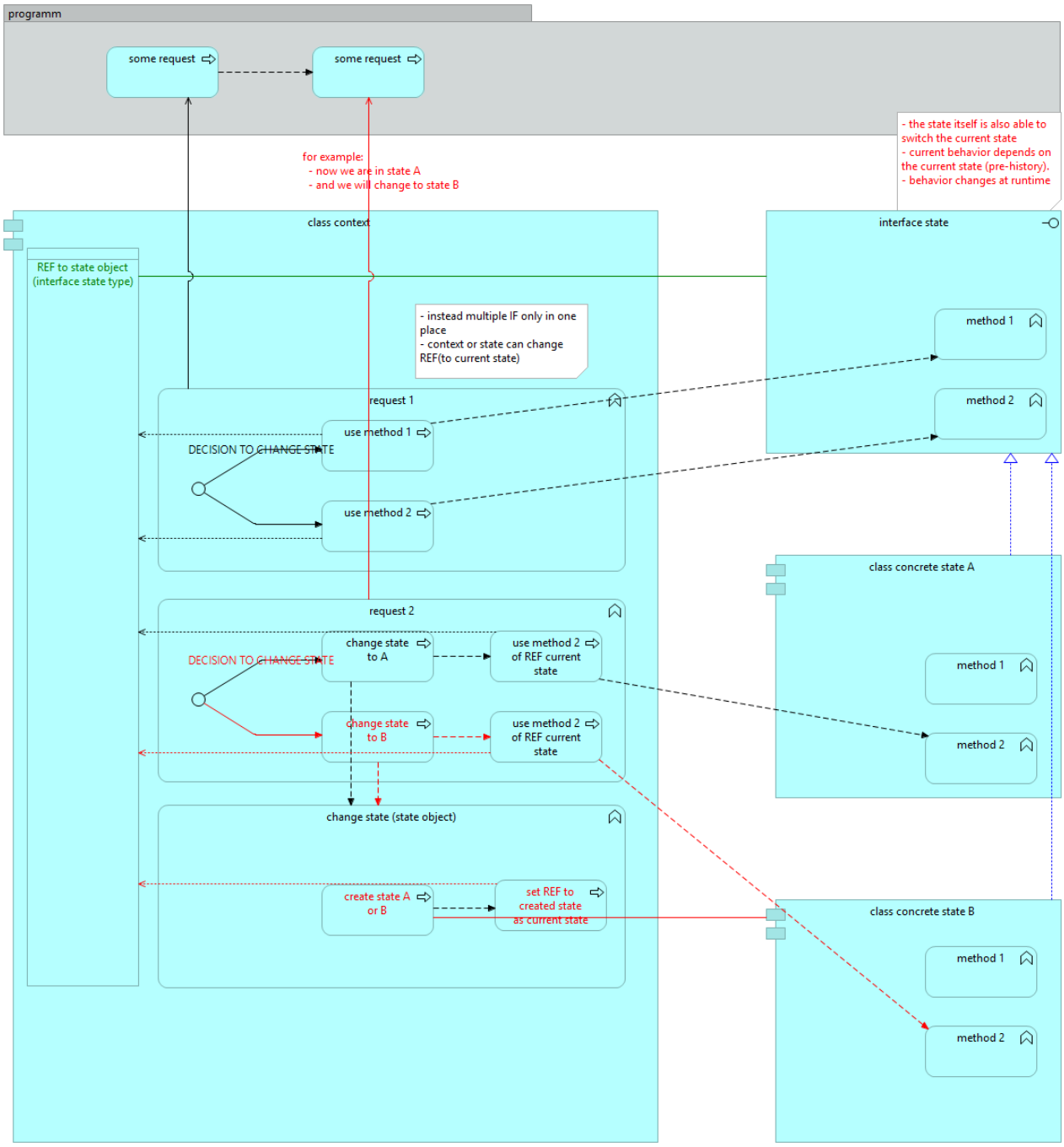

caretaker
caretaker
memento
class originator
class originator
state O
create memento
return new
Memento (state)
set memento
(MEMENTO)
state = memento ->
get state ()
class memento
class memento
state M
set state(STATE O)
get state

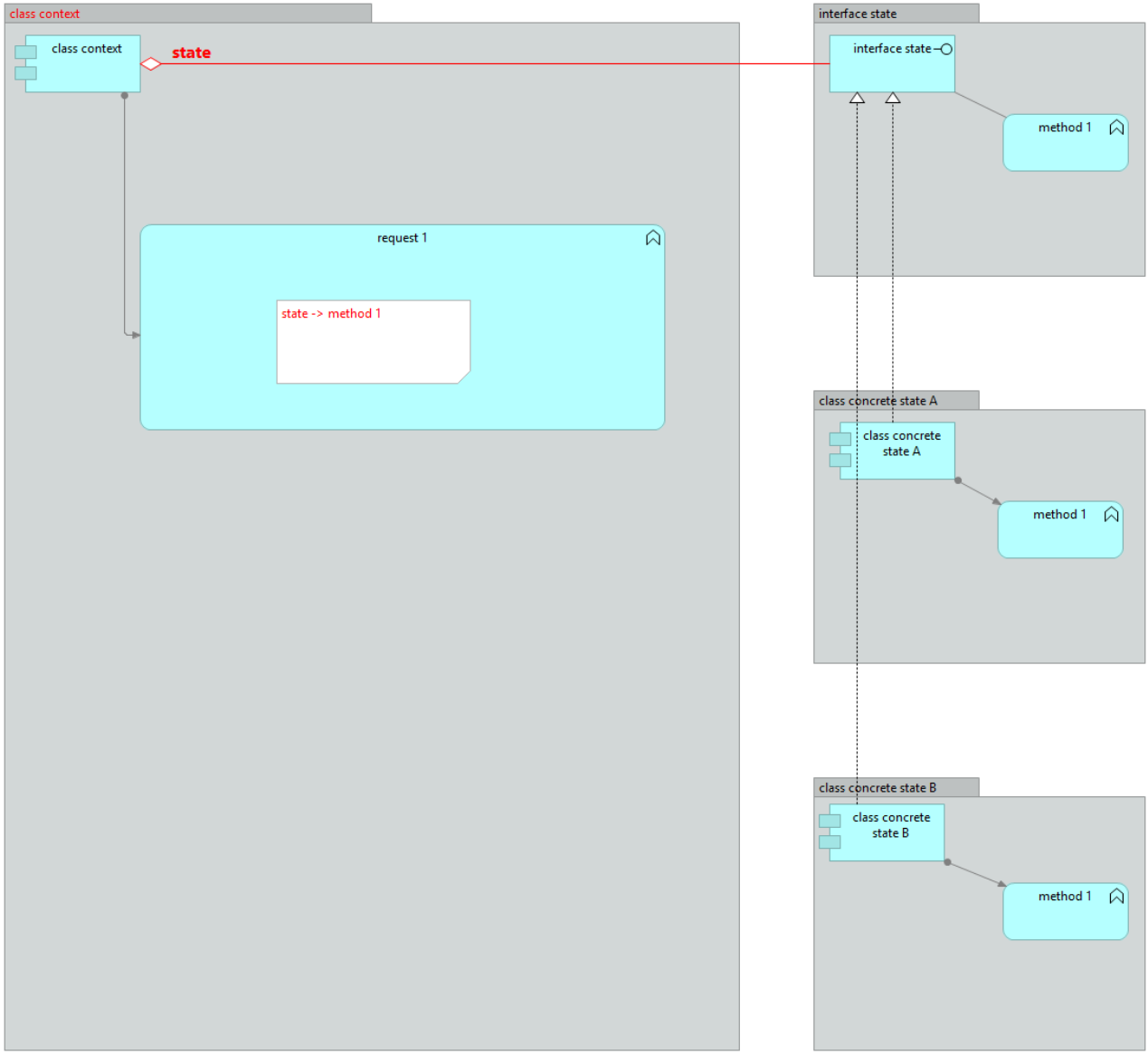
OBSERVER


programm
now states of observer and object automaticaly synced
set state of concrete object
create subject and attach observer
create observer
decentralized sharing subject can be an observer
interface subject
attach (OBSERVER)
ARRay of OBSERVERS
REF to observer 1
REF to observer 2
REF to observer 3
detach (OBSERVER)
notify ()
for each observers
REF . update(this)
interface observer
update (SUBJECT)
class concrete subject
get state
return subject state
set state (some data)
set subject state = some data
call notify
subject state
interface concrete observer
update (SUBJECT)
get subject . state
set observer state = subject . get state
observer state
states syncing (subject state and observer state)

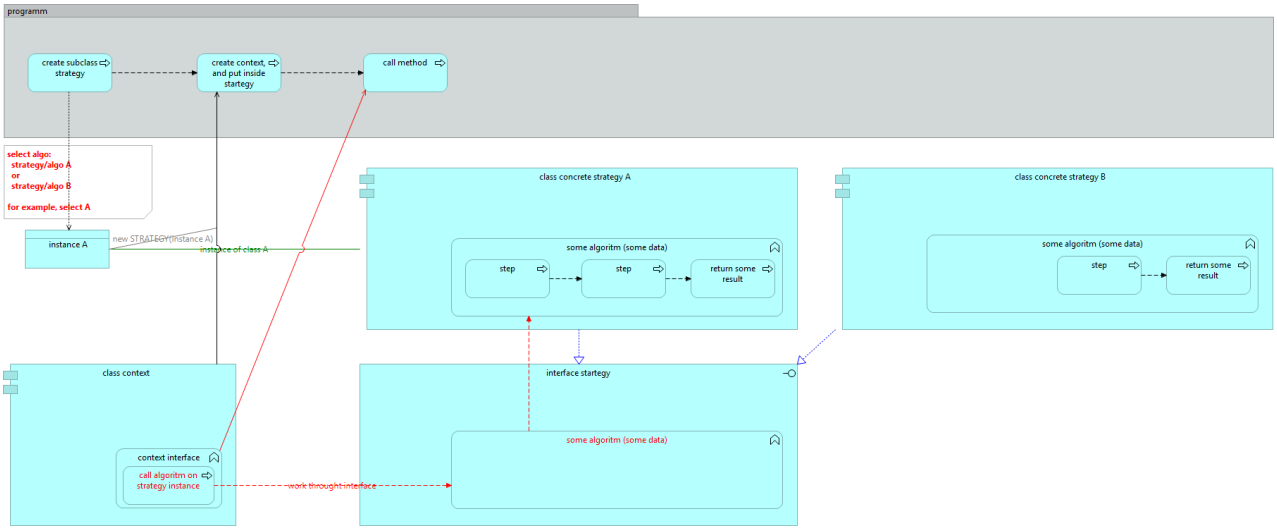

interface subject
observers
interface subject
attach (OBSERVER)
detach (OBSERVER)
notify ()
for each observers
{
observer -> update ( )
}
interface observer
interface observer
update (SUBJECT)
class concrete subject
class concrete subject
get state
return subject state
set state (some data)
subject state
subject
interface concrete observer
interface concrete observer
update (SUBJECT)
observer state = subject - > get state ()
observer state

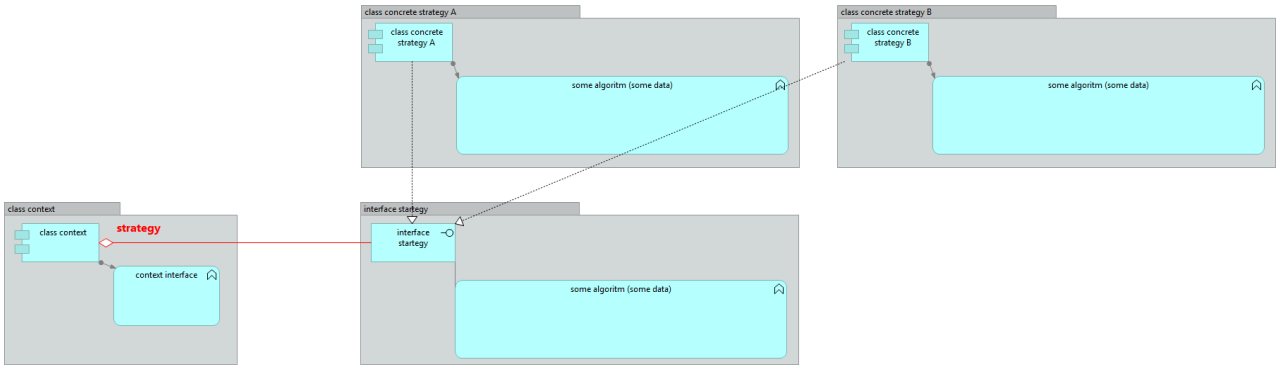
STATE


programm
some request
some request
- the state itself is also able to switch the current state
- current behavior depends on the current state (pre-history).
- behavior changes at runtime
for example:
- now we are in state A
- and we will change to state B
class context
interface state
REF to state object (interface state type)
- instead multiple IF only in one place
- context or state can change REF(to current state)
method 1
method 2
request 1
use method 1
DECISION TO CHANGE STATE
use method 2
class concrete state A
request 2
change state to A
use method 2 of REF current state
DECISION TO CHANGE STATE
change state to B
use method 2 of REF current state
method 1
method 2
change state (state object)
create state A or B
set REF to created state as current state
class concrete state B
method 1
method 2

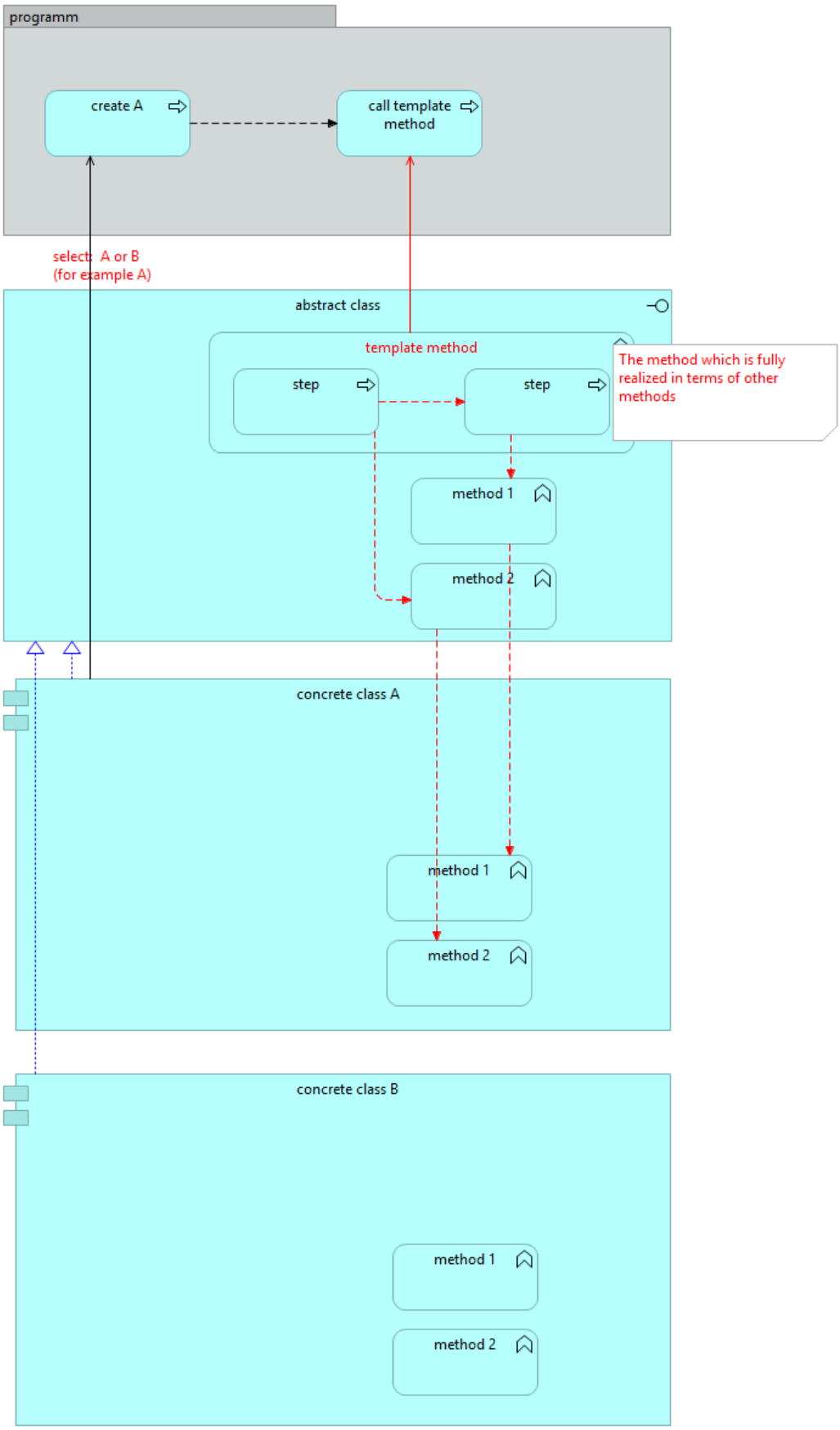

class context
class context
state
request 1
state -> method 1
interface state
interface state
method 1
class concrete state A
class concrete state A
method 1
class concrete state B
class concrete state B
method 1

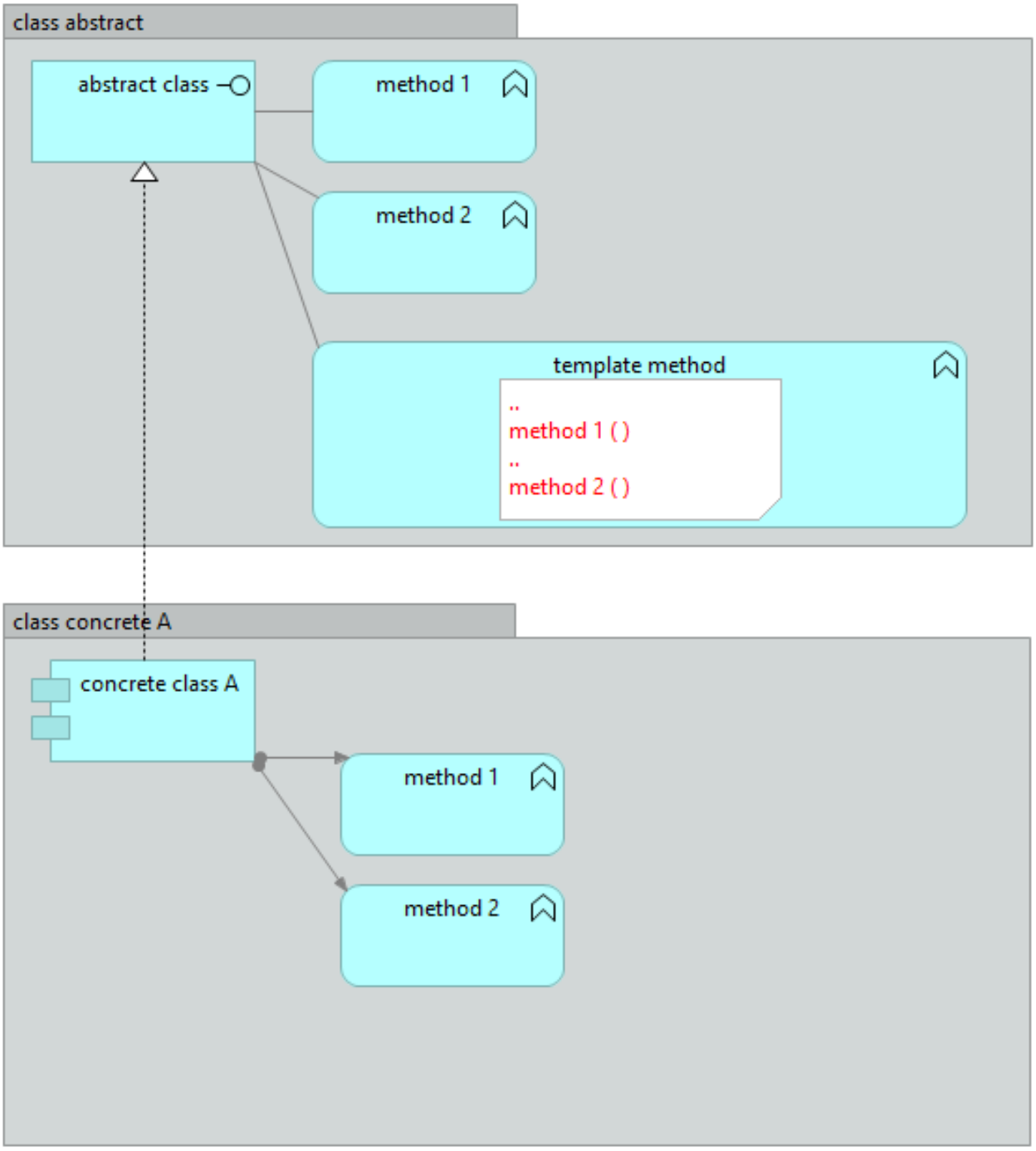
STRATEGY


programm
create subclass strategy
create context, and put inside startegy
call method
select algo:
strategy/algo A
or
strategy/algo B
for example, select A
instance A
new STRATEGY(instance A)
instance of class A
class concrete strategy A
some algoritm (some data)
step
step
return some result
class concrete strategy B
some algoritm (some data)
step
return some result
class context
context interface
call algoritm on strategy instance
work throught interface
interface startegy
some algoritm (some data)


class concrete strategy A
class concrete strategy A
some algoritm (some data)
class concrete strategy B
class concrete strategy B
some algoritm (some data)
class context
class context
strategy
context interface
interface startegy
interface startegy
some algoritm (some data)

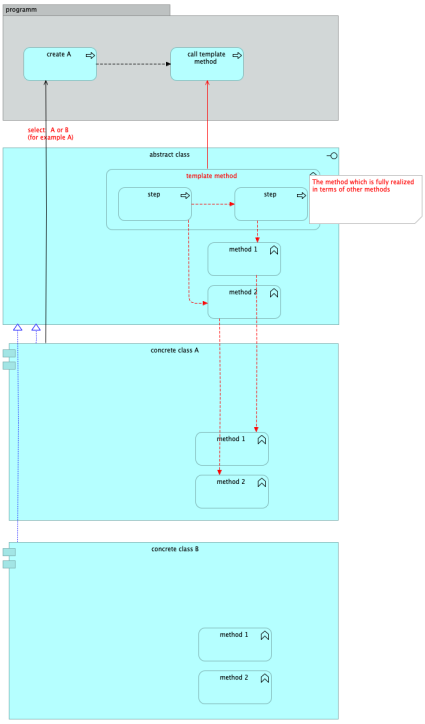
TEMPLATE METHOD



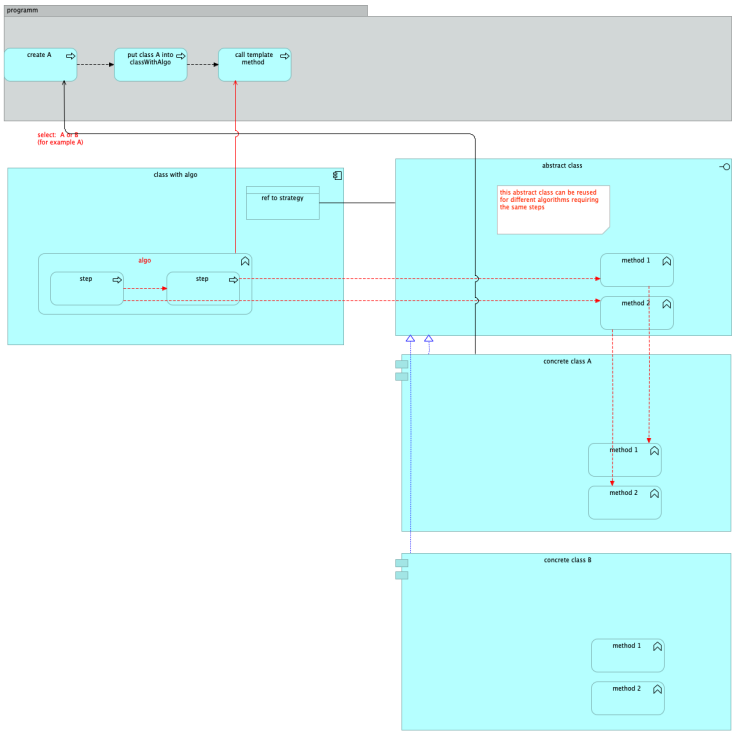

class abstract
abstract class
method 1
method 2
template method
..
method 1 ( )
..
method 2 ( )
class concrete A
concrete class A
method 1
method 2

difference between template method and strategy patterns

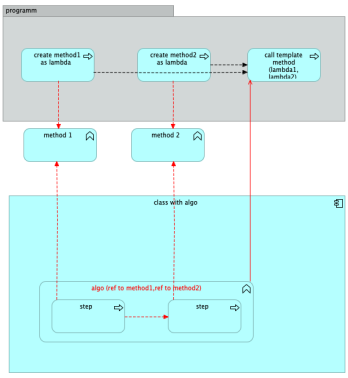
Varian1) template method



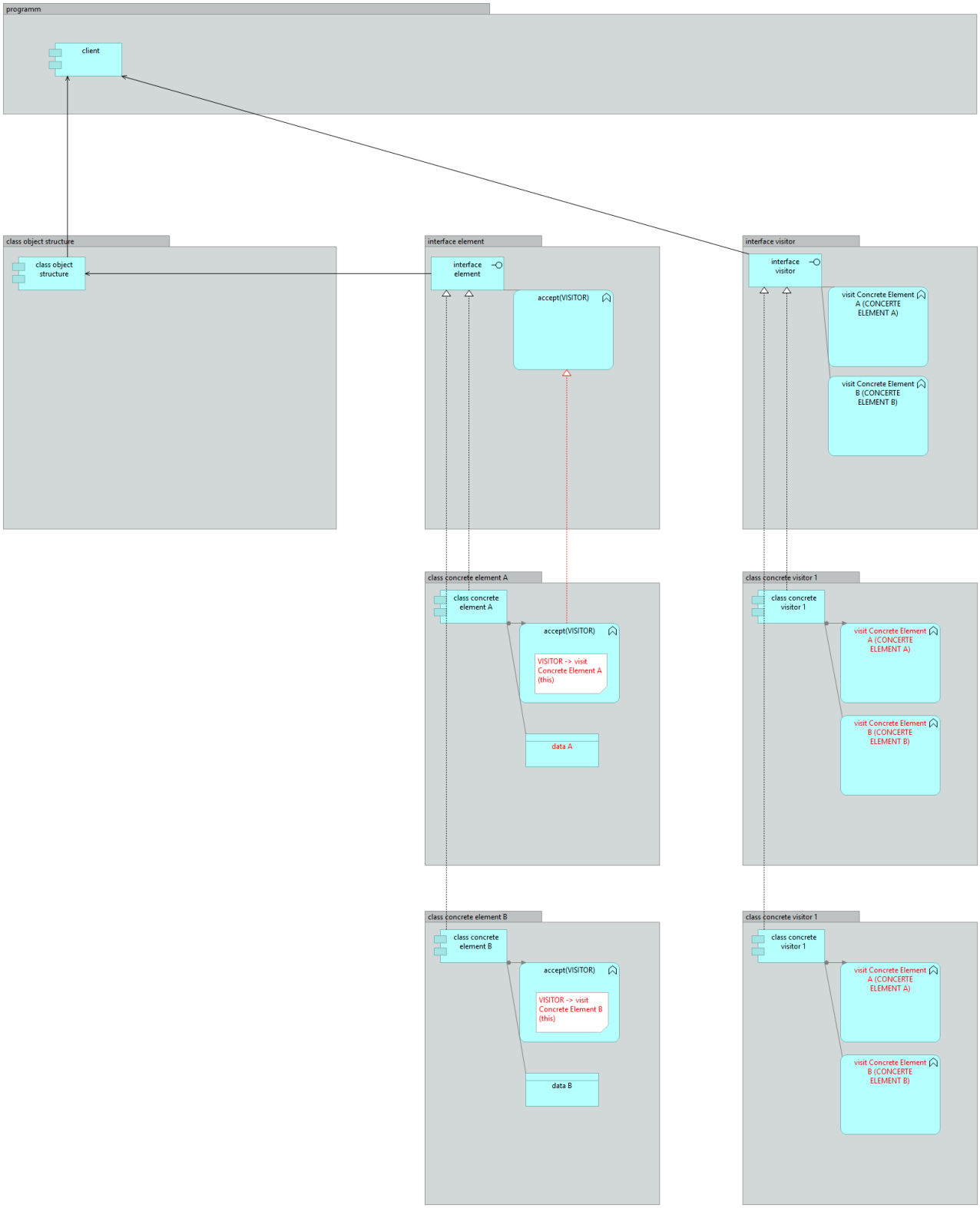
Varian2) strategy

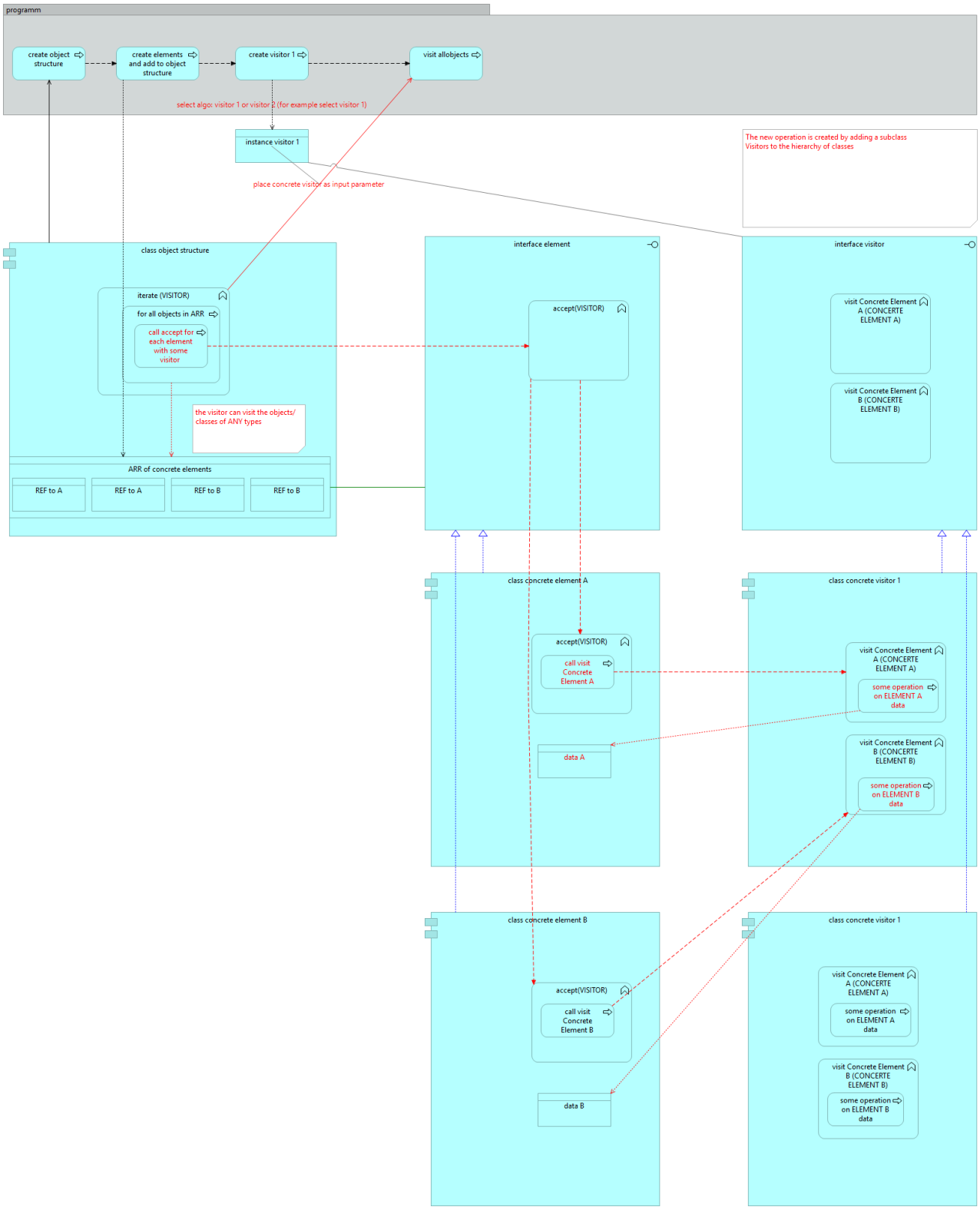


Varian3) lambdas as input parameters



VISITOR


programm
client
class object structure
class object structure
interface element
interface element
accept(VISITOR)
interface visitor
interface visitor
visit Concrete Element A (CONCERTE ELEMENT A)
visit Concrete Element B (CONCERTE ELEMENT B)
class concrete element A
class concrete element A
accept(VISITOR)
VISITOR -> visit Concrete Element A (this)
data A
class concrete visitor 1
class concrete visitor 1
visit Concrete Element A (CONCERTE ELEMENT A)
visit Concrete Element B (CONCERTE ELEMENT B)
class concrete element B
class concrete element B
accept(VISITOR)
VISITOR -> visit Concrete Element B (this)
data B
class concrete visitor 1
class concrete visitor 1
visit Concrete Element A (CONCERTE ELEMENT A)
visit Concrete Element B (CONCERTE ELEMENT B)


programm
create object structure
create elements and add to object structure
create visitor 1
visit allobjects
select algo: visitor 1 or visitor 2 (for example select visitor 1)
instance visitor 1
place concrete visitor as input parameter
The new operation is created by adding a subclass Visitors to the hierarchy of classes
class object structure
iterate (VISITOR)
for all objects in ARR
call accept for each element with some visitor
the visitor can visit the objects/ classes of ANY types
ARR of concrete elements
REF to A
REF to A
REF to B
REF to B
interface element
accept(VISITOR)
interface visitor
visit Concrete Element A (CONCERTE ELEMENT A)
visit Concrete Element B (CONCERTE ELEMENT B)
class concrete element A
accept(VISITOR)
call visit Concrete Element A
data A
class concrete visitor 1
visit Concrete Element A (CONCERTE ELEMENT A)
some operation on ELEMENT A data
visit Concrete Element B (CONCERTE ELEMENT B)
some operation on ELEMENT B data
class concrete element B
accept(VISITOR)
call visit Concrete Element B
data B
class concrete visitor 1
visit Concrete Element A (CONCERTE ELEMENT A)
some operation on ELEMENT A data
visit Concrete Element B (CONCERTE ELEMENT B)
some operation on ELEMENT B data

02

Enterprise Patterns

Catalog of Patterns of Enterprise Application Architecture

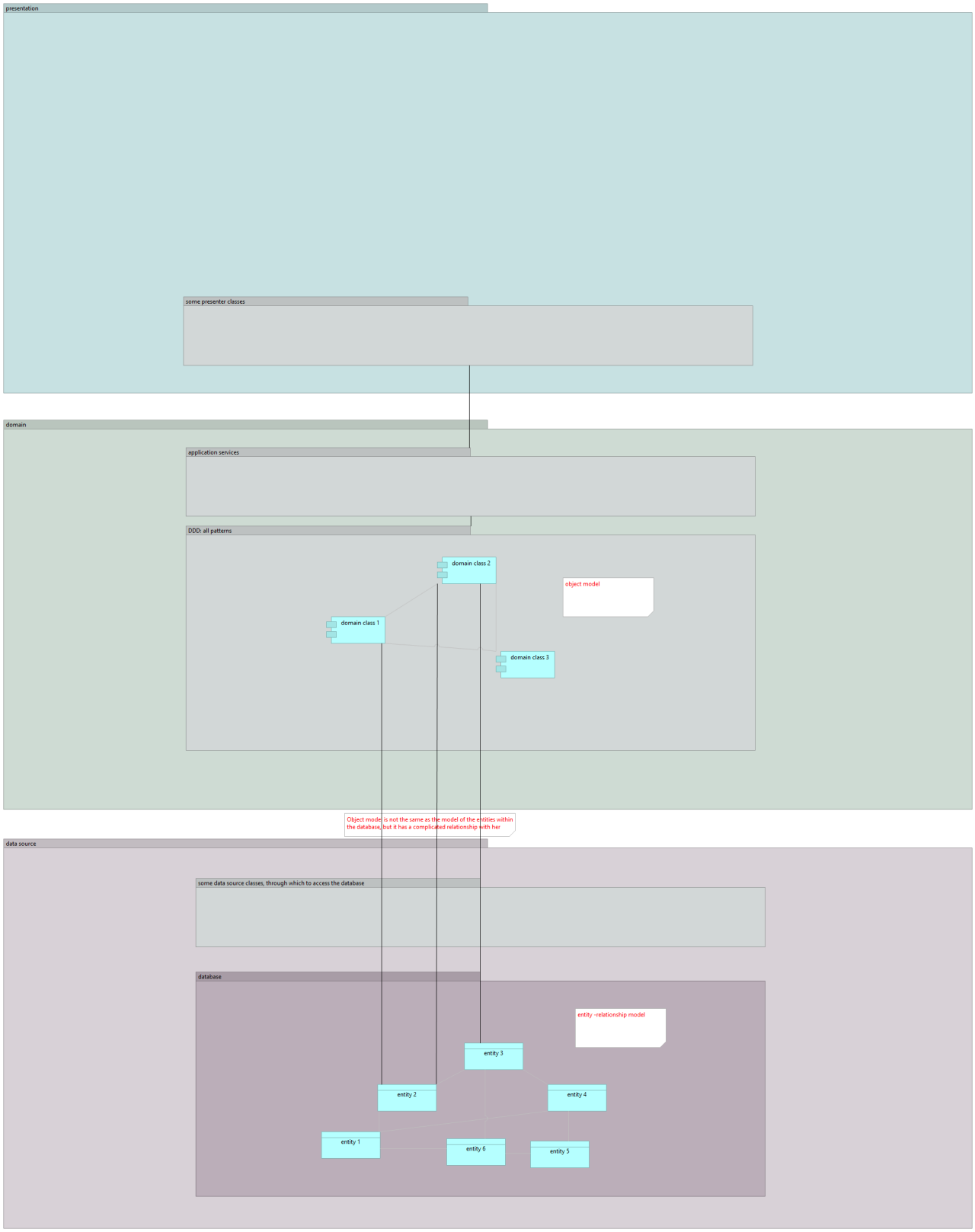
MARTIN FOWLER

BUSINESS LOGIC

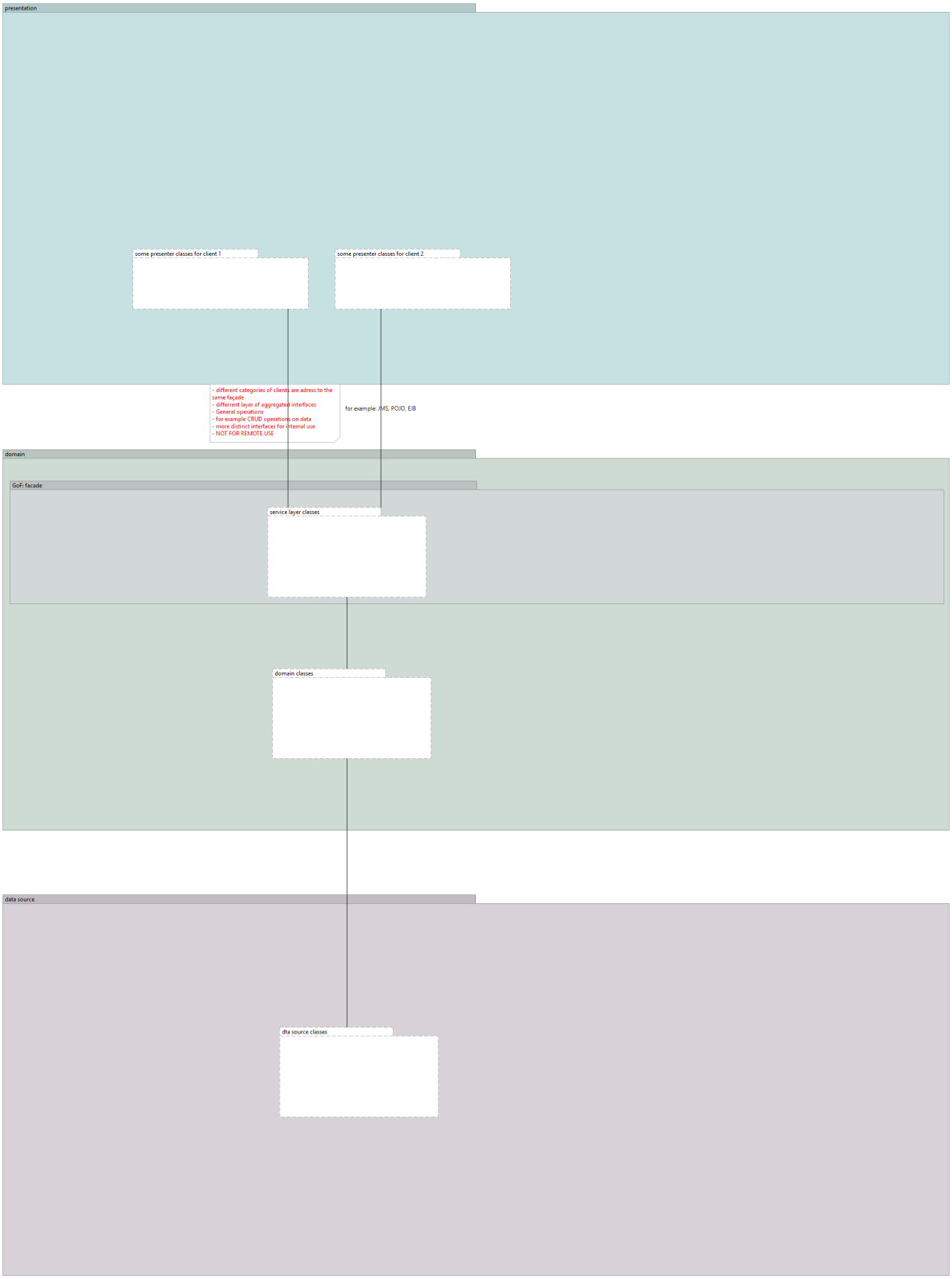
ENTERPRISE PATTERNS

Martin Fowler

DOMAIN MODEL



SERVICE LAYER



TRANSACTION SCRIPT

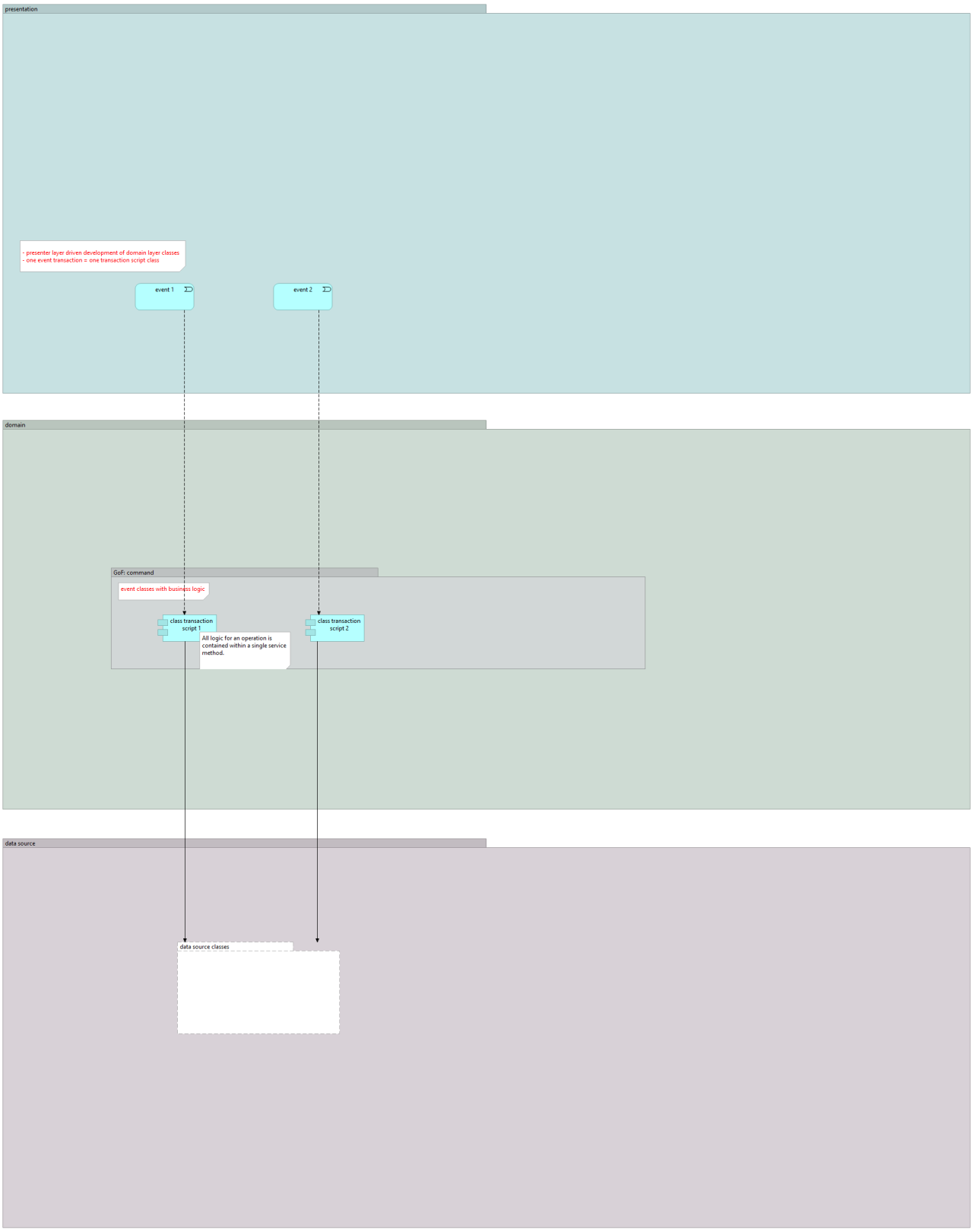
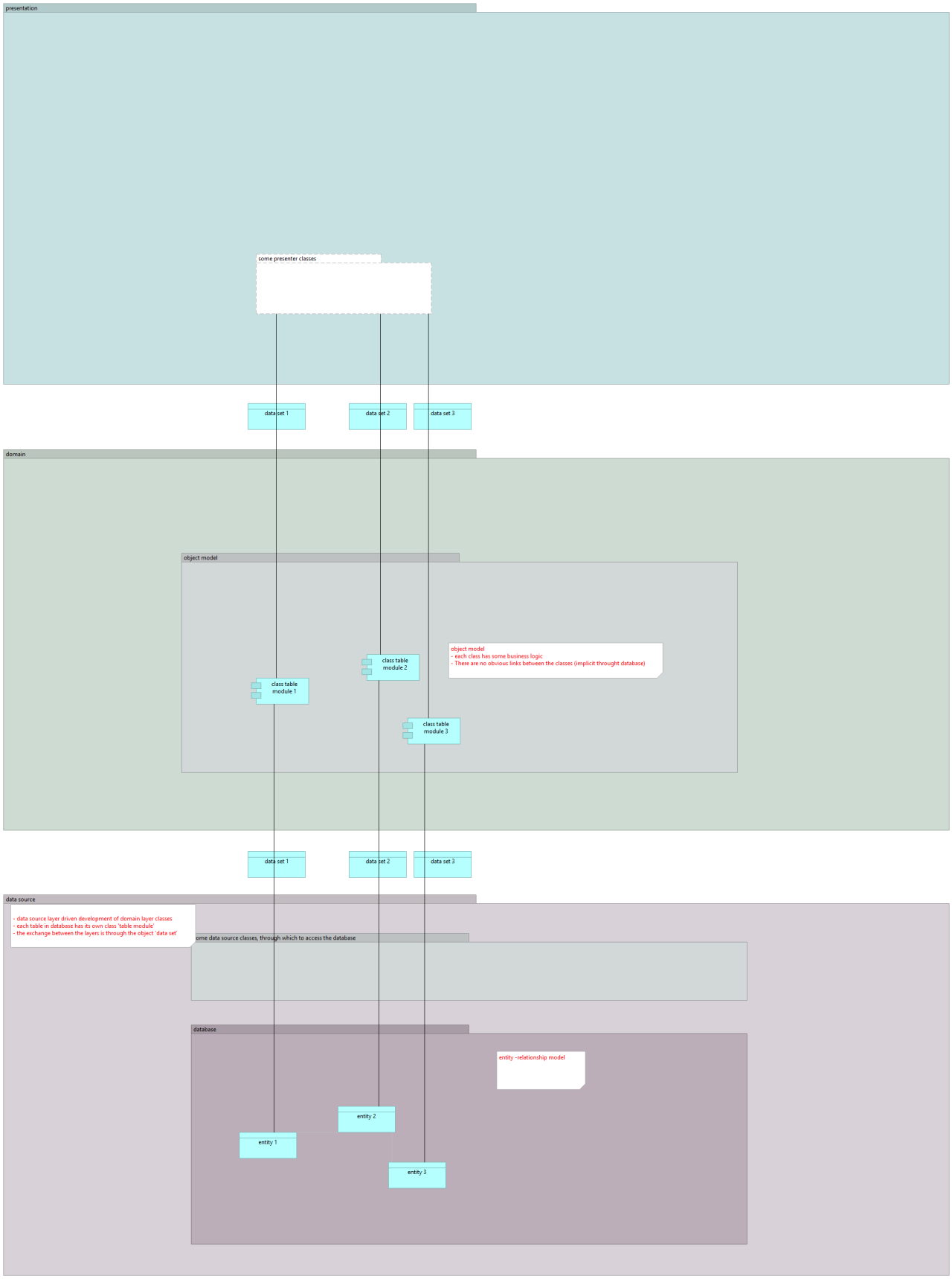


TABLE MODULE

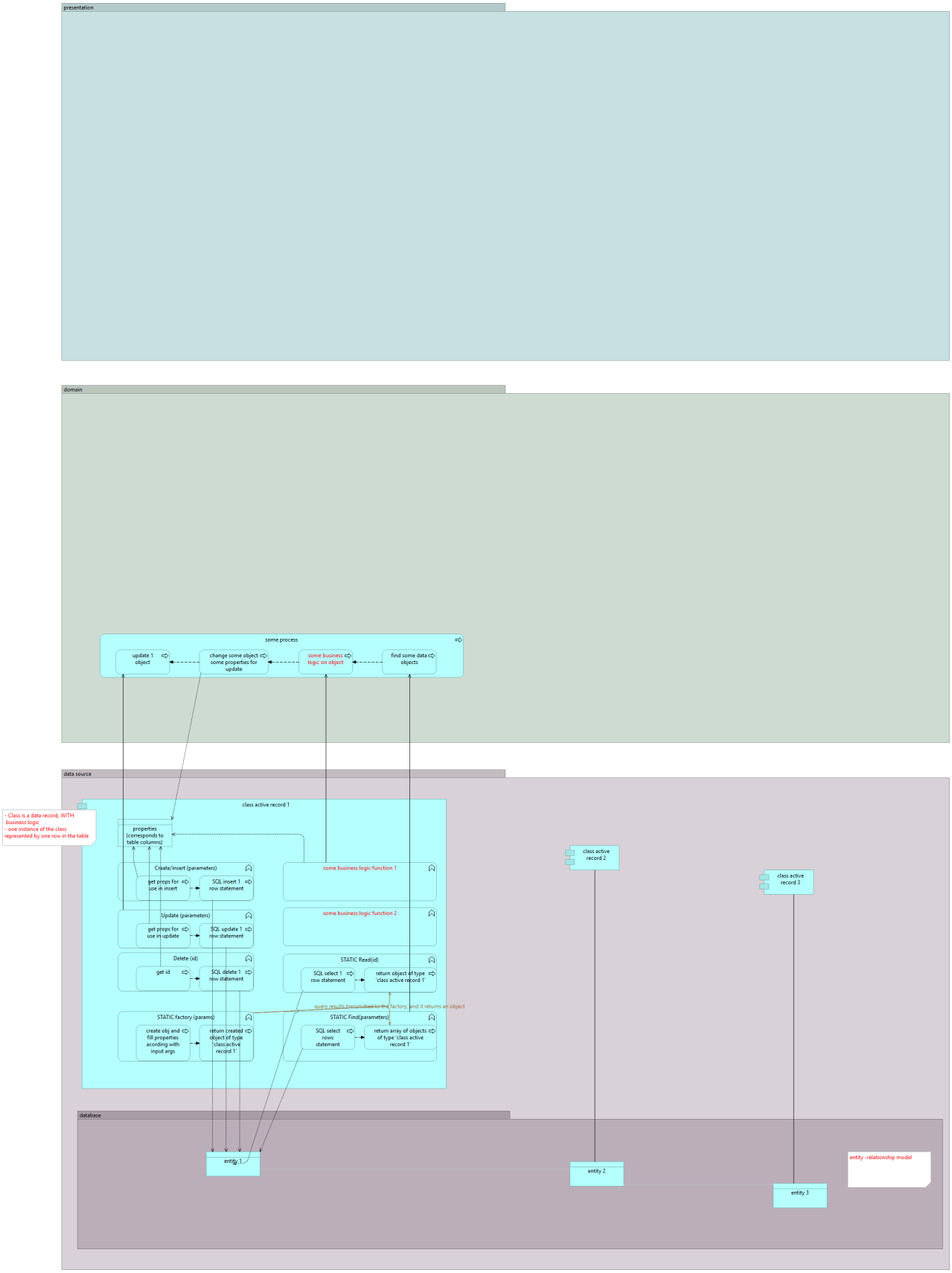


DATA SOURCES

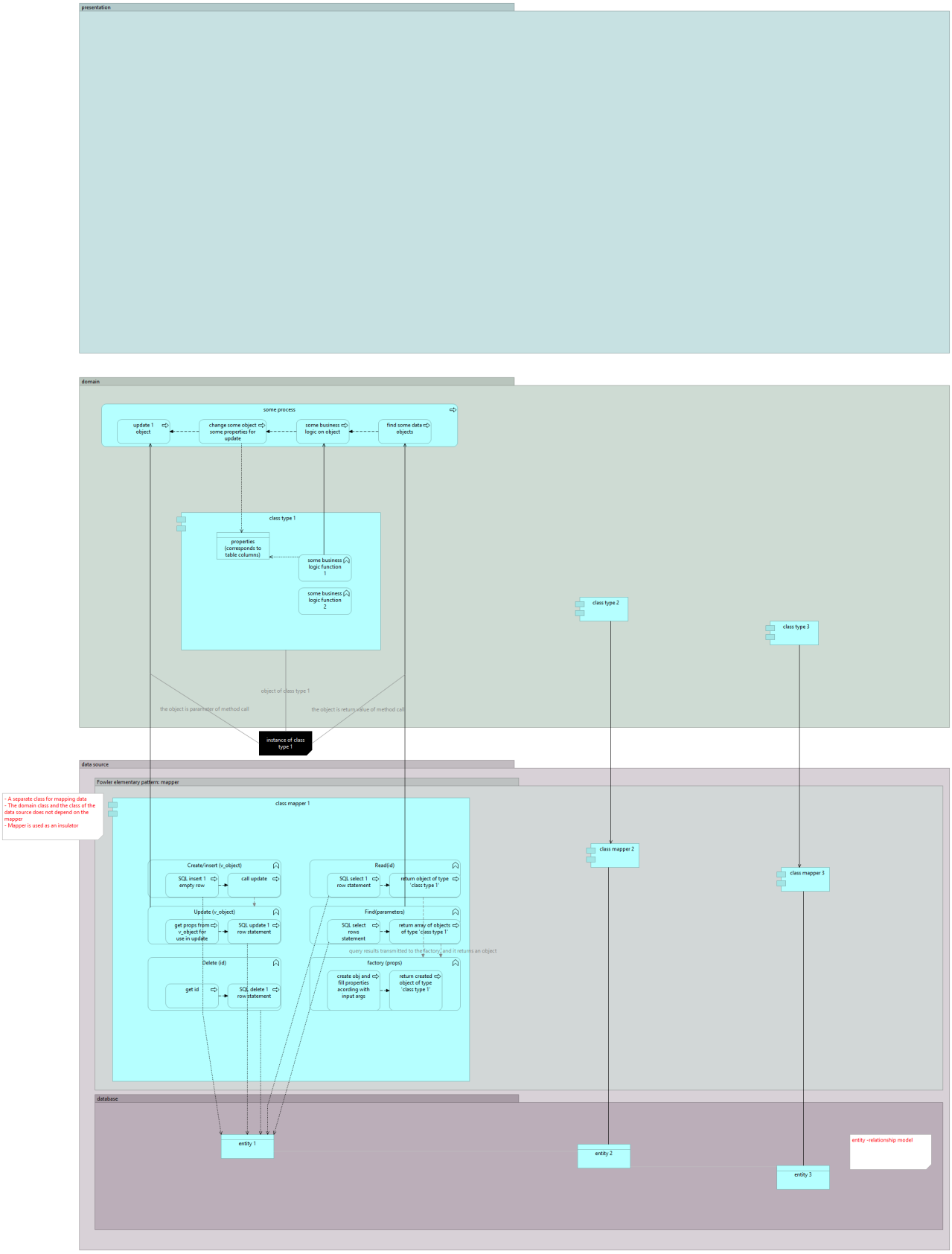
ENTERPRISE PATTERNS

Martin Fowler

ACTIVE RECORD



DATA MAPPER


presentation
domain
some process
update 1 object
change some object some properties for update
some business logic on object
find some data objects
class type 1
properties (corresponds to table columns)
some business logic function 1
some business logic function 2
class type 2
class type 3
object of class type 1
the object is parameter of method call
the object is return value of method call
instance of class type 1
data source
Fowler elementary pattern: mapper
- A separate class for mapping data
- The domain class and the class of the data source does not depend on the mapper
- Mapper is used as an insulator
class mapper 1
Create/insert (v_object)
SQL insert 1 empty row
call update
Read(id)
SQL select 1 row statement
return object of type `class type 1`
Update (v_object)
get props from v_object for use in update
SQL update 1 row statement
Find(parameters)
SQL select rows statement
return array of objects of type `class type 1`
query results transmitted to the factory, and it returns an object
Delete (id)
get id
SQL delete 1 row statement
factory (props)
create obj and fill properties acording with input args
return created object of type `class type 1`
class mapper 2
class mapper 3
database
entity 1
entity 2
entity 3
entity -relationship model

ROW DATA GATEWAY

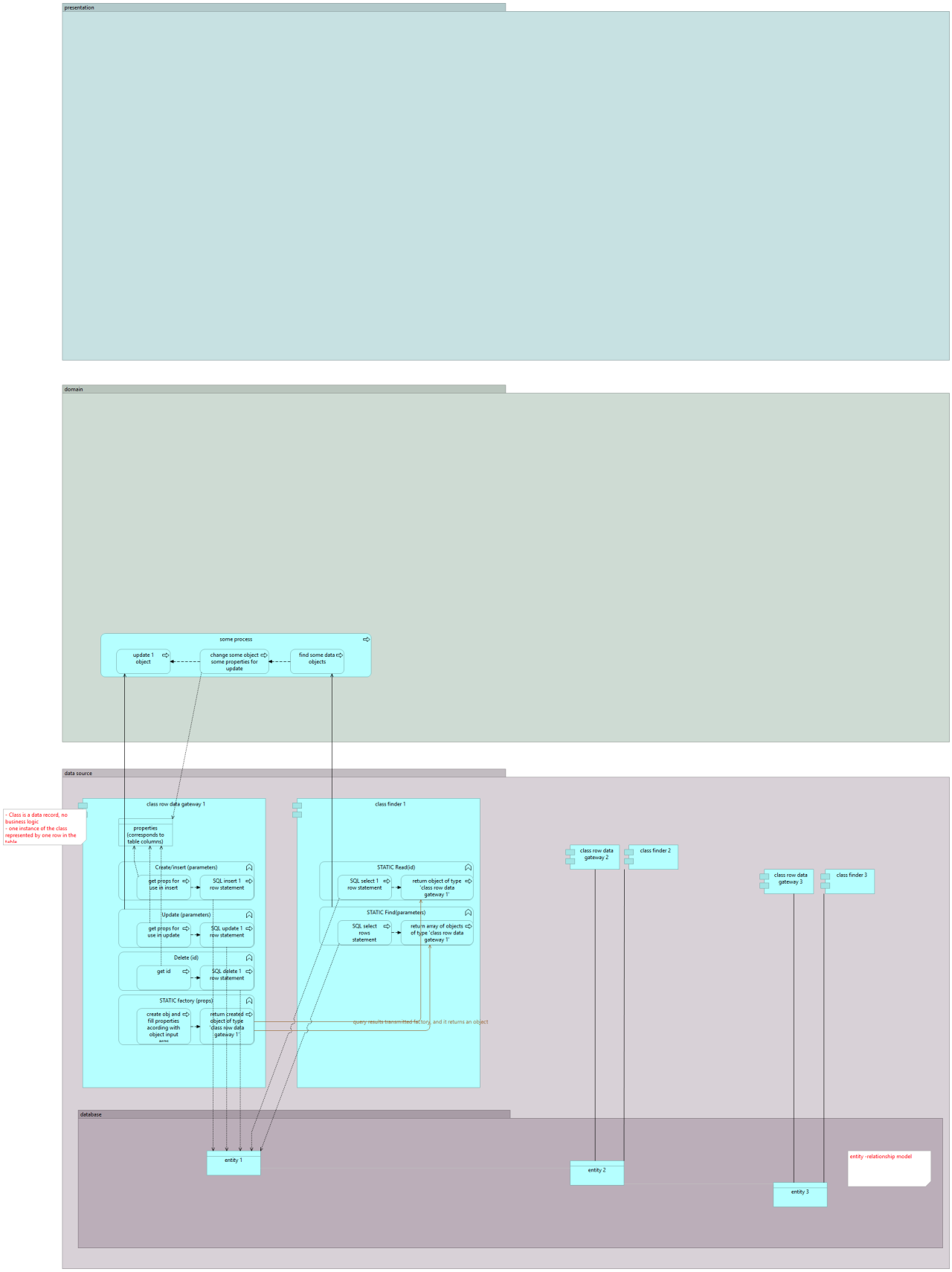
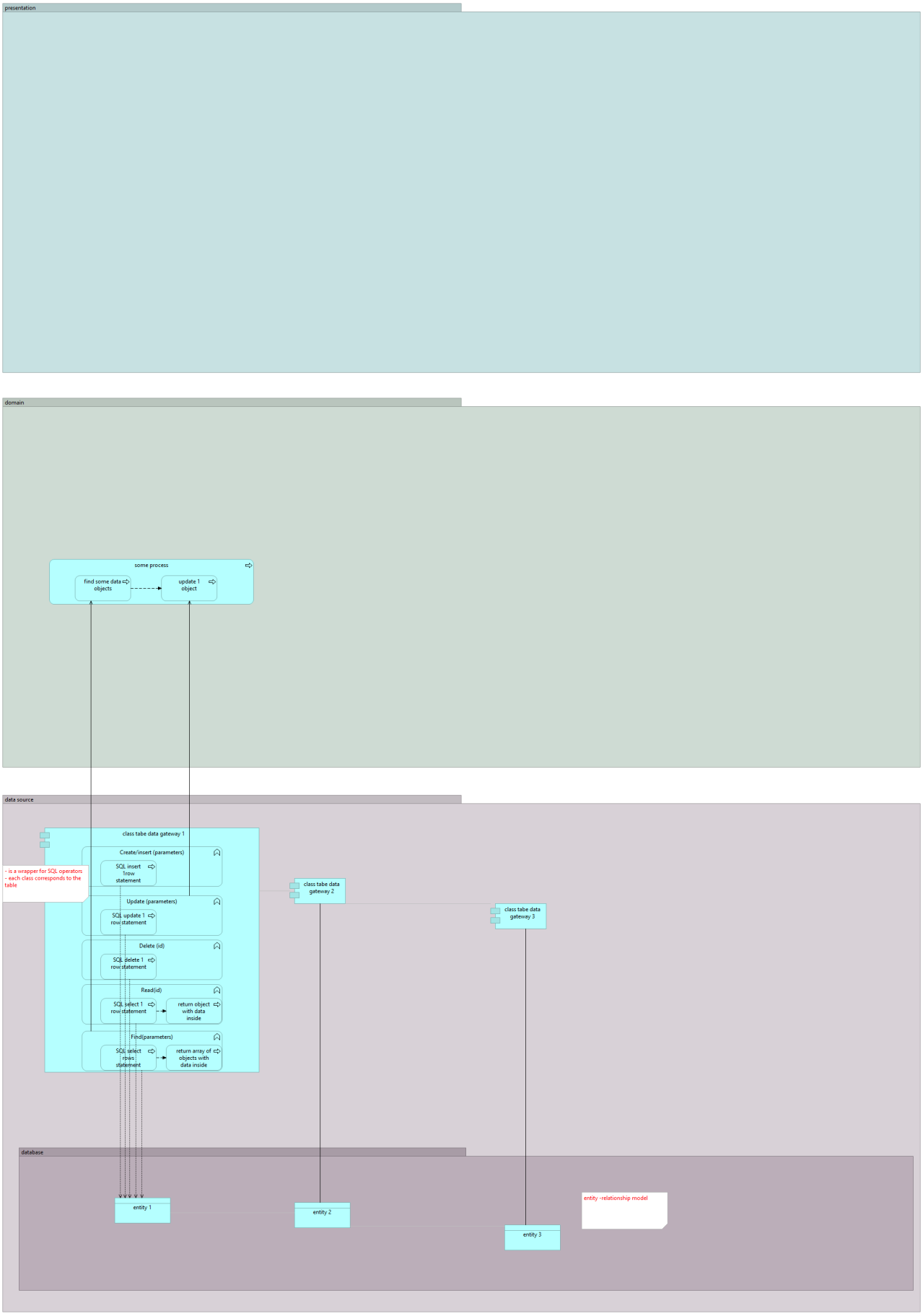


TABLE DATA GATEWAY

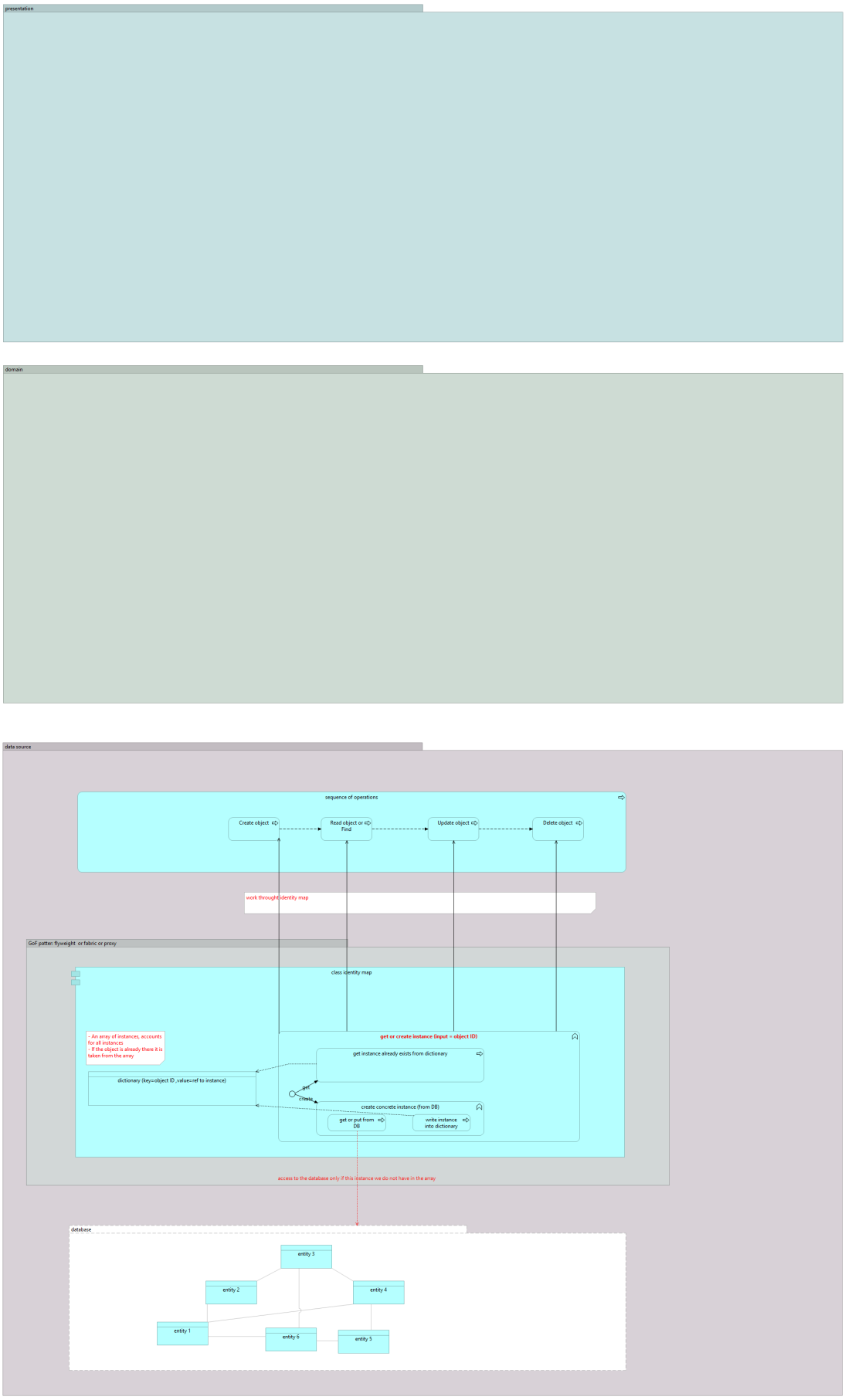


MODELING BEHAVIOR

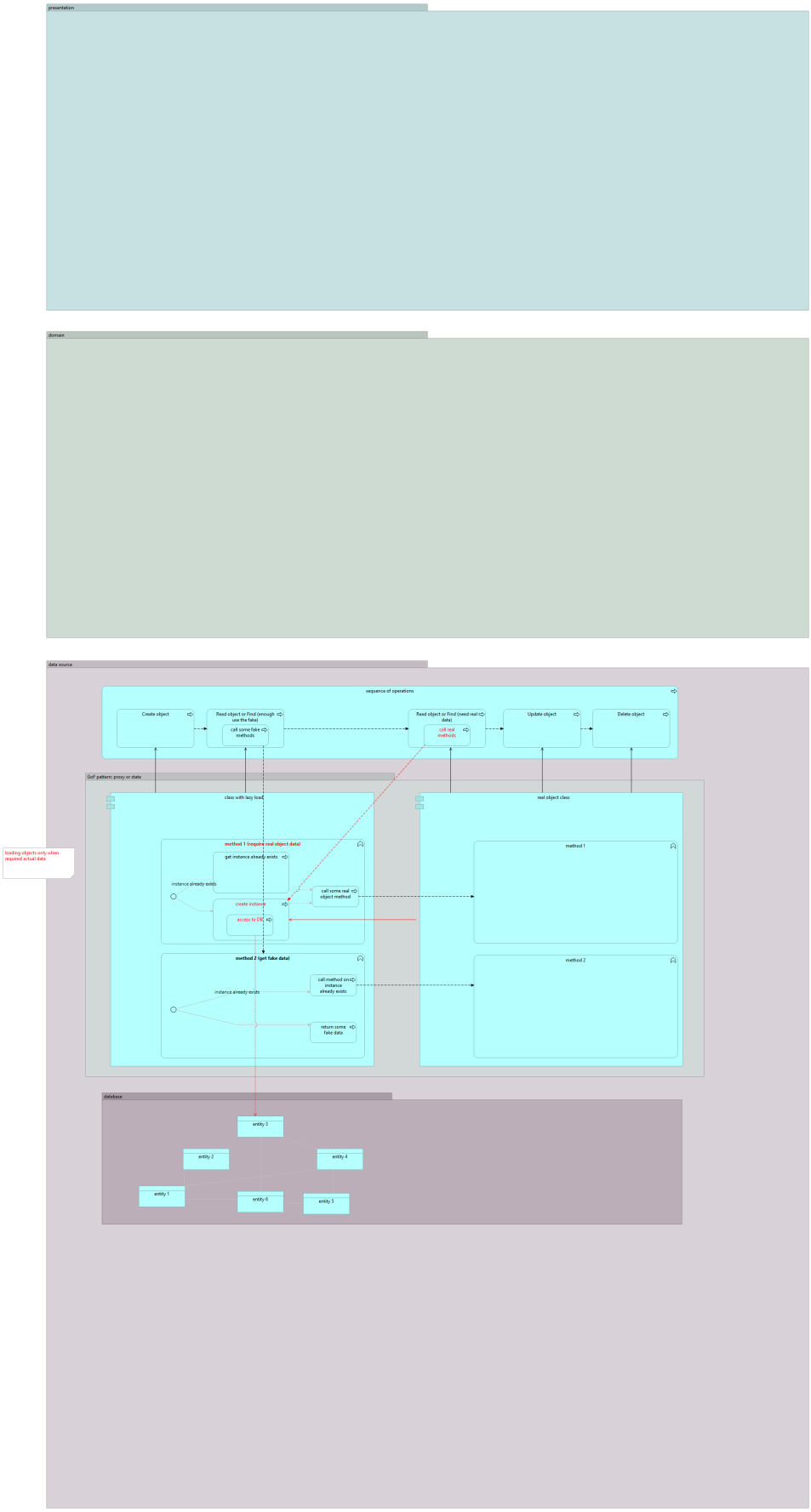
ENTERPRISE PATTERNS

Martin Fowler

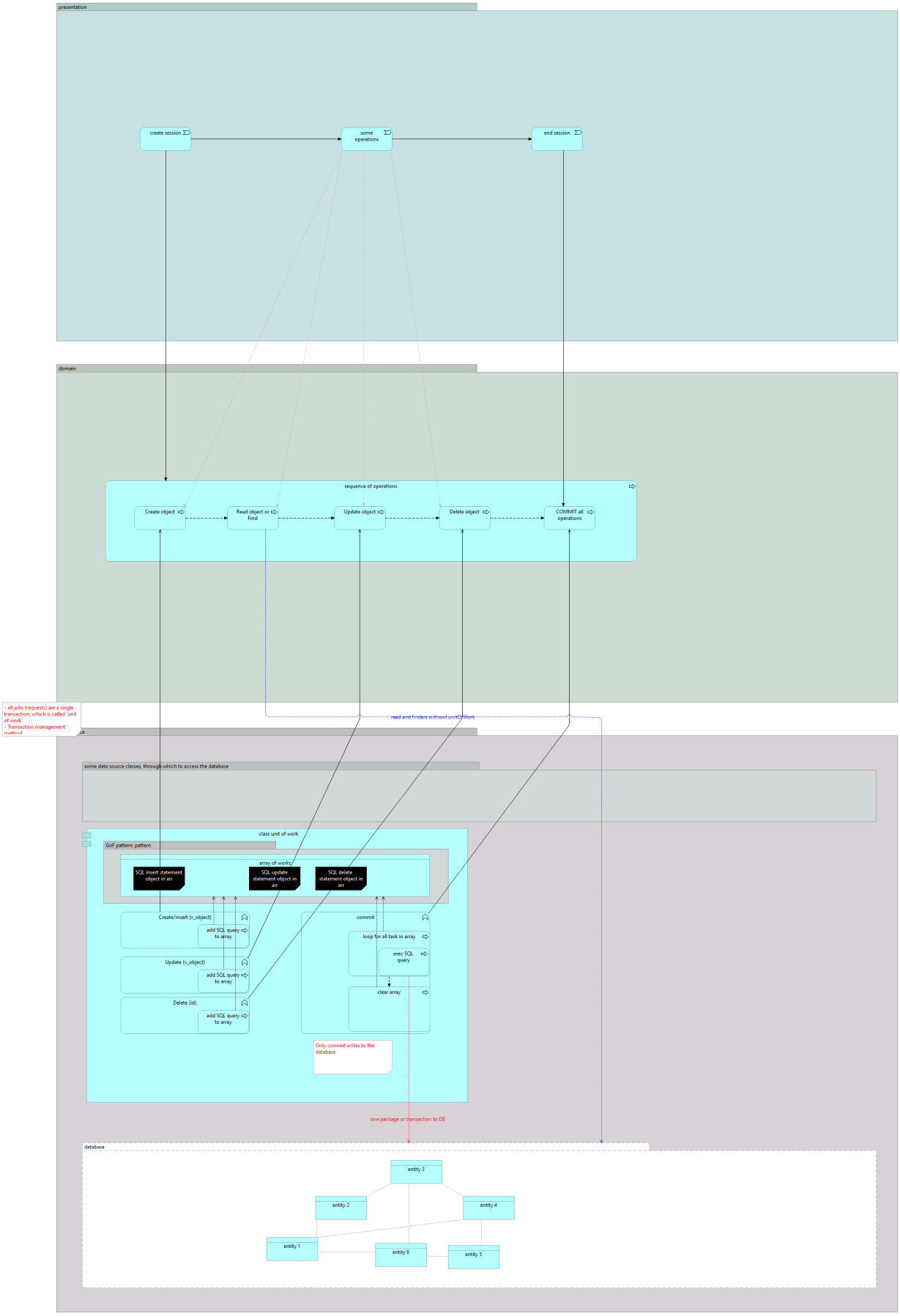
IDENTITY MAP



LAZY LOAD


presentation
domain
data source
sequence of operations
Create object
Read object or Find (enough use the fake)
call some fake methods
Read object or Find (need real data)
call real methods
Update object
Delete object
GoF pattern: proxy or state
class with lazy load
real object class
loading objects only when required actual data
method 1 (require real object data)
get instance already exists
instance already exists
create instance
access to DB
call some real object method
method 1
method 2 (get fake data)
instance already exists
call method on instance already exists
return some fake data
method 2
database
entity 3
entity 2
entity 4
entity 1
entity 6
entity 5

UNIT OF WORK.

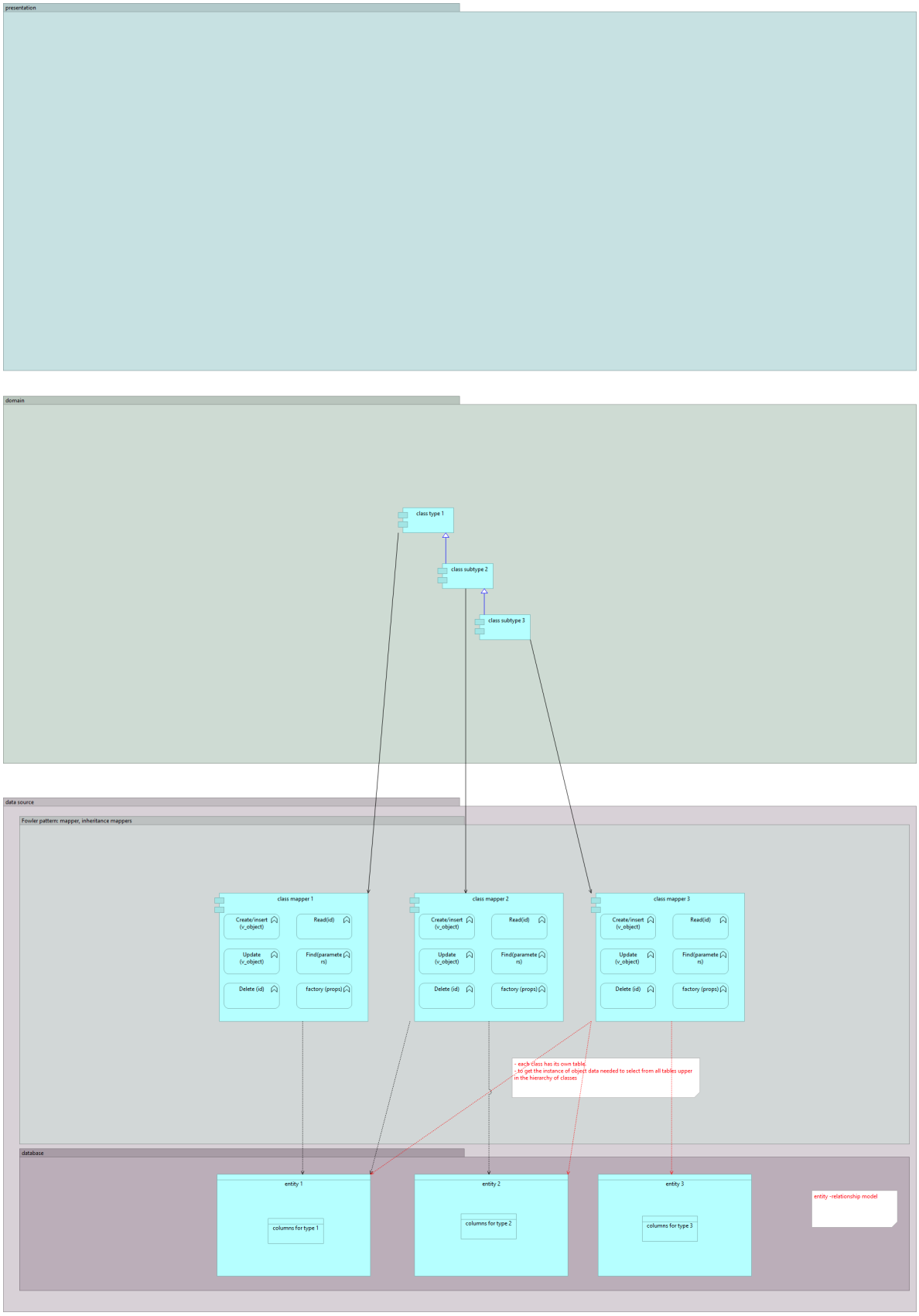

presentation
create session
some operations
end session
domain
sequence of operations
Create object
Read object or Find
Update object
Delete object
COMMIT all operations
- all jobs (requests) are a single transaction, which is called 'unit of work'
- Transaction management method
read and finders without unitOfWork
some data source classes, through which to access the database
class unit of work
GoF pattern: pattern
array of works
SQL insert statement object in arr
SQL update statement object in arr
SQL delete statement object in arr
Create/insert (v_object)
add SQL query to array
Update (v_object)
add SQL query to array
Delete (id)
add SQL query to array
commit
loop for all task in array
exec SQL query
clear array
Only commit writes to the database
one package or transaction to DB
database
entity 3
entity 2
entity 4
entity 1
entity 6
entity 5

MODELING STRUCTURE HIERARCHY

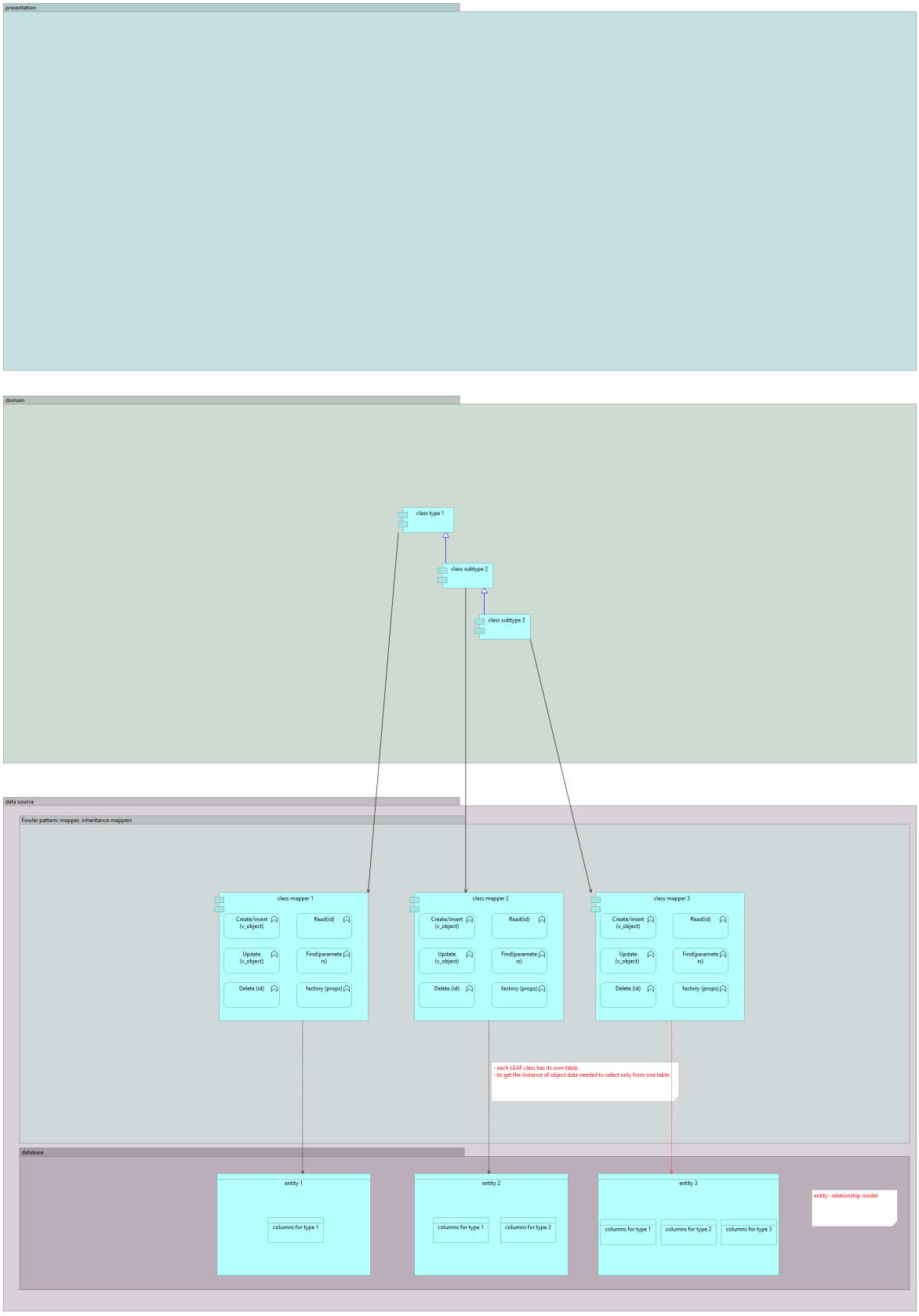
ENTERPRISE PATTERNS

Martin Fowler

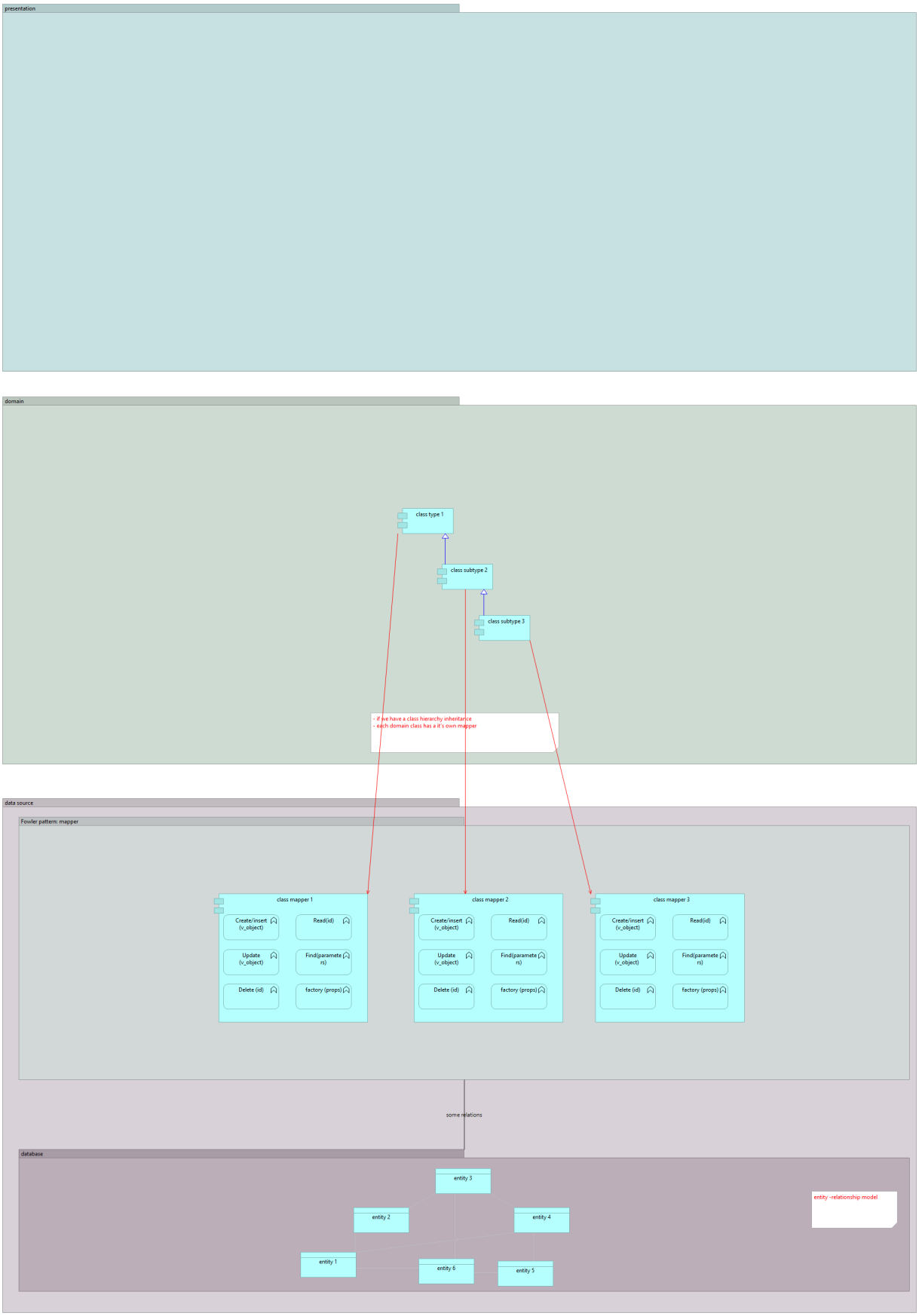
CLASS TABLE INHERITANCE



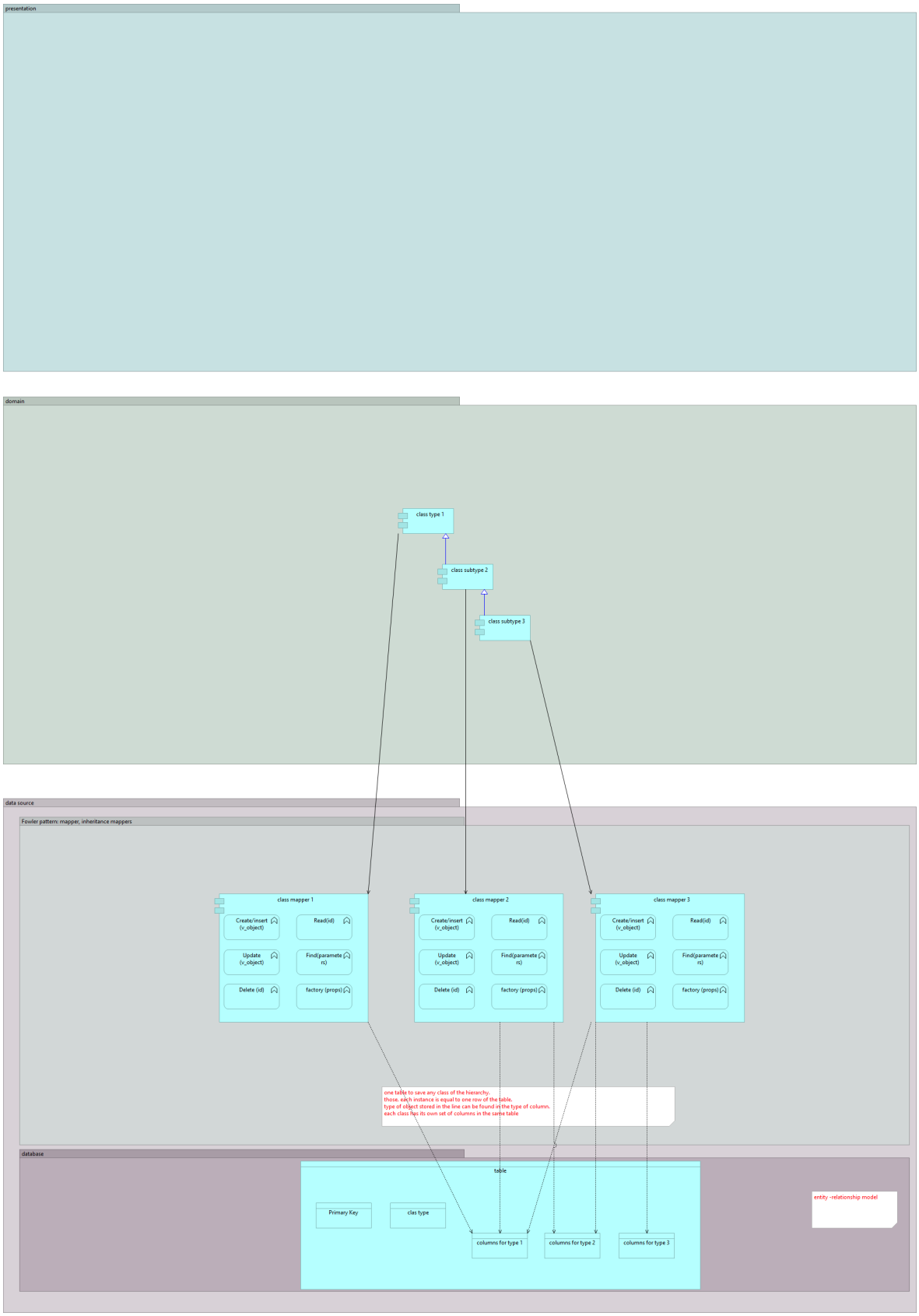
CONCRETE TABLE INHERITANCE



INHERITANCE MAPPERS



SINGLE TABLE INHERITANCE

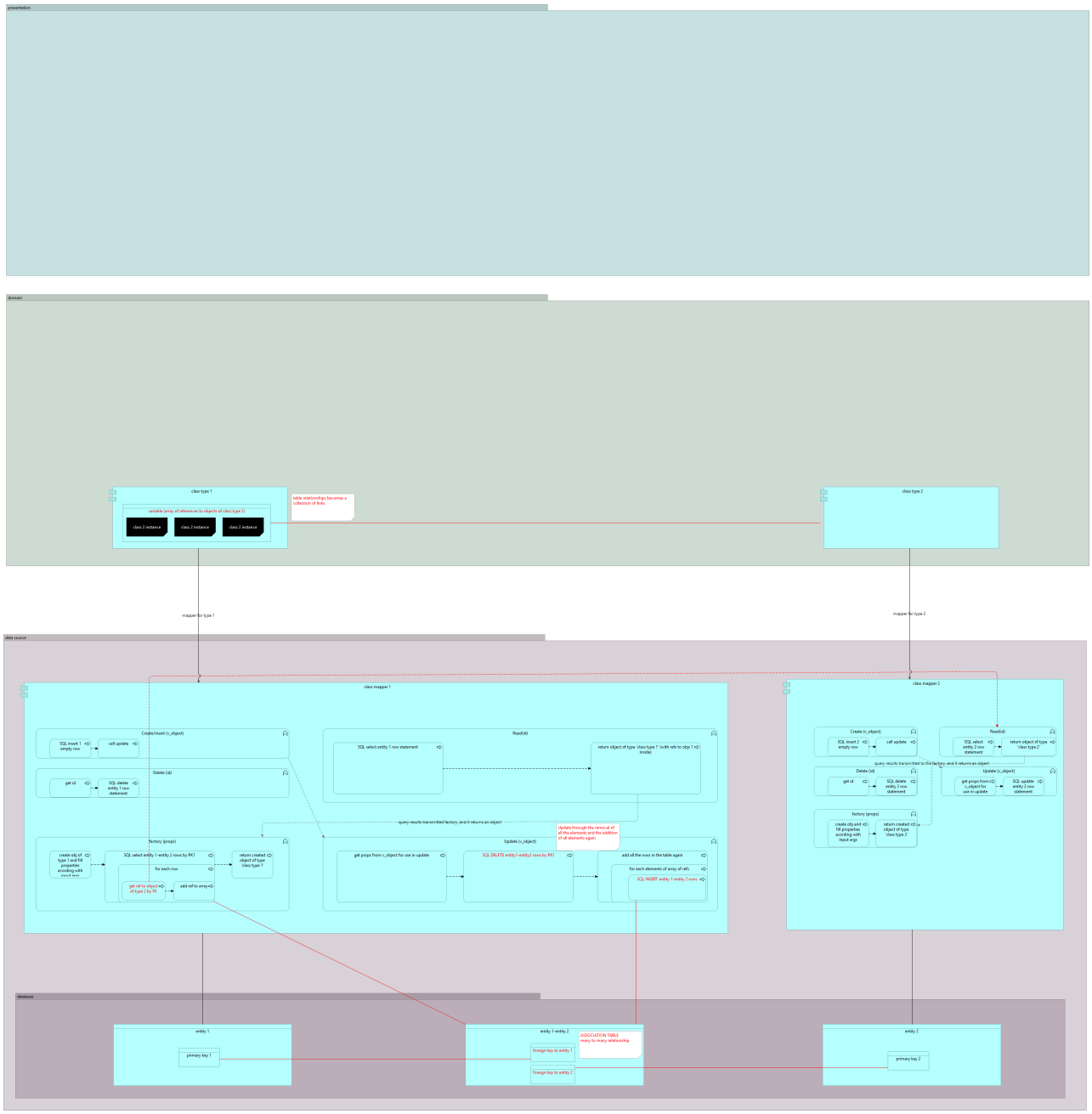


MODELING STRUCTURE RELATIONS

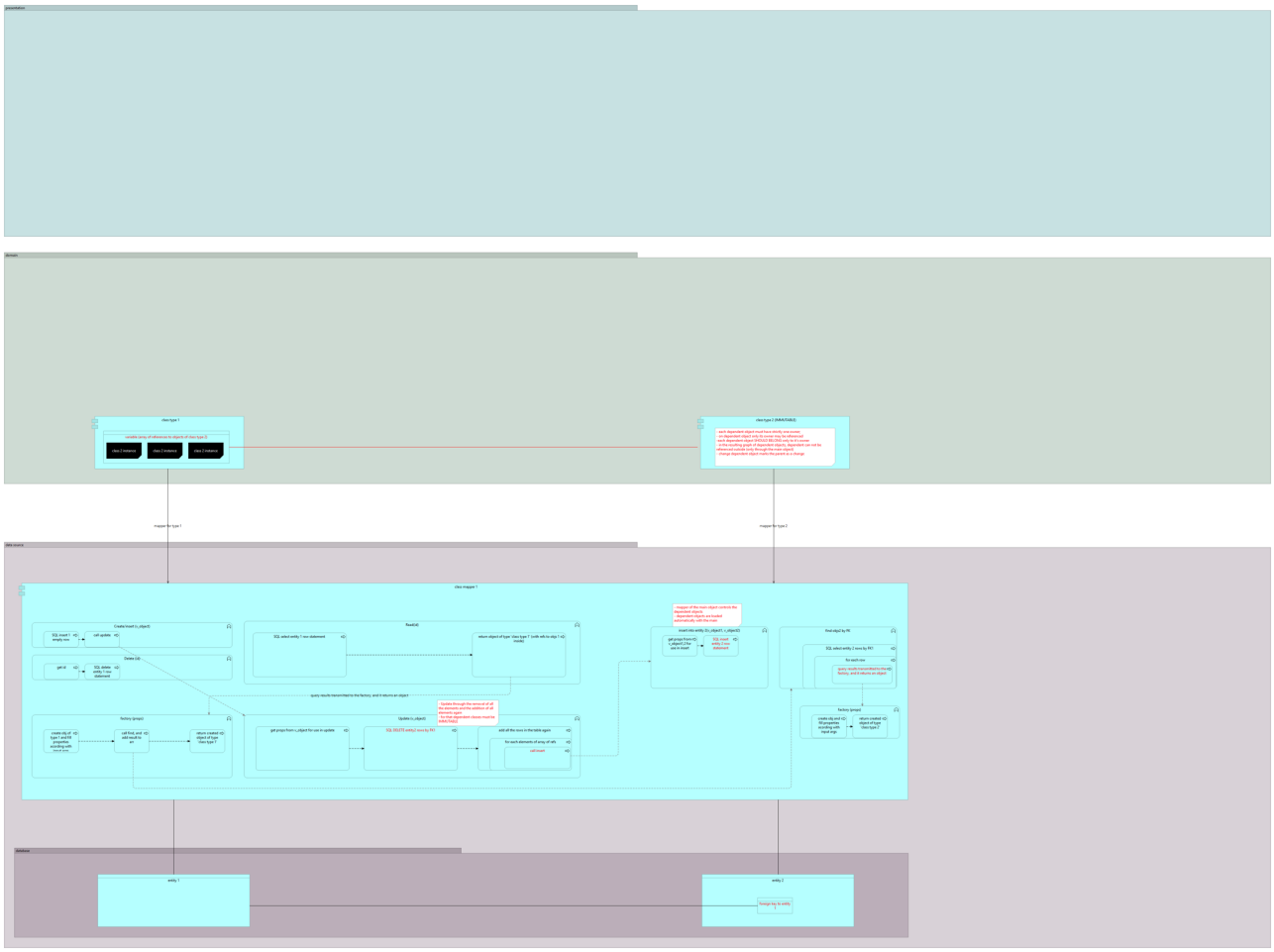
ENTERPRISE PATTERNS

Martin Fowler

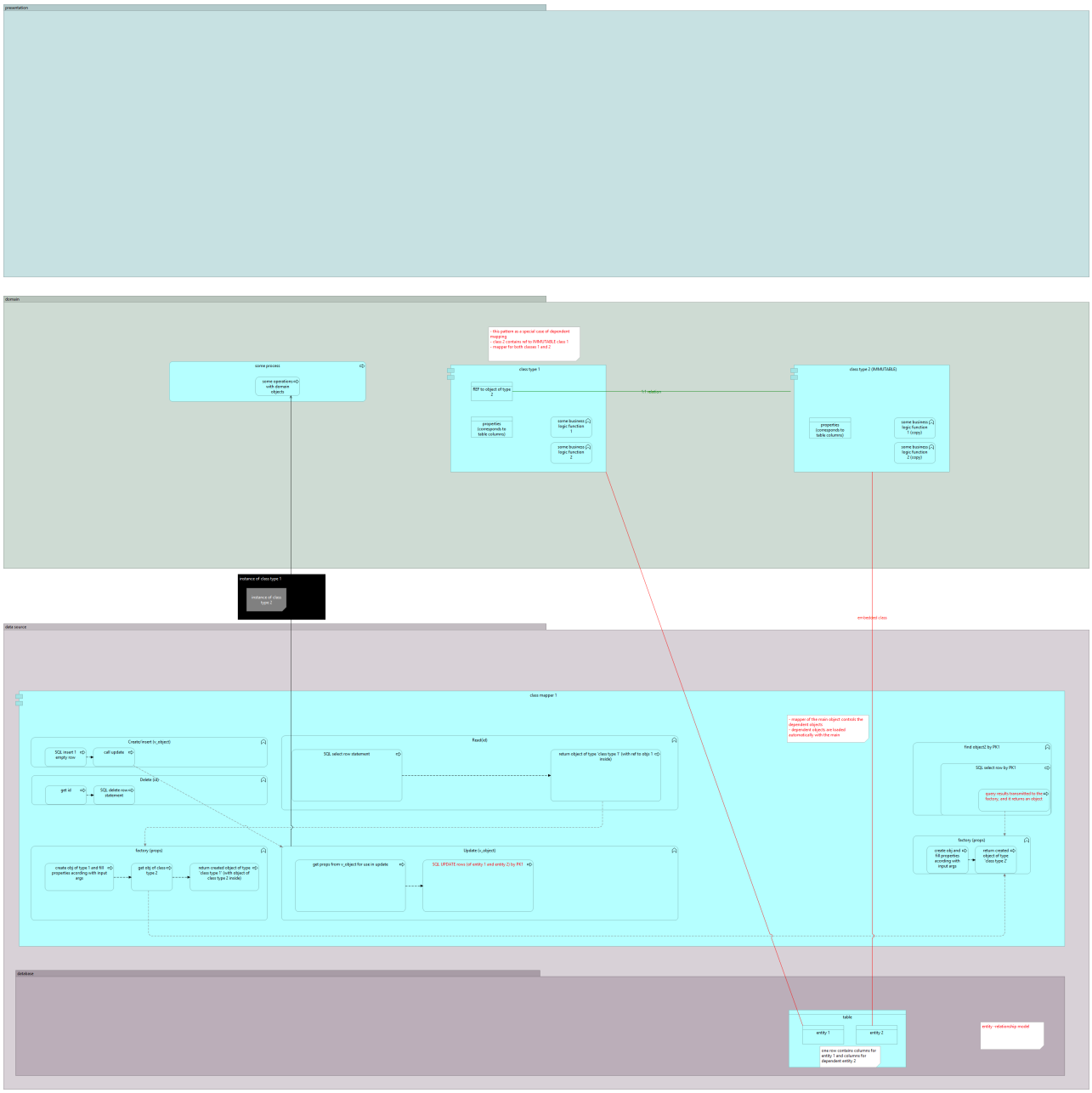
ASSOCIATION TABLE MAPPING



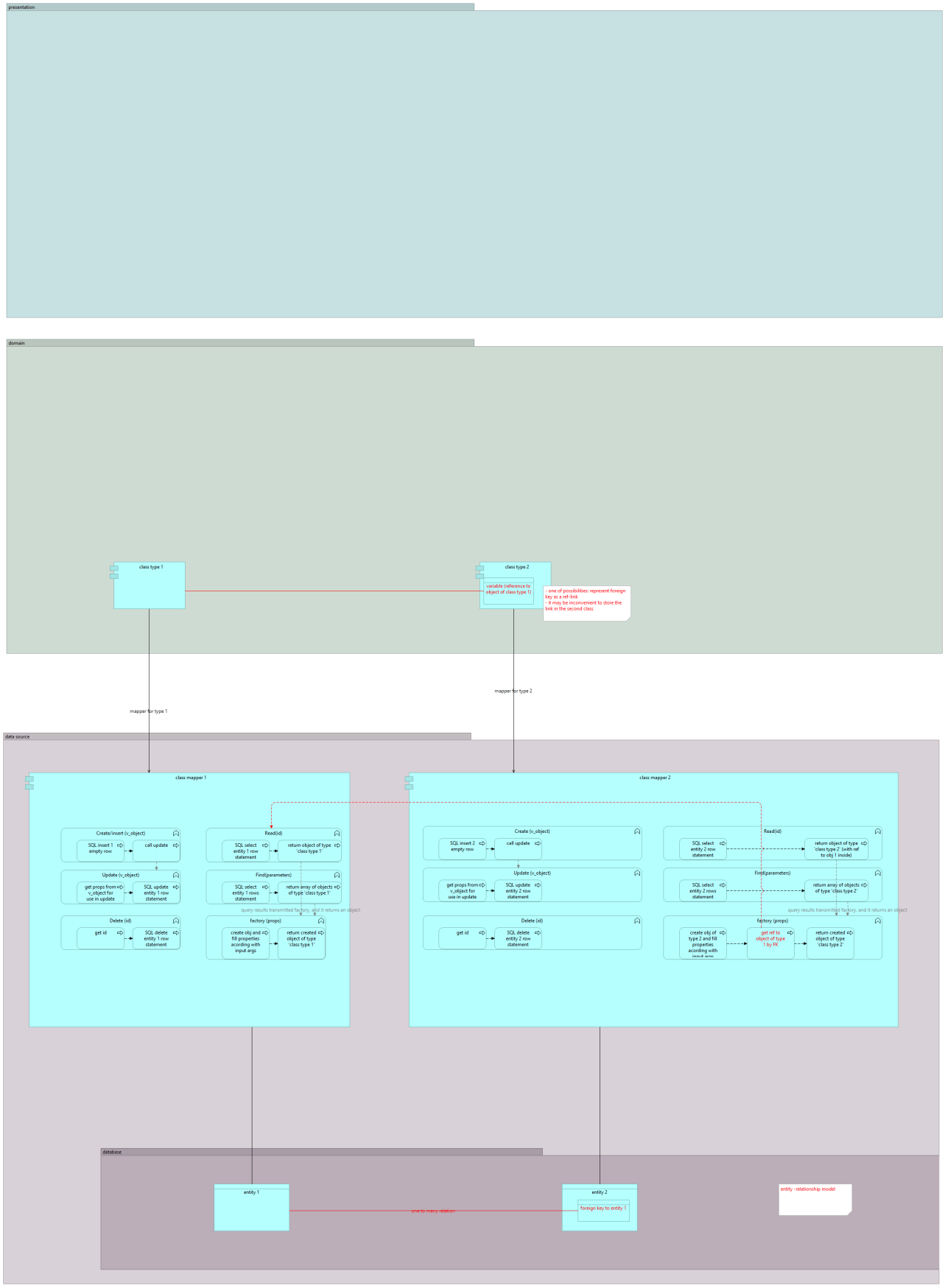
DEPENDENT MAPPING



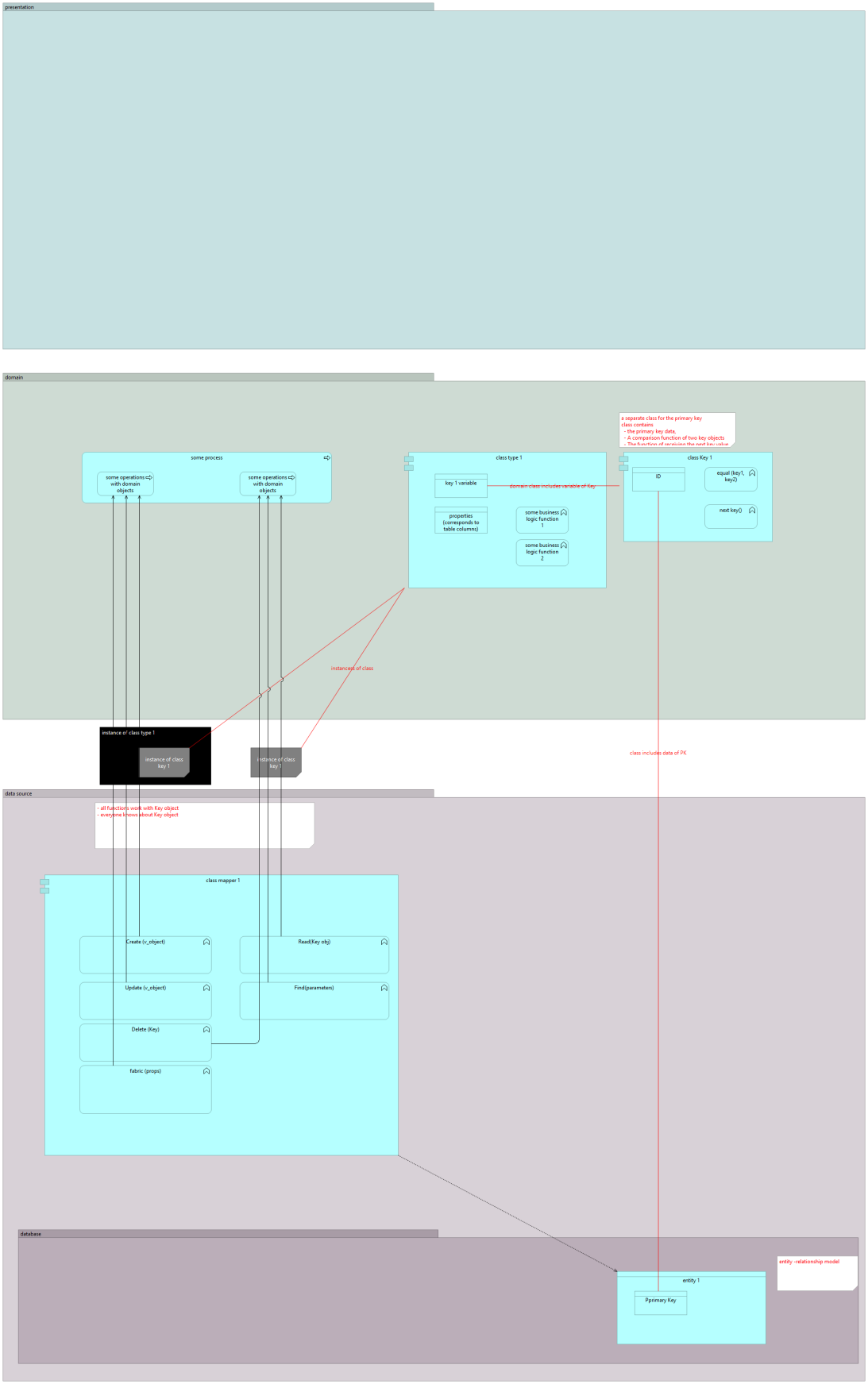
EMBEDDED VALUE



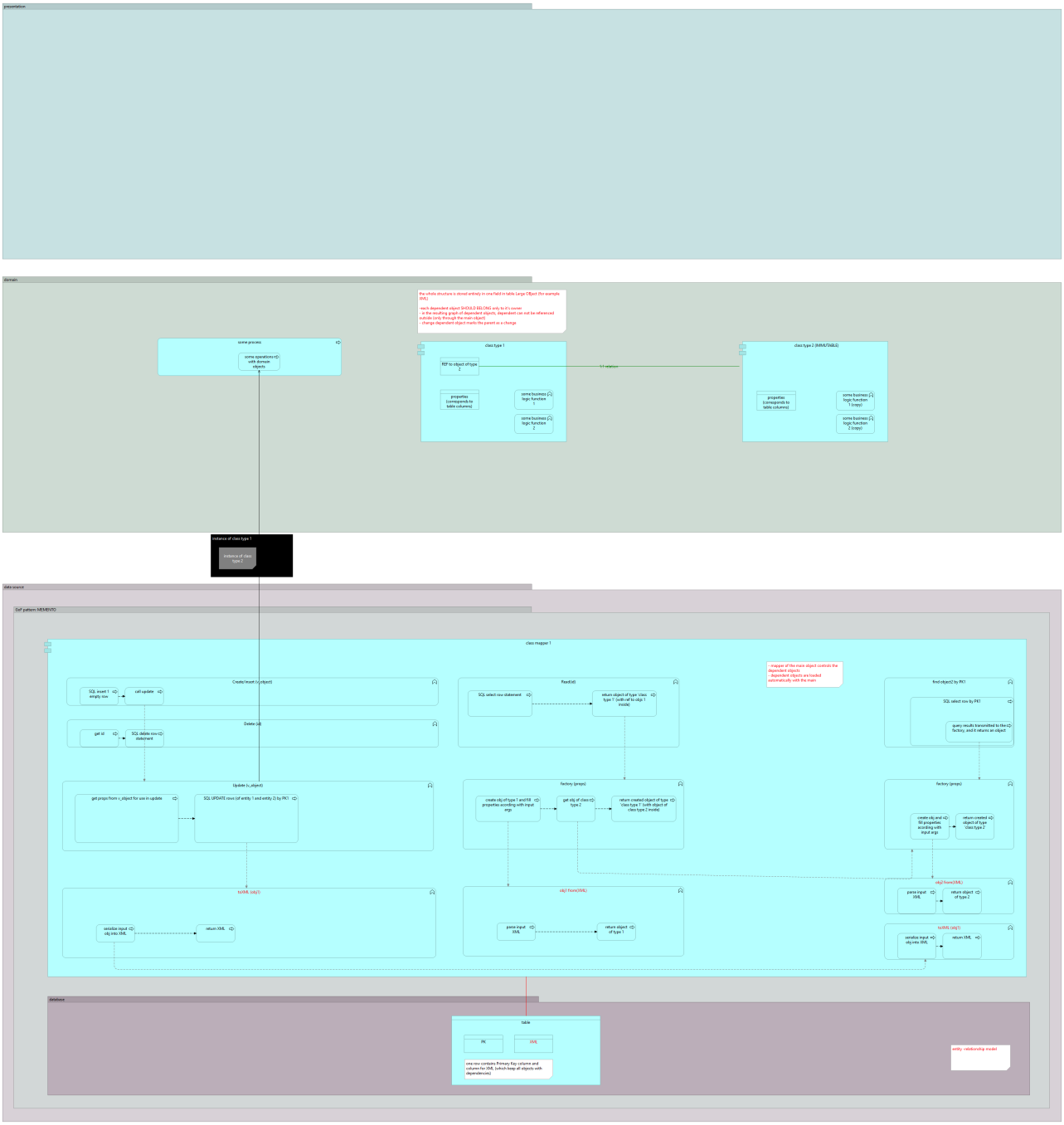
FOREIGN KEY MAPPING



IDENTITY FIELD



SERIALIZED LOB

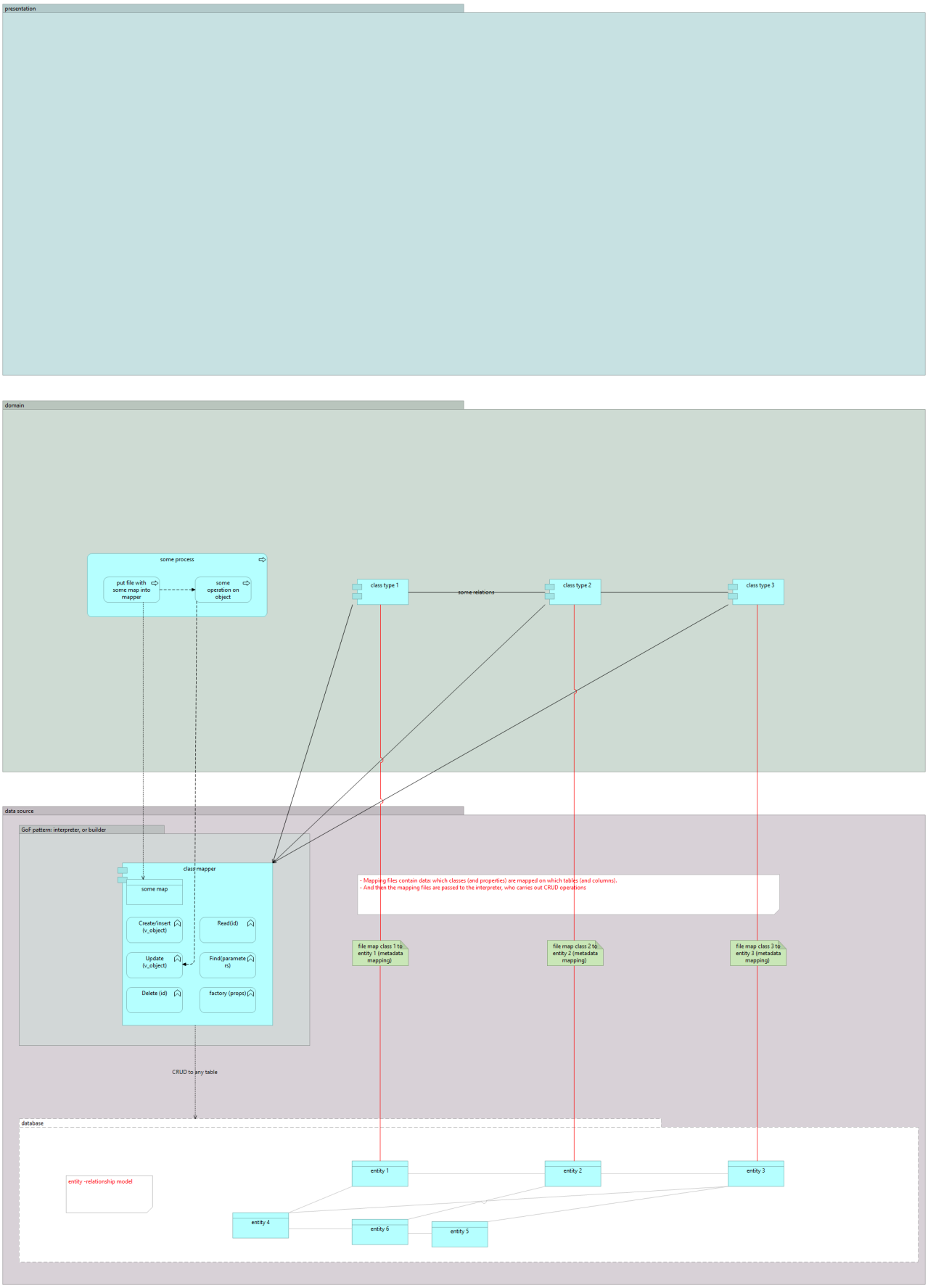


METADATA

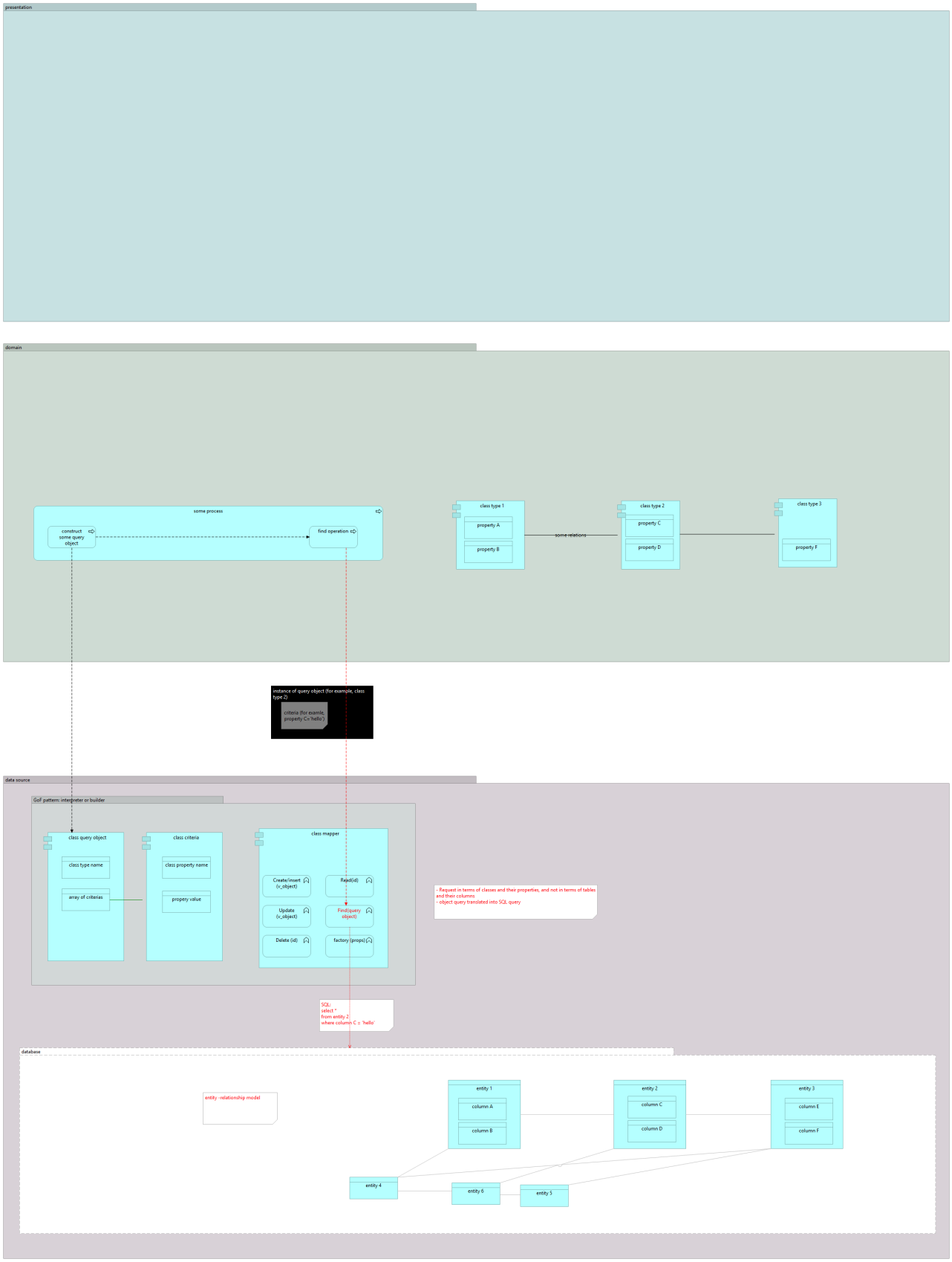
ENTERPRISE PATTERNS

Martin Fowler

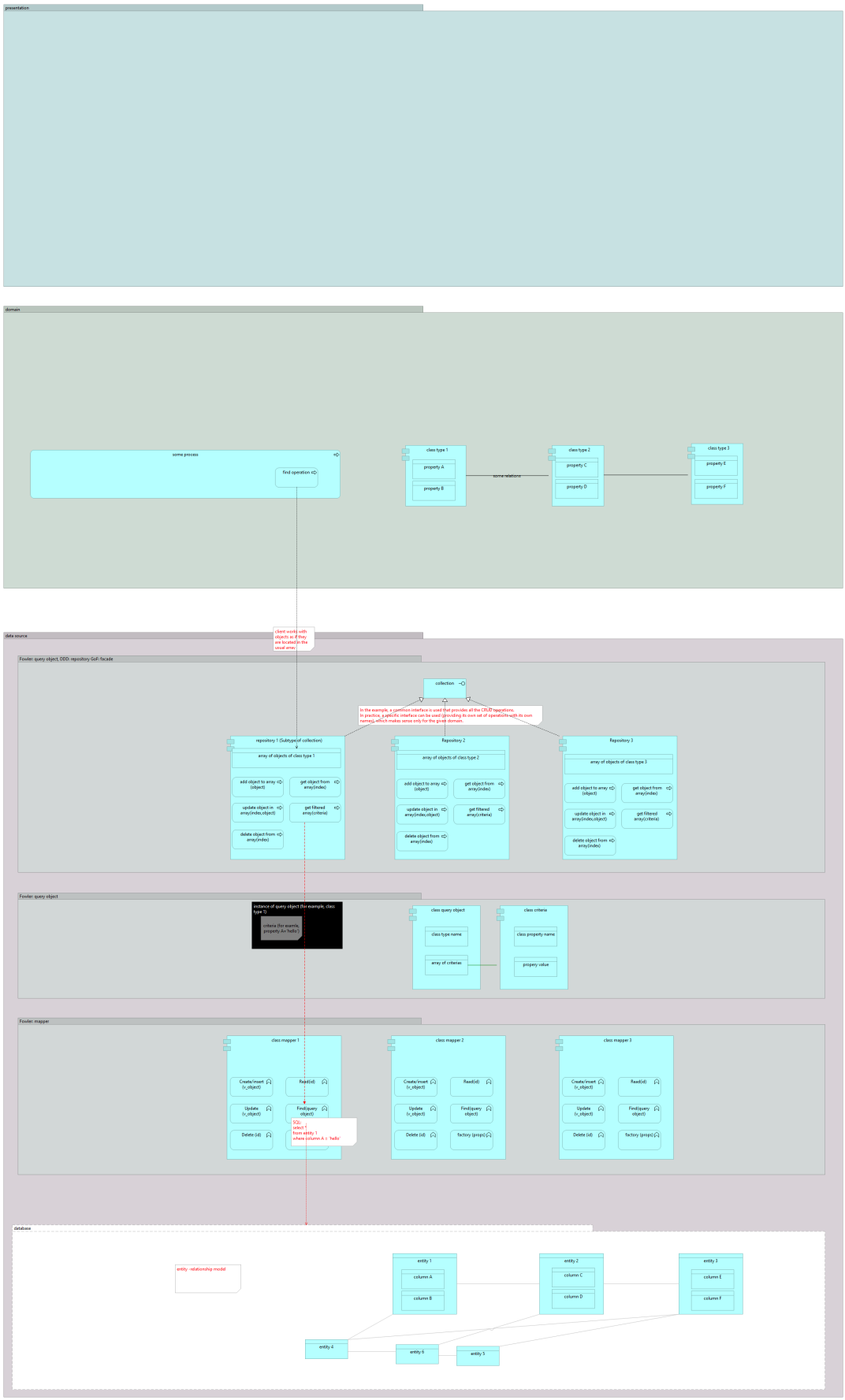
METADATA MAPPING



QUERY OBJECT



REPOSITORY

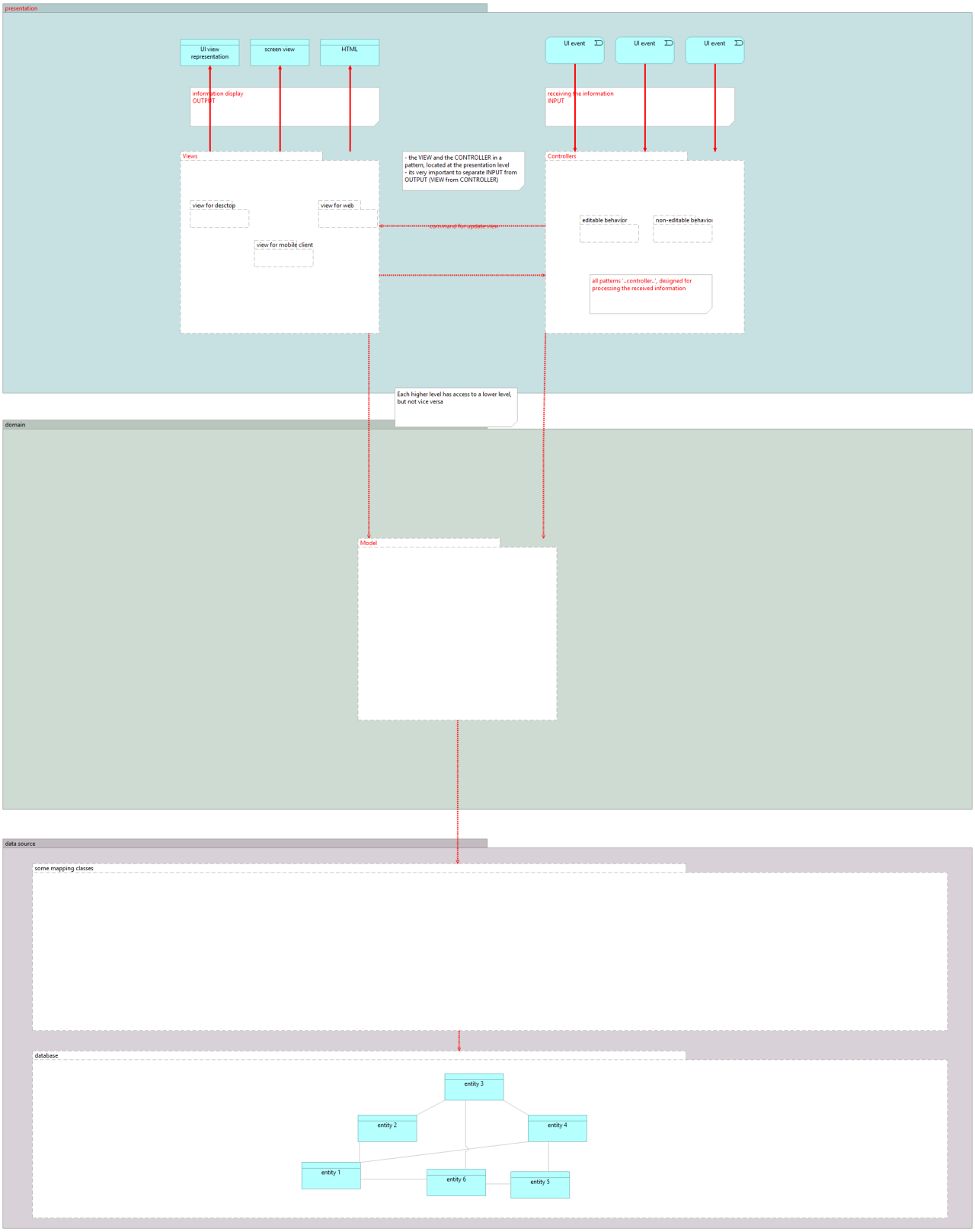

presentation
domain
some process
find operation
class type 1
property A
property B
some relations
class type 2
property C
property D
class type 3
property E
property F
data source
client works with objects as if they are located in the usual array.
Fowler: query object, DDD: repository GoF: facade
collection
In the example, a common interface is used that provides all the CRUD operations.
In practice, a specific interface can be used (providing its own set of operations with its own names), which makes sense only for the given domain.
repository 1 (Subtype of collection)
array of objects of class type 1
add object to array (object)
get object from array(index)
update object in array(index,object)
get filtered array(criteria)
delete object from array(index)
Repository 2
array of objects of class type 2
add object to array (object)
get object from array(index)
update object in array(index,object)
get filtered array(criteria)
delete object from array(index)
Repository 3
array of objects of class type 3
add object to array (object)
get object from array(index)
update object in array(index,object)
get filtered array(criteria)
delete object from array(index)
Fowler: query object
instance of query object (for example, class type 1)
criteria (for example, property A= 'hello')
class query object
class type name
array of criterias
class criteria
class property name
propery value
Fowler: mapper
class mapper 1
Create/insert (v_object)
Read(id)
Update (v_object)
Find(query object)
Delete (id)
SQL:
select *
from entity 1
where column A = 'hello'
class mapper 2
Create/insert (v_object)
Read(id)
Update (v_object)
Find(query object)
Delete (id)
factory (props)
class mapper 3
Create/insert (v_object)
Read(id)
Update (v_object)
Find(query object)
Delete (id)
factory (props)
database
entity - relationship model
entity 1
column A
column B
entity 2
column C
column D
entity 3
column E
column F
entity 4
entity 6
entity 5

WEB REPRESENTATION CONTROLLER

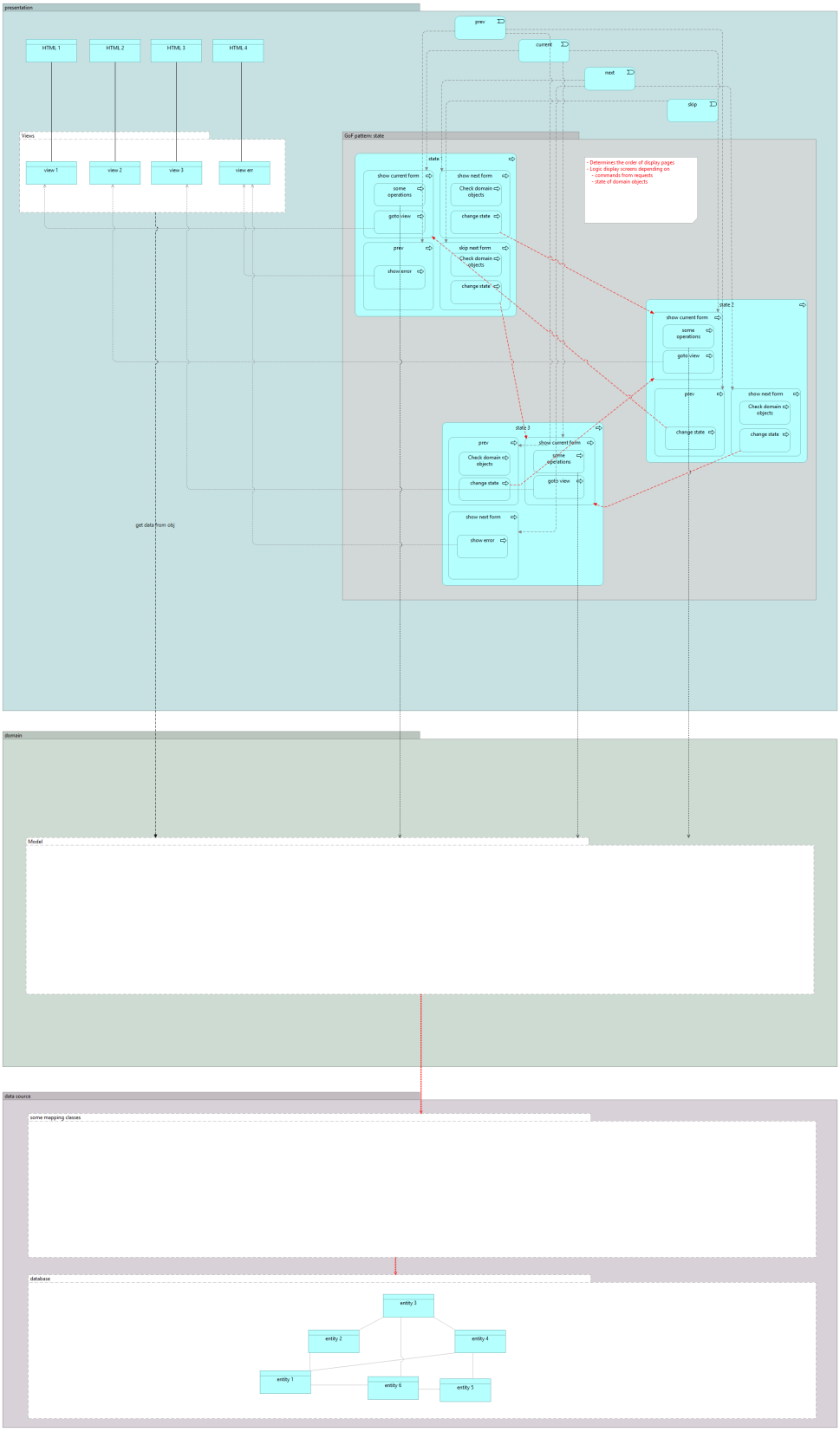
ENTERPRISE PATTERNS

Martin Fowler

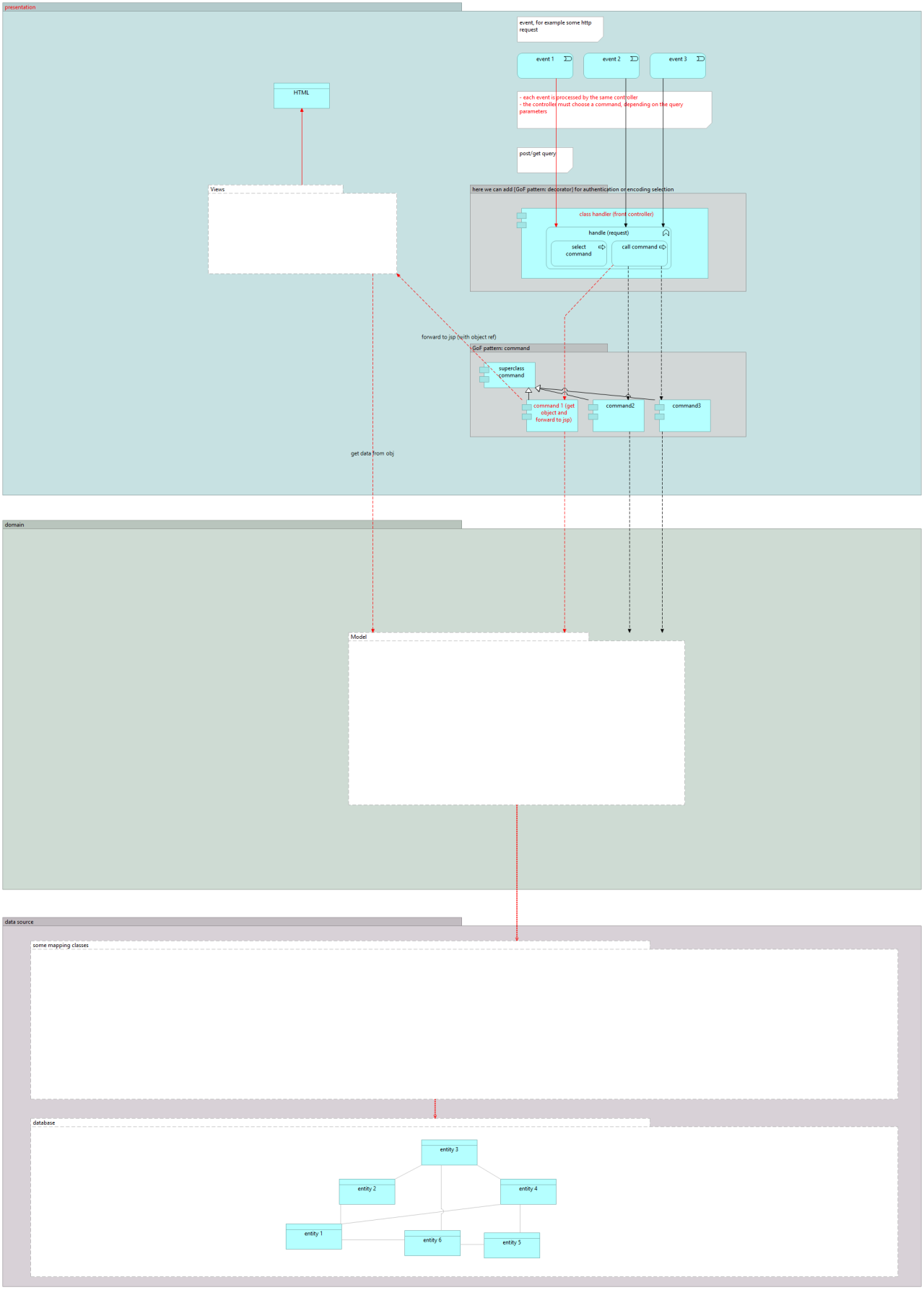
MODEL VIEW CONTROLLER



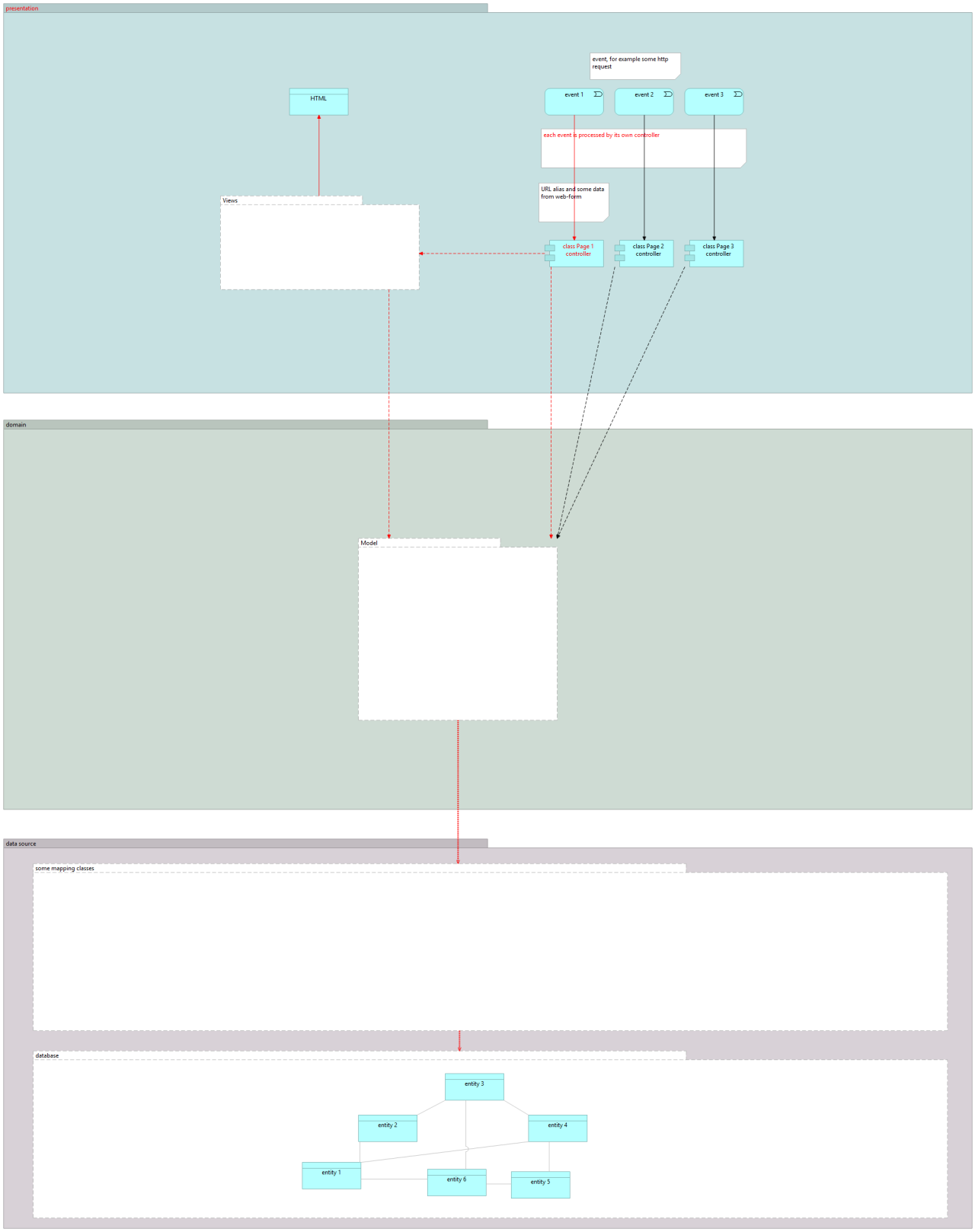
APPLICATION CONTROLLER



FRONT CONTROLLER



PAGE CONTROLLER

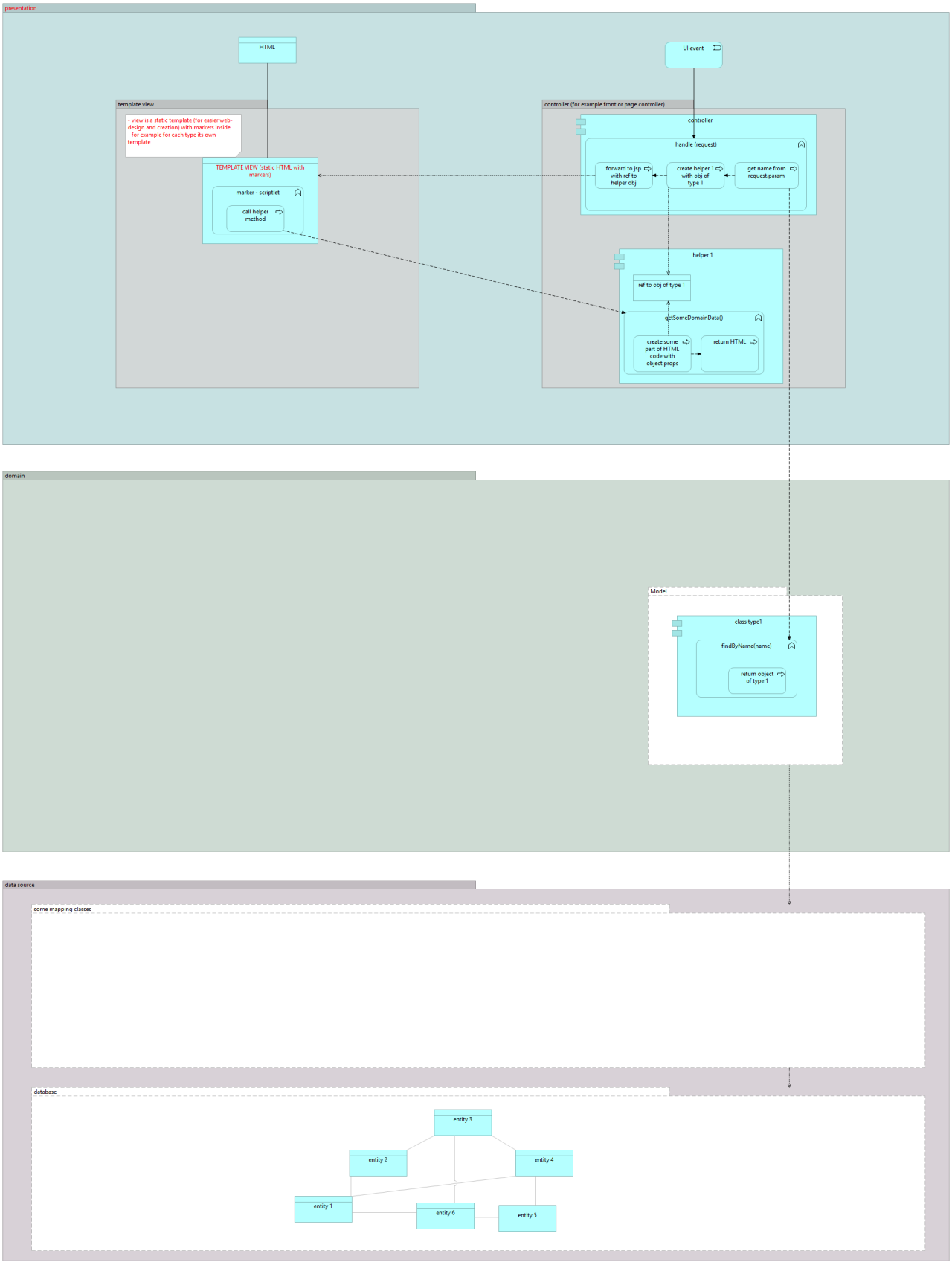


WEB REPRESENTATION VIEW

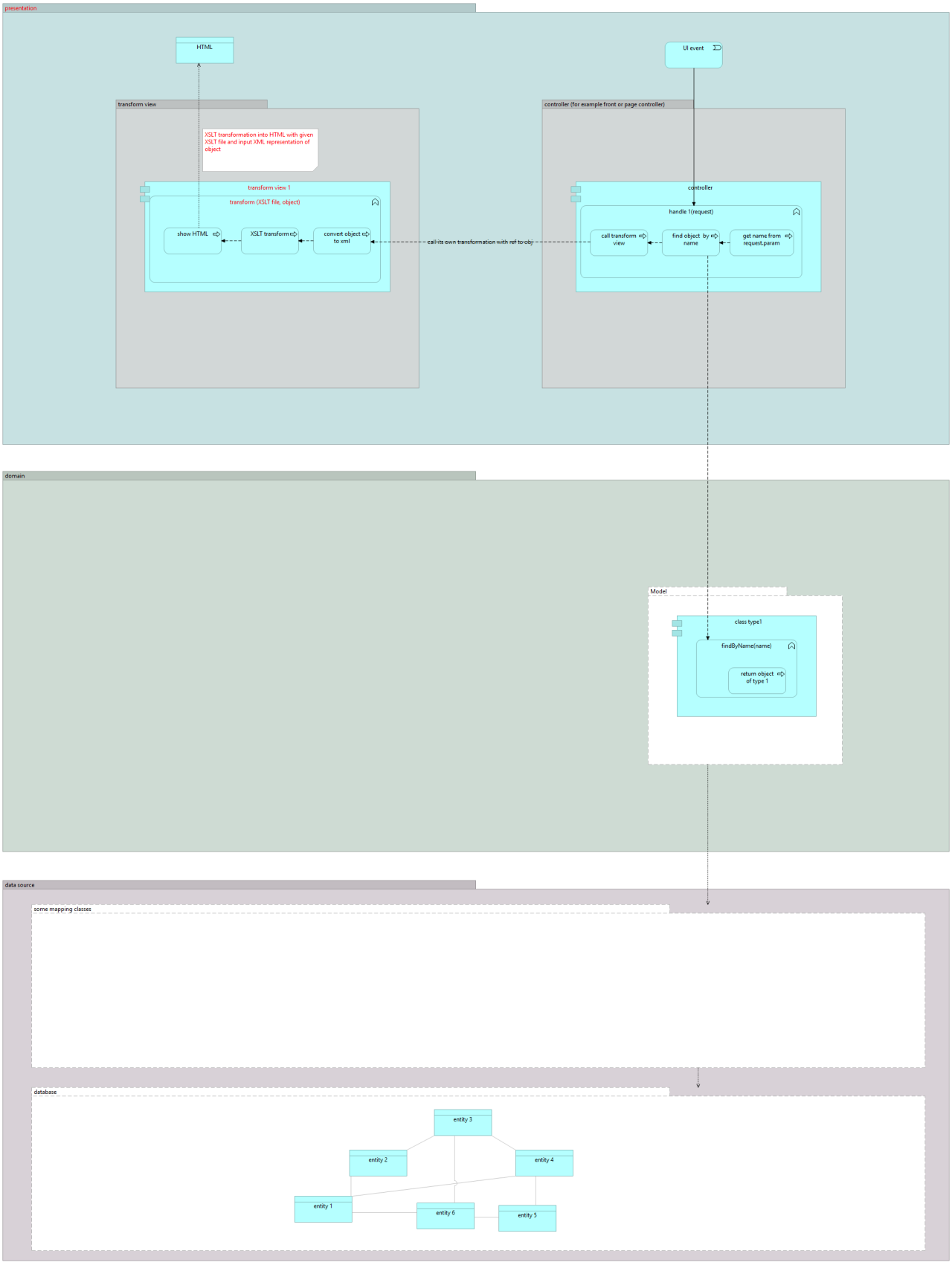
ENTERPRISE PATTERNS

Martin Fowler

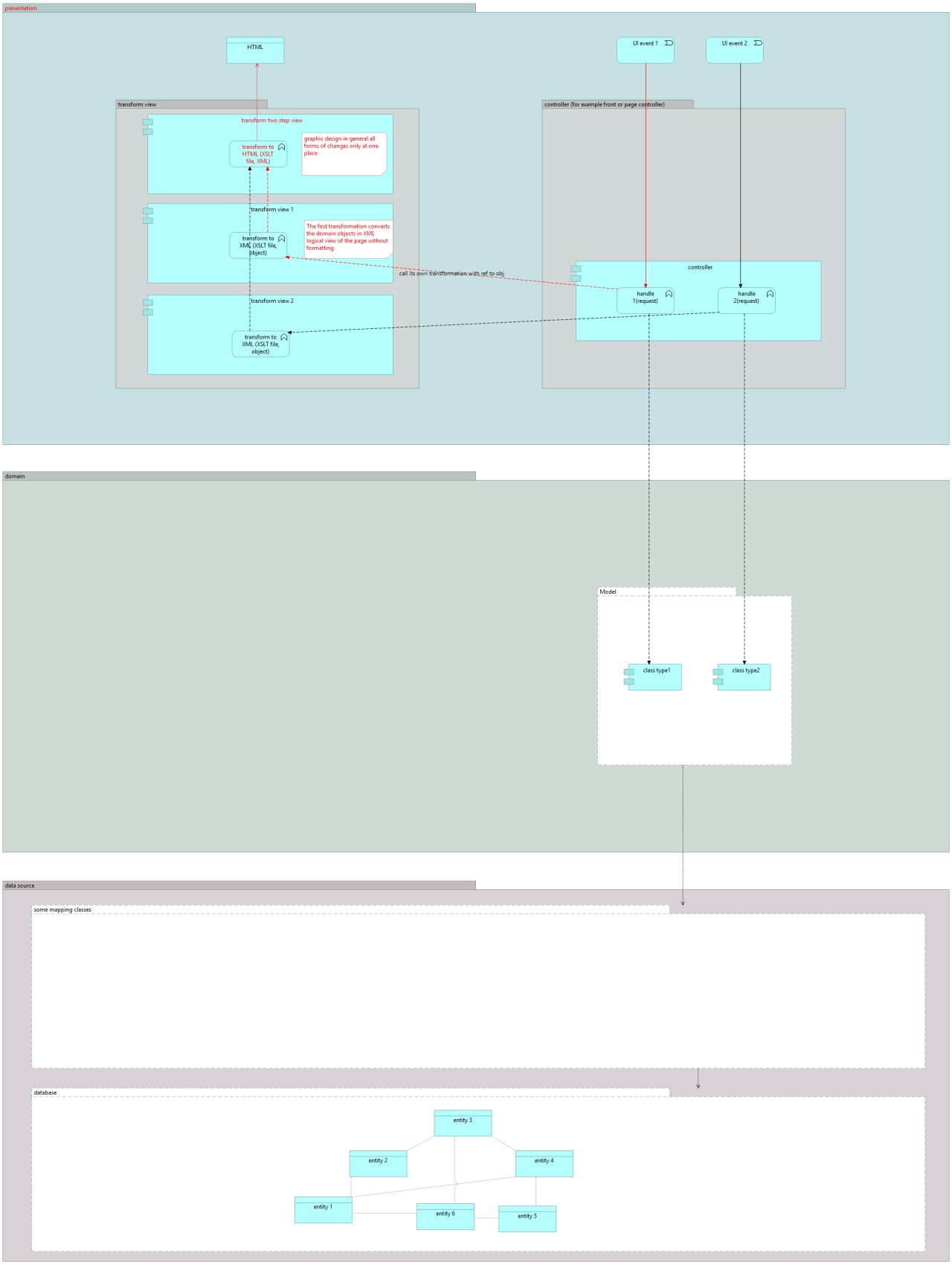
TEMPLATE VIEW



TRANSFORM VIEW



TWO STEP VIEW

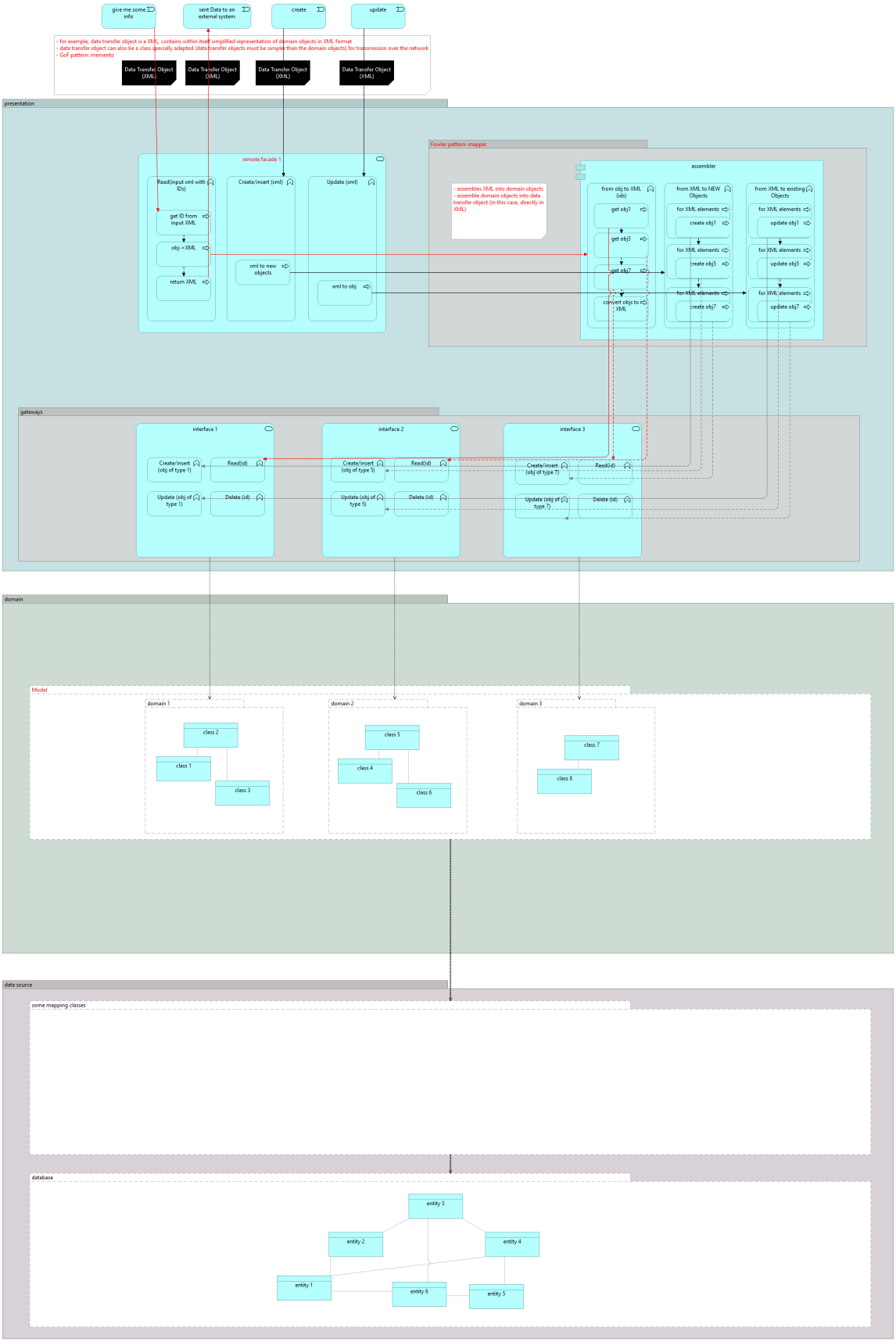


DISTRIBUTED PROCESSING

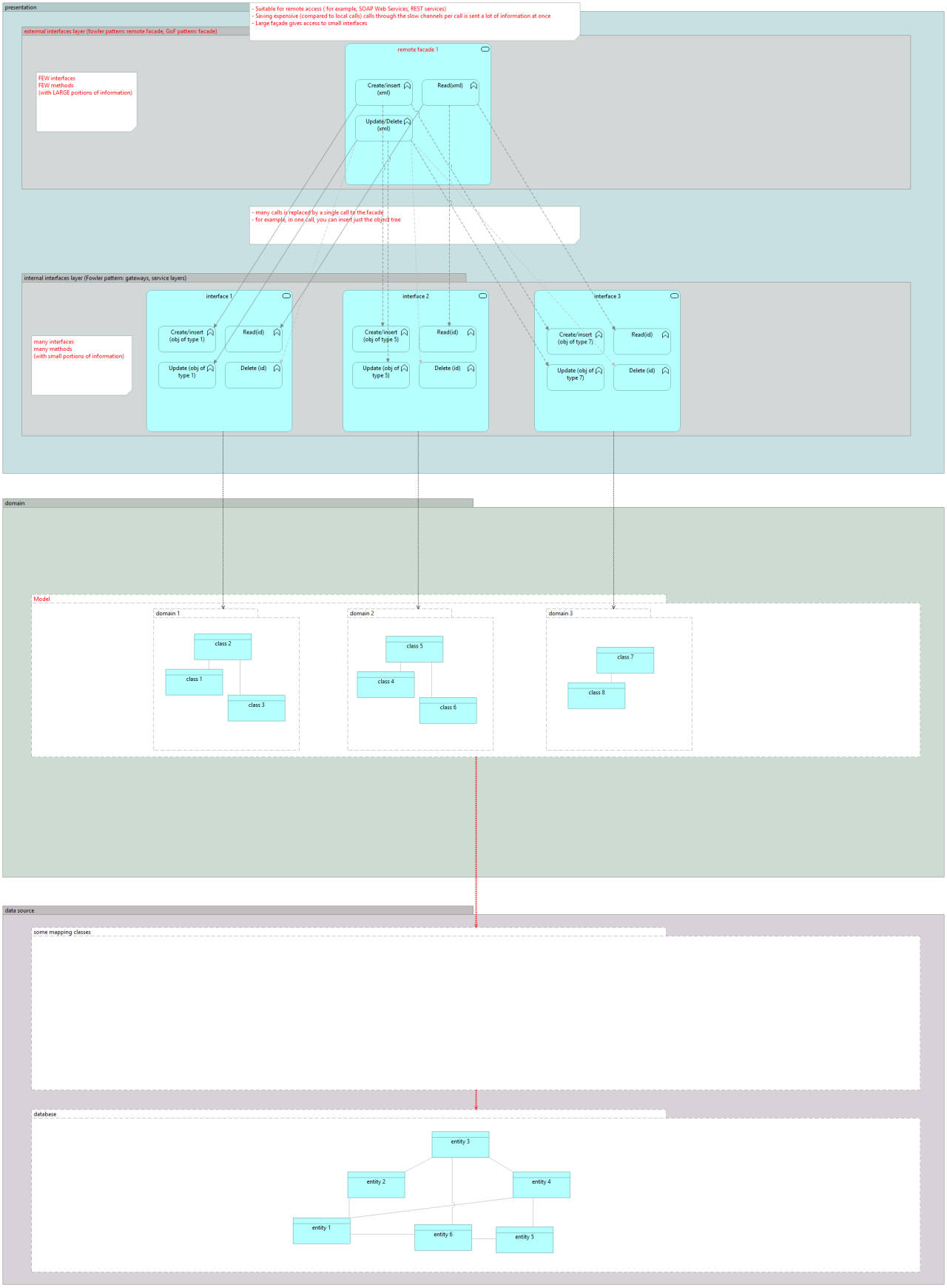
ENTERPRISE PATTERNS

Martin Fowler

DATA TRANSFER OBJECT



REMOTE FAÇADE

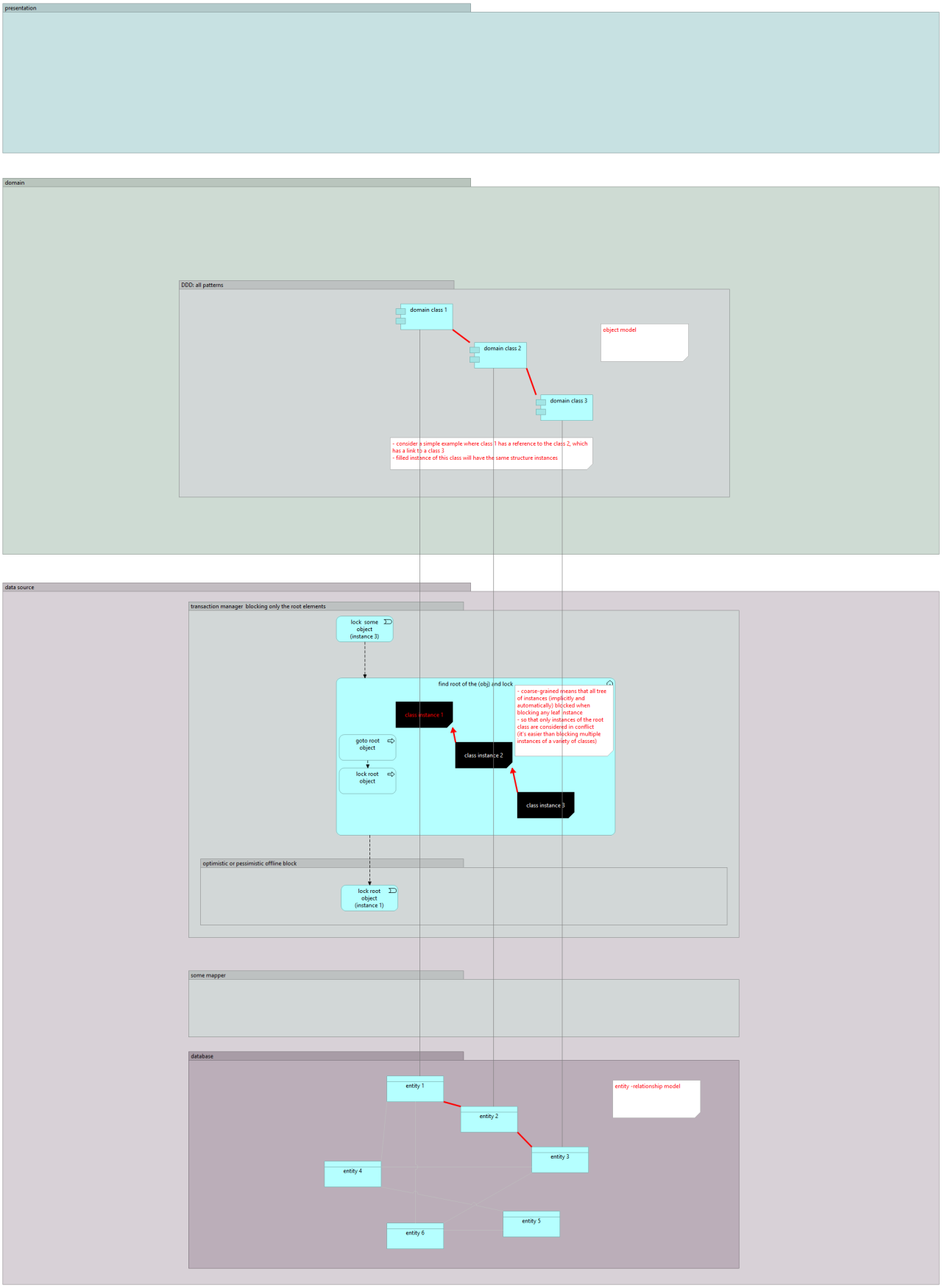


PARALLEL PROCESSING

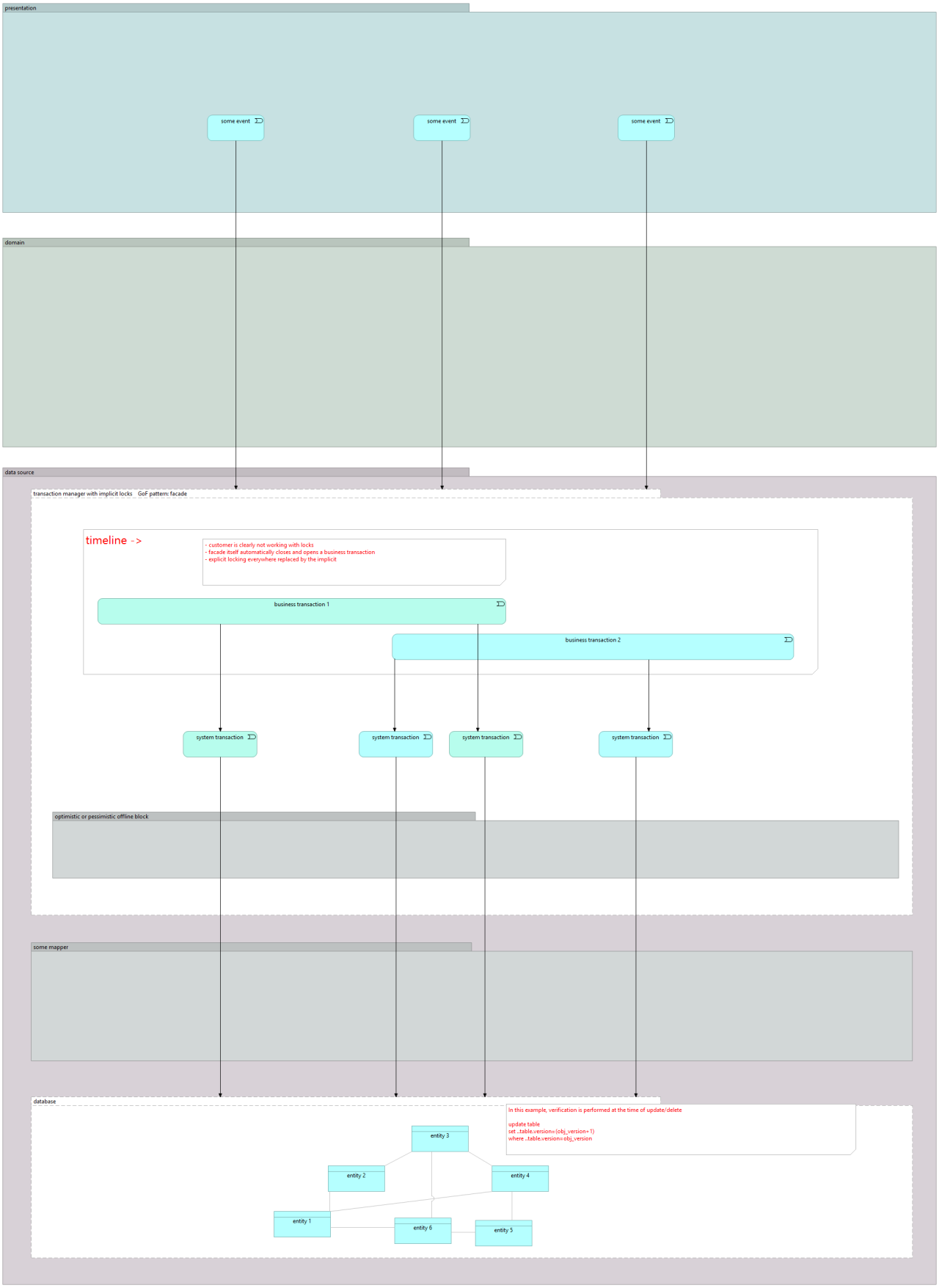
ENTERPRISE PATTERNS

Martin Fowler

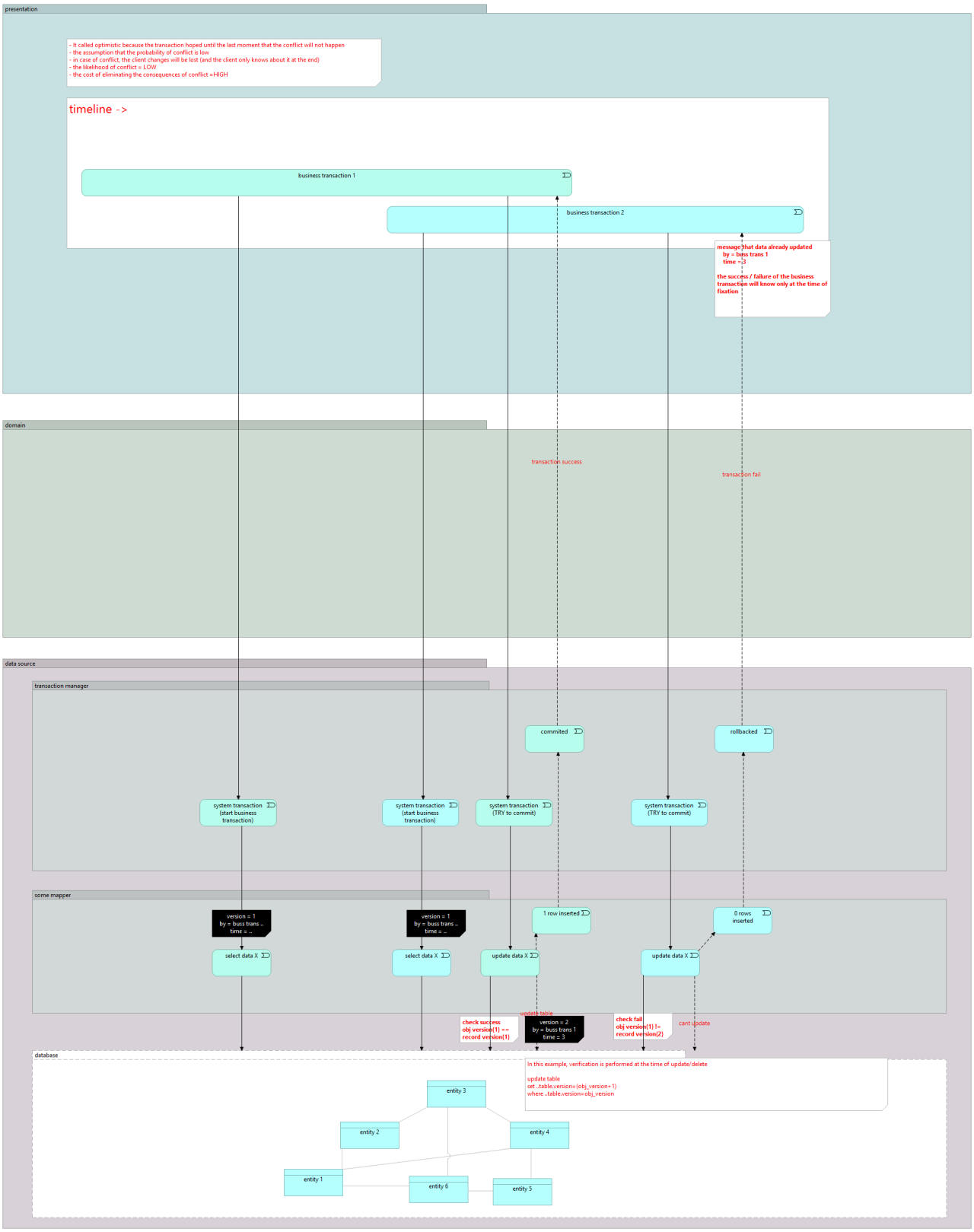
COARSE-GRAINED LOCK



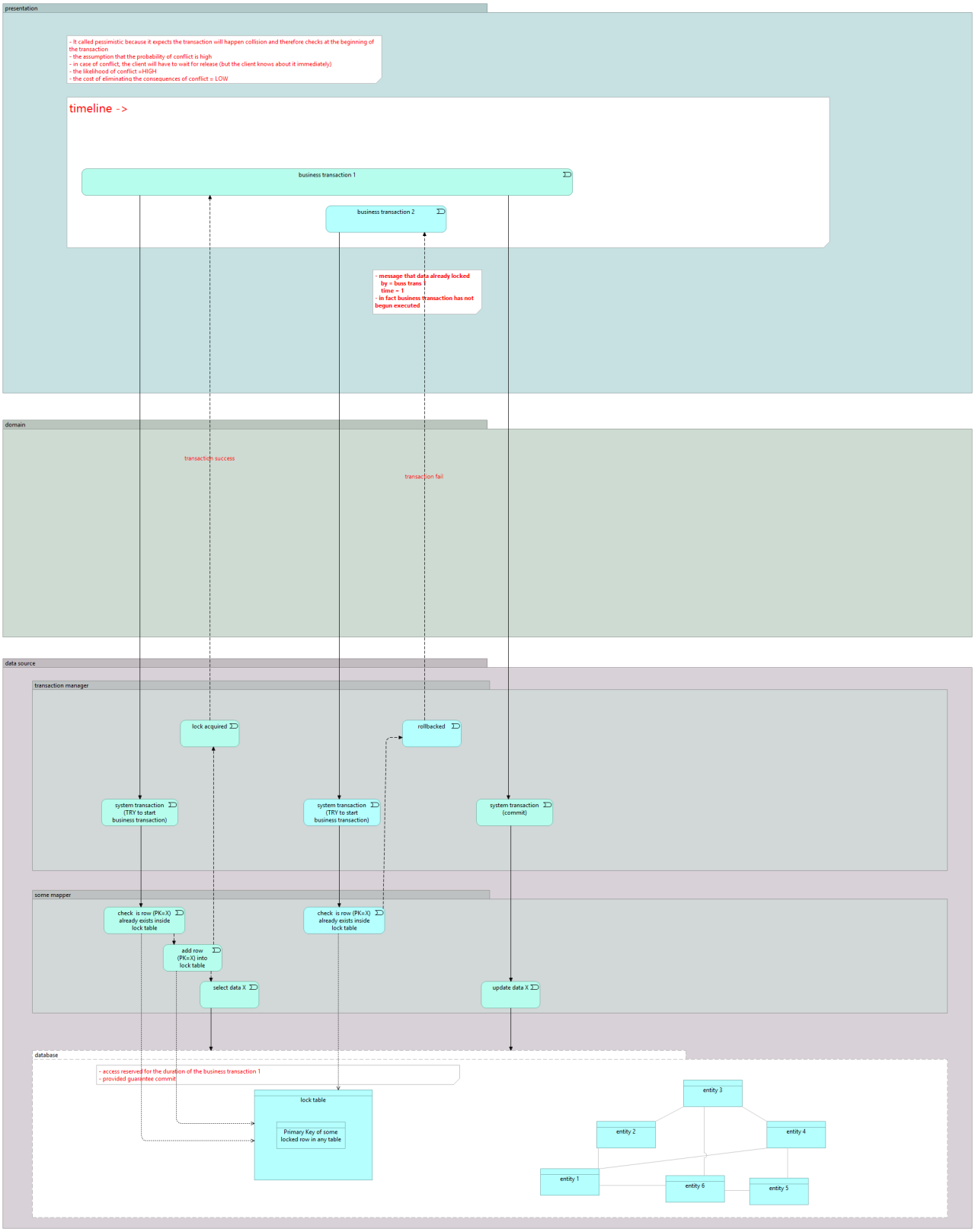
IMPLICIT LOCK



OPTIMISTIC OFFLINE LOCK



PESSIMISTIC OFFLINE LOCK

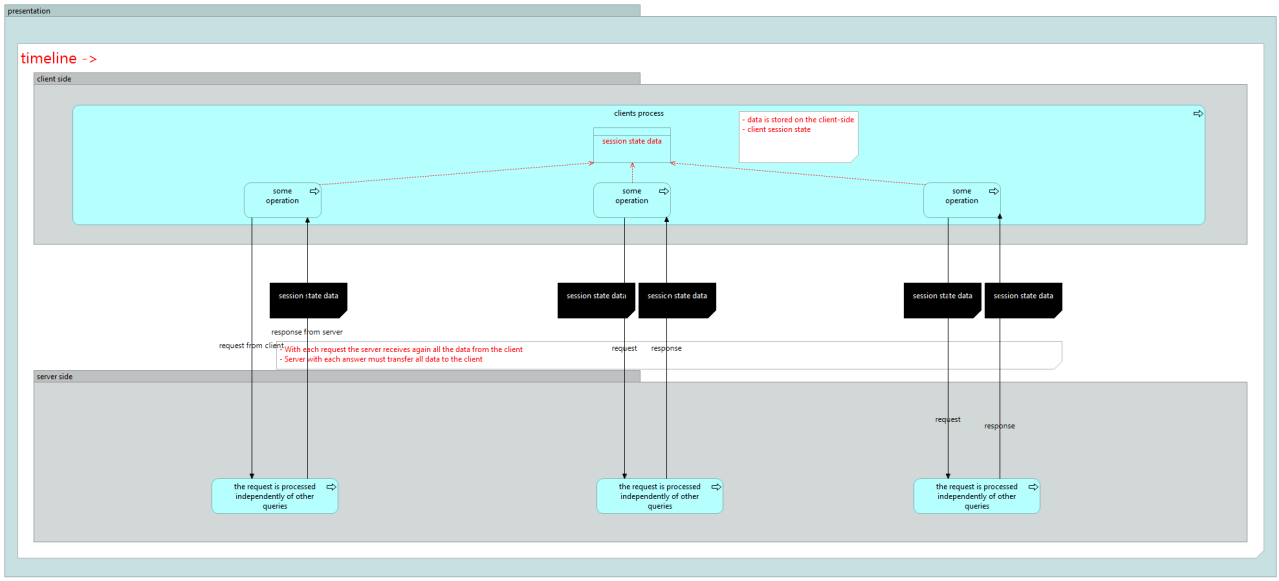


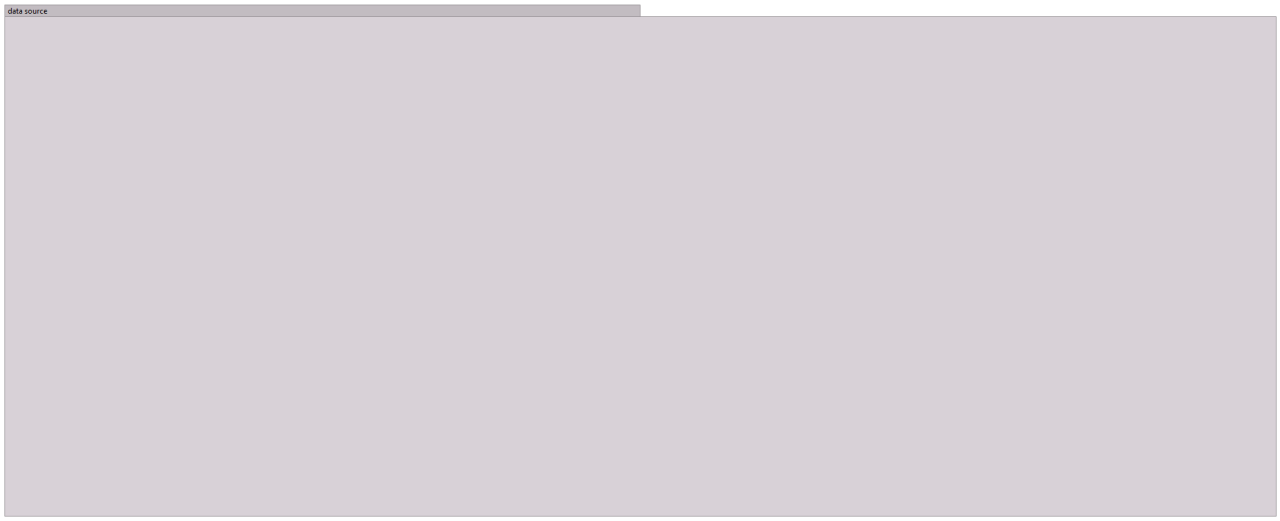
SESSION STATE

ENTERPRISE PATTERNS

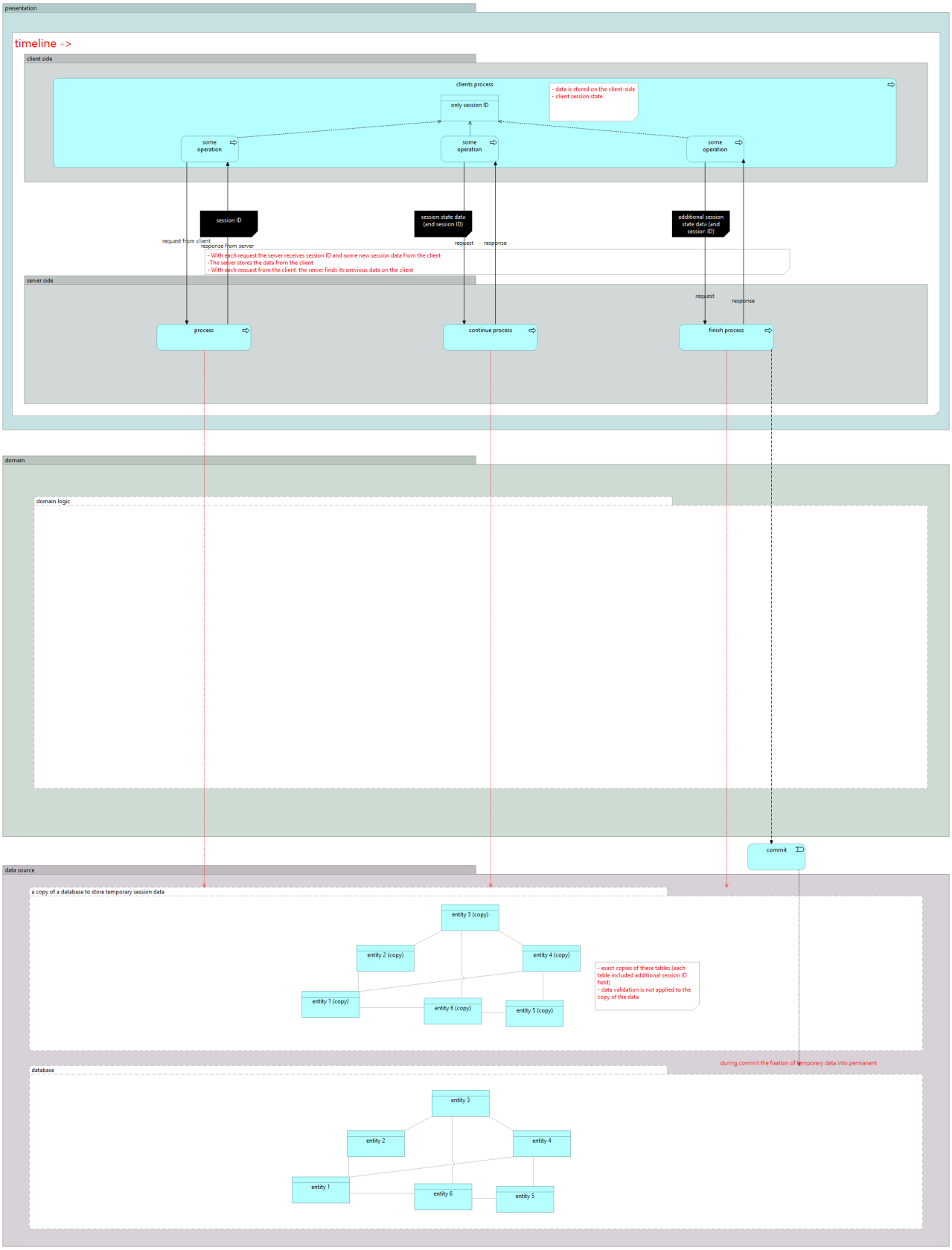
Martin Fowler

CLIENT SESSION STATE

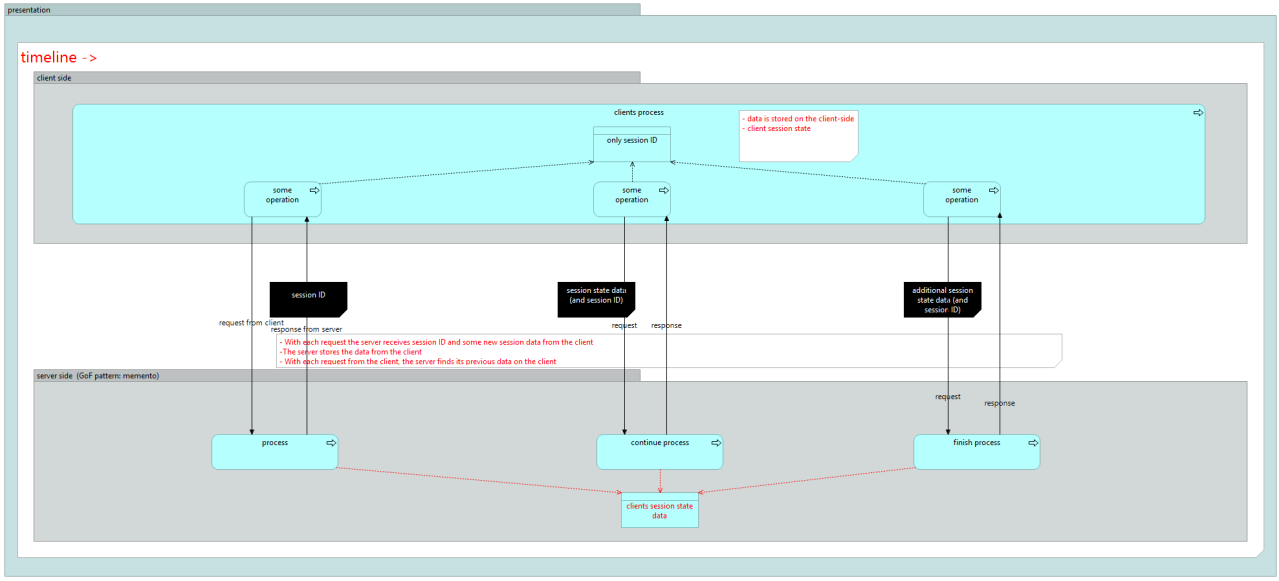


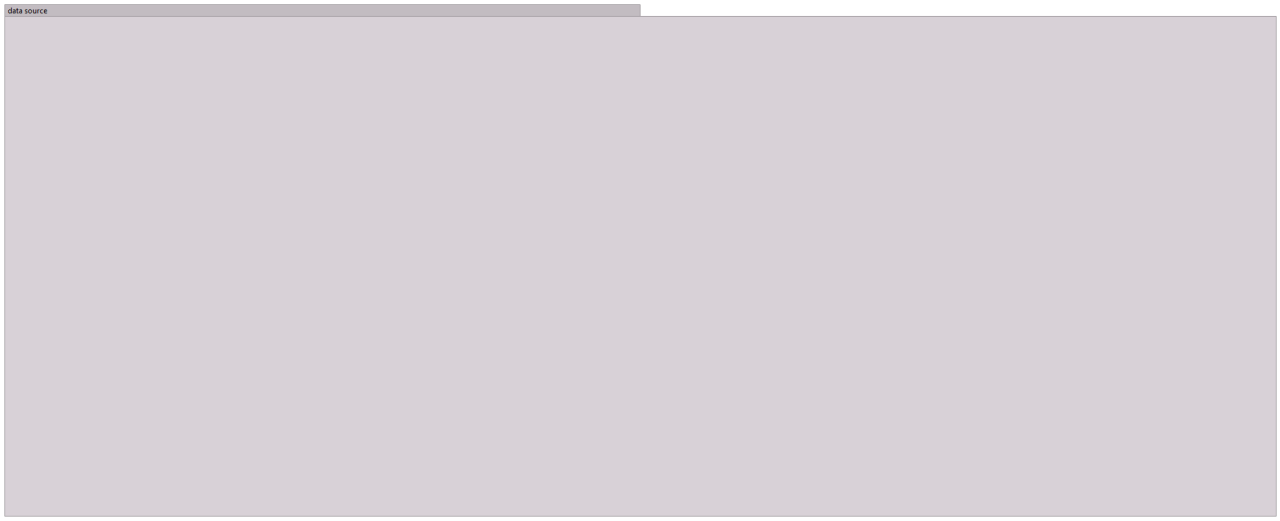


DATABASE SESSION STATE



SERVER SESSION STATE



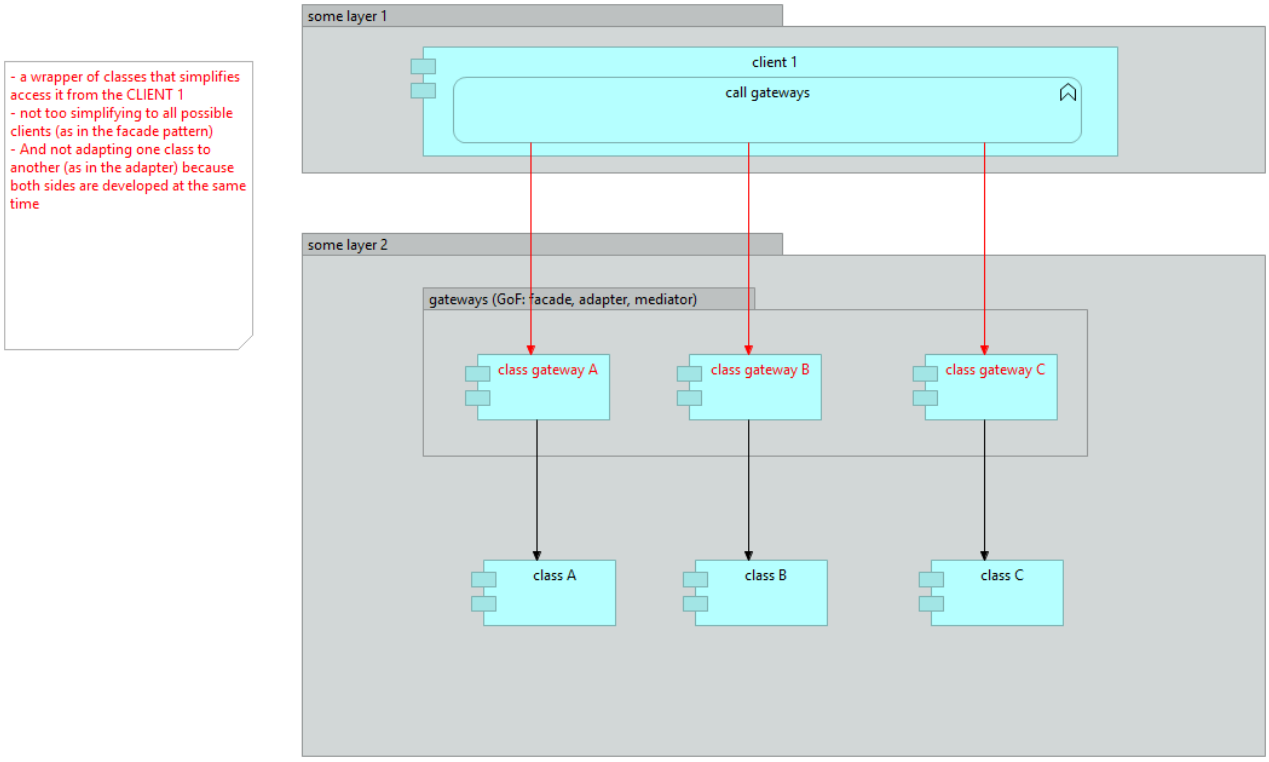


COMMON PATTERNS

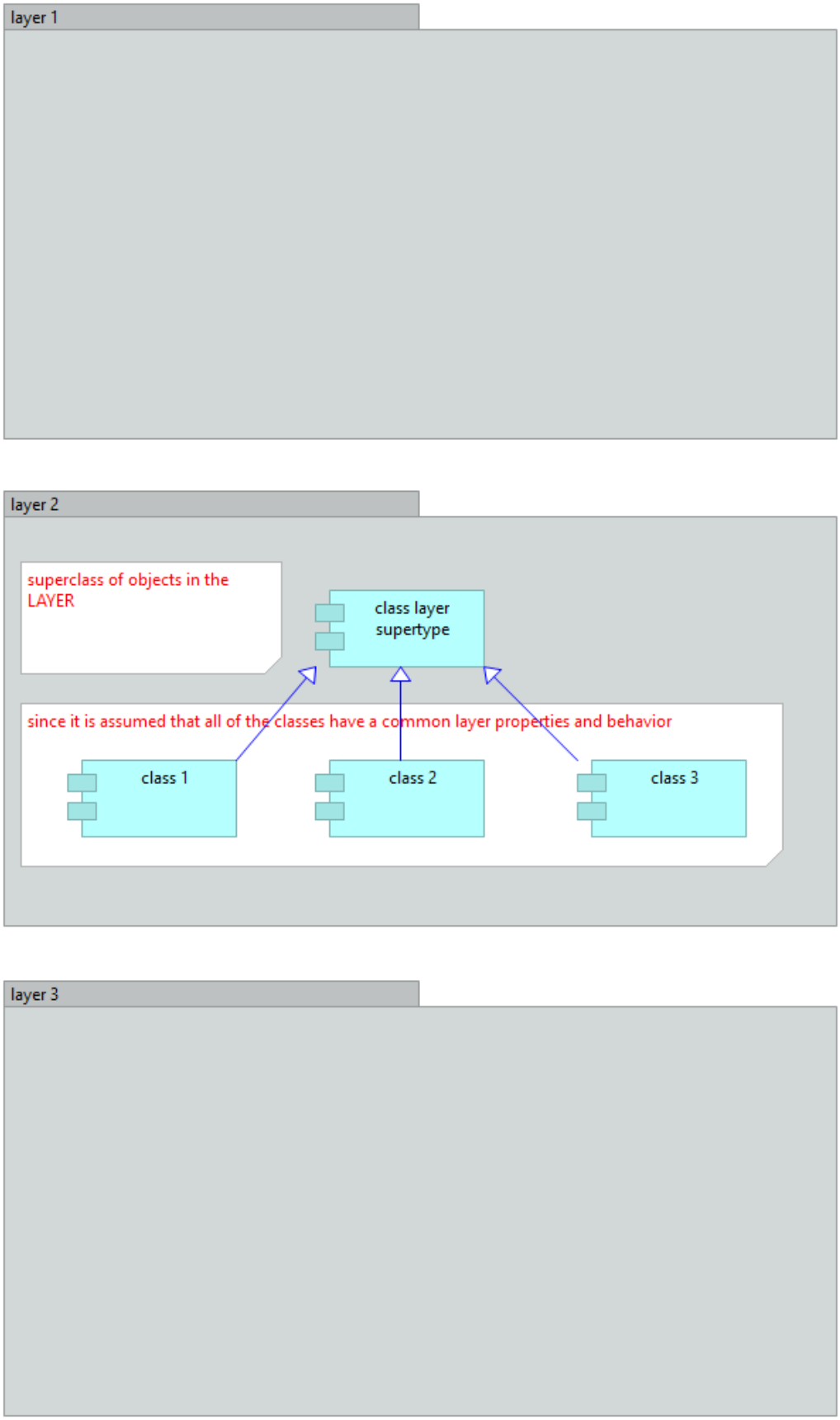
ENTERPRISE PATTERNS

Martin Fowler

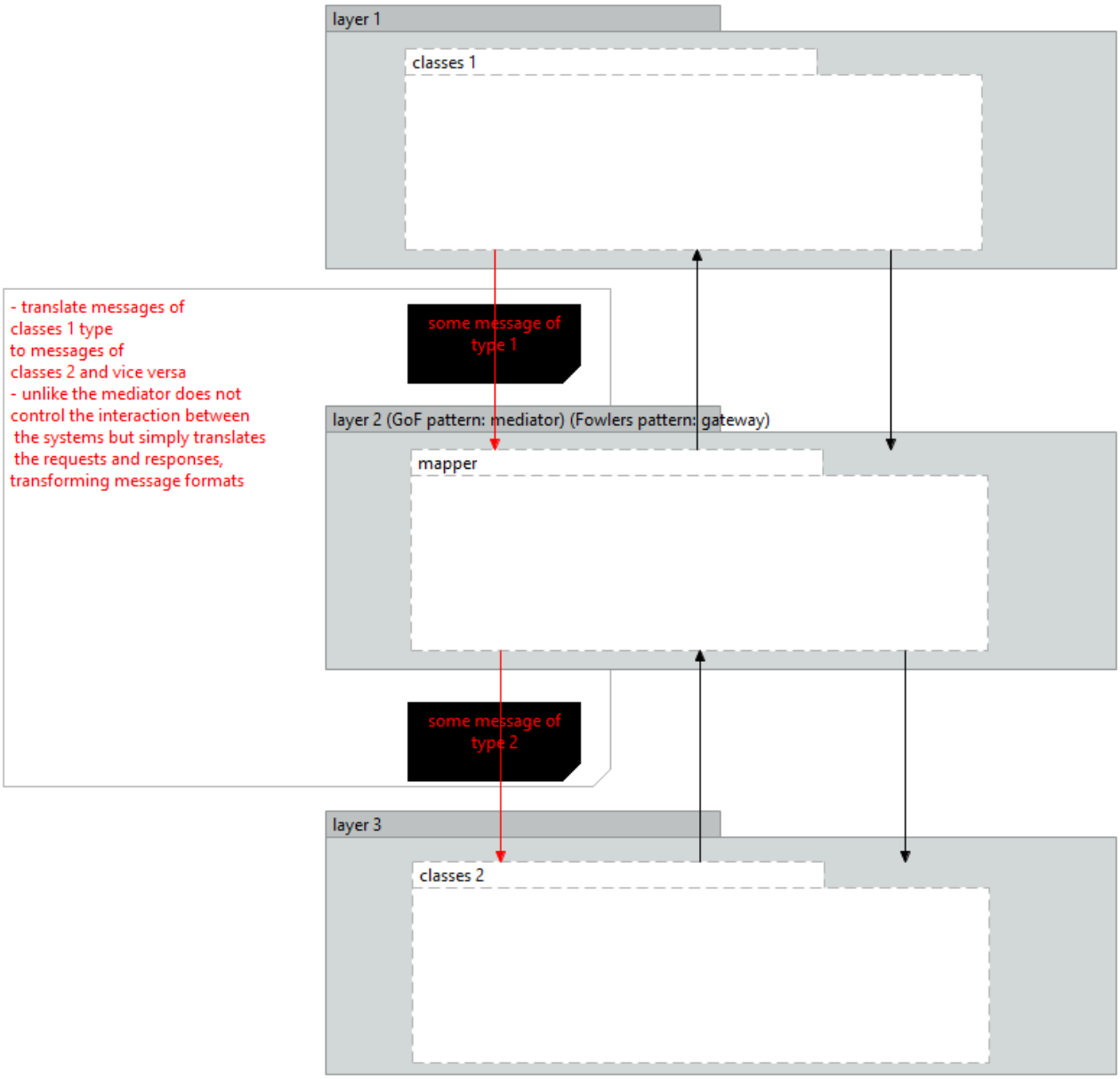
GATEWAY


- a wrapper of classes that simplifies access it from the CLIENT 1
- not too simplifying to all possible clients (as in the facade pattern)
- And not adapting one class to another (as in the adapter) because both sides are developed at the same time
some layer 1
client 1
call gateways
some layer 2
gateways (GoF: facade, adapter, mediator)
class gateway A
class gateway B
class gateway C
class A
class B
class C

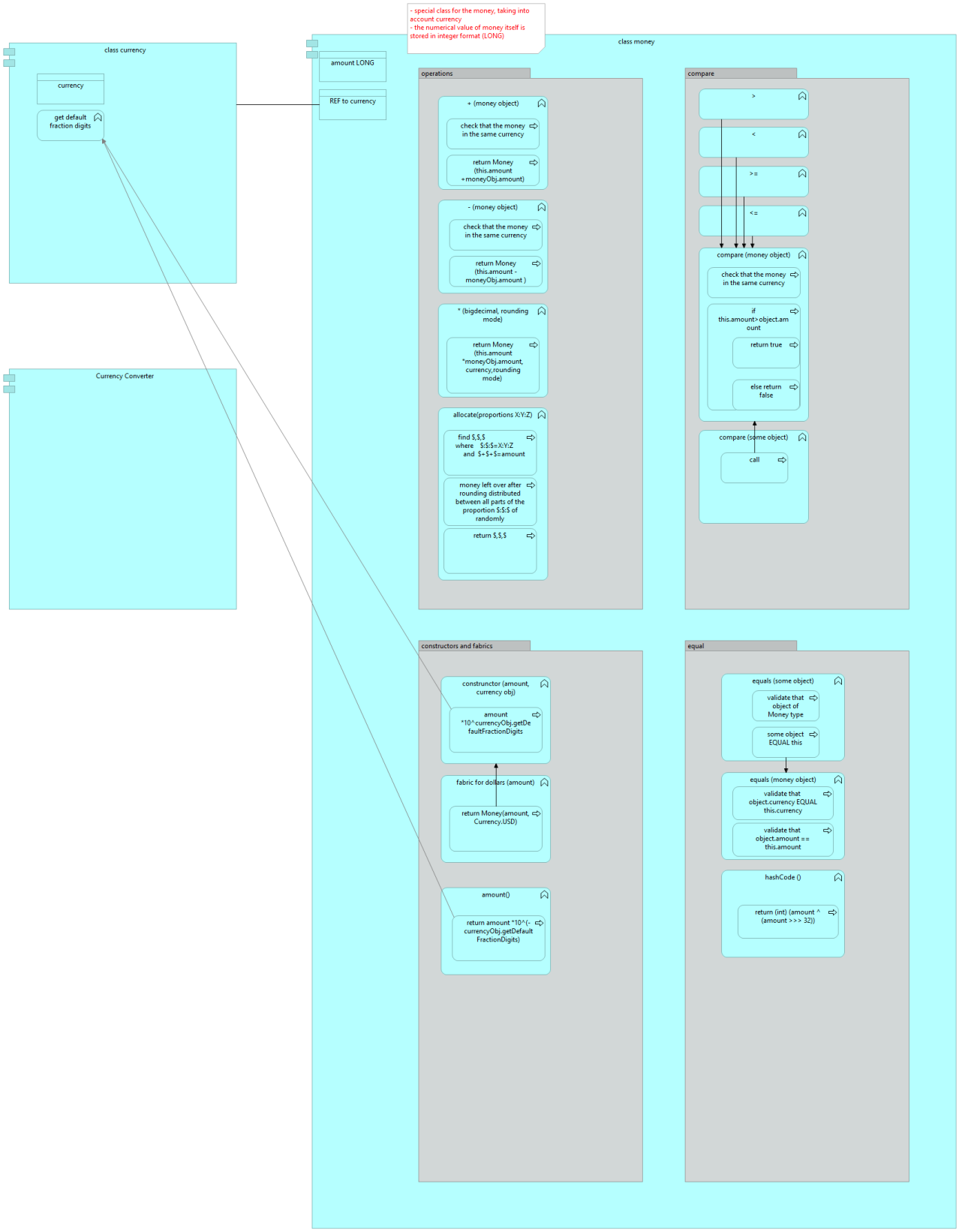
LAYER SUPERTYPE



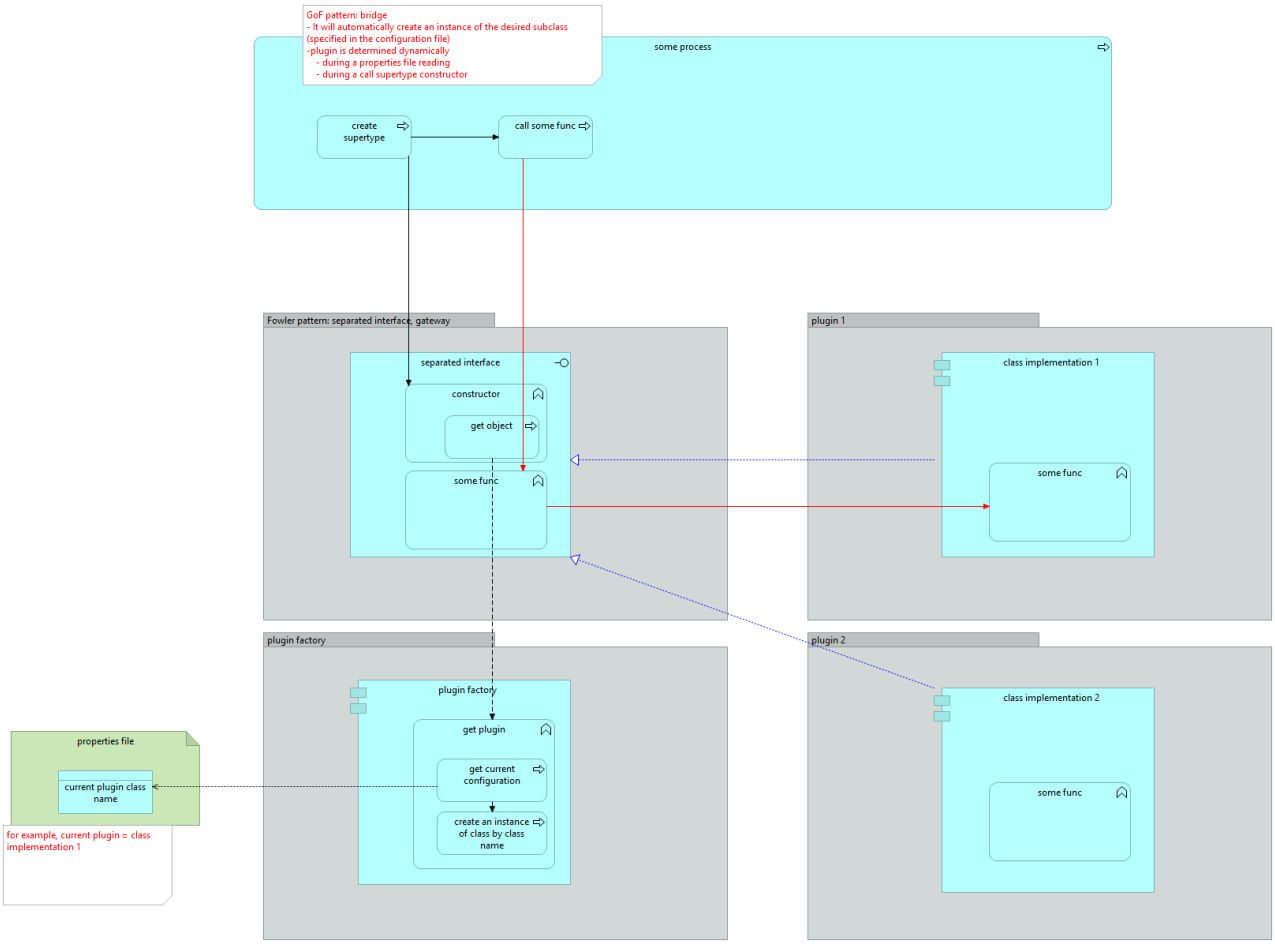
MAPPER


layer 1
classes 1
- translate messages of
classes 1 type
to messages of
classes 2 and vice versa
- unlike the mediator does not
control the interaction between
the systems but simply translates
the requests and responses,
transforming message formats
some message of
type 1
layer 2 (GoF pattern: mediator) (Fowlers pattern: gateway)
mapper
some message of
type 2
layer 3
classes 2

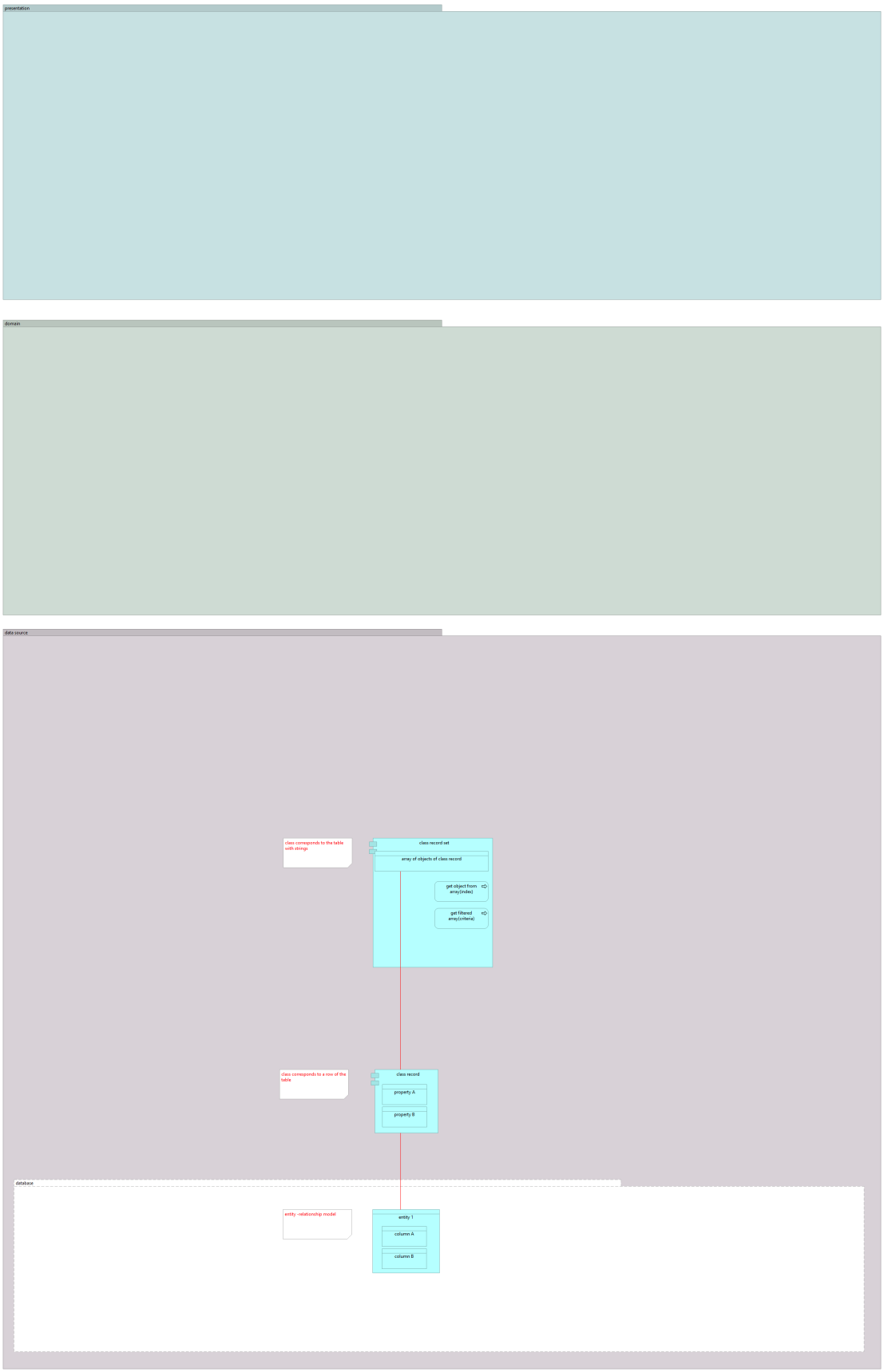
MONEY


- special class for the money, taking into account currency
- the numerical value of money itself is stored in integer format (LONG)
class currency
currency
get default fraction digits
Currency Converter
class money
amount LONG
REF to currency
operations
+ (money object)
check that the money in the same currency
return Money (this.amount +moneyObj.amount)
- (money object)
check that the money in the same currency
return Money (this.amount - moneyObj.amount)
* (bigdecimal, rounding mode)
return Money (this.amount *moneyObj.amount, currency,rounding mode)
allocate(proportions X:Y:Z)
find $,$,$ where $:$:$=X:Y:Z and $+$+$=amount
money left over after rounding distributed between all parts of the proportion $:$:$ of randomly
return $,$,$
compare
>
<
>=
<=
compare (money object)
check that the money in the same currency
if this.amount>object.amount
return true
else return false
compare (some object)
call
constructors and fabrics
construnctor (amount, currency obj)
amount *10^currencyObj.getDefaultFractionDigits
fabric for dollars (amount)
return Money(amount, Currency.USD)
amount()
return amount *10^(-currencyObj.getDefaultFractionDigits)
equal
equals (some object)
validate that object of Money type
some object EQUAL this
equals (money object)
validate that object.currency EQUAL this.currency
validate that object.amount == this.amount
hashCode ()
return (int) (amount ^ (amount >>> 32))

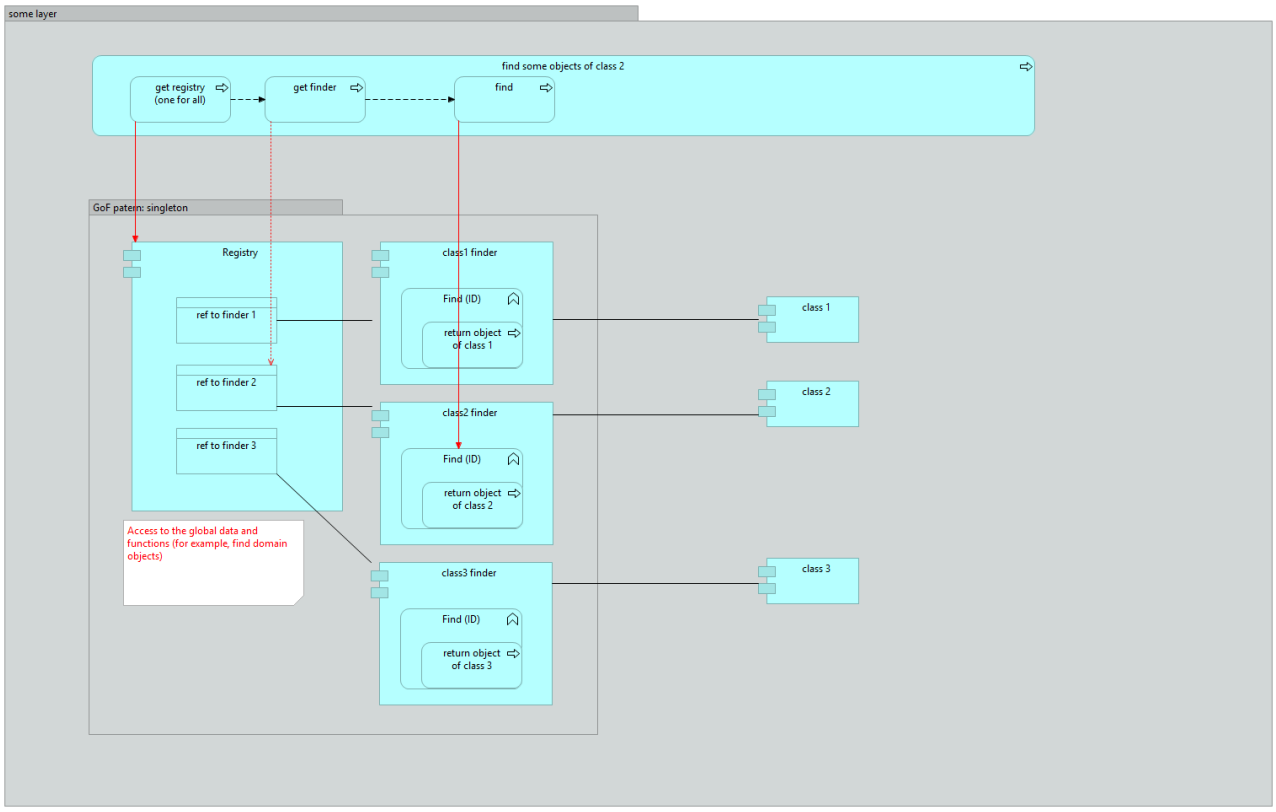
PLUGIN


GoF pattern: bridge
- It will automatically create an instance of the desired subclass (specified in the configuration file)
-plugin is determined dynamically
- during a properties file reading
- during a call supertype constructor
some process
create supertype
call some func
Fowler pattern: separated interface, gateway
separated interface
constructor
get object
some func
plugin 1
class implementation 1
some func
plugin factory
plugin factory
get plugin
get current configuration
create an instance of class by class name
plugin 2
class implementation 2
some func
properties file
current plugin class name
for example, current plugin = class implementation 1

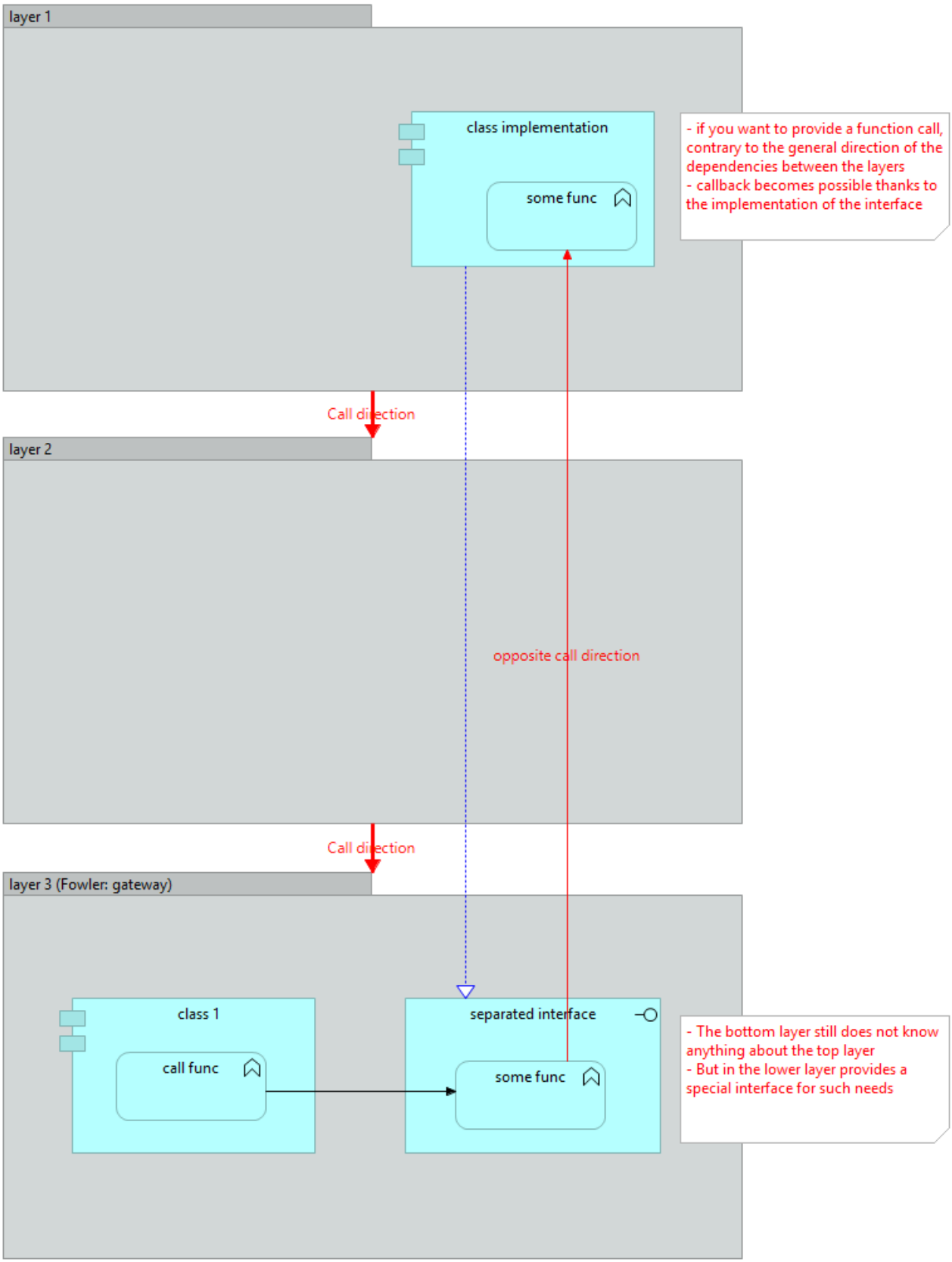
RECORD SET


presentation
domain
data source
class corresponds to the table with strings
class record set
array of objects of class record
get object from array(index)
get filtered array(criteria)
class corresponds to a row of the table
class record
property A
property B
database
entity -relationship model
entity 1
column A
column B

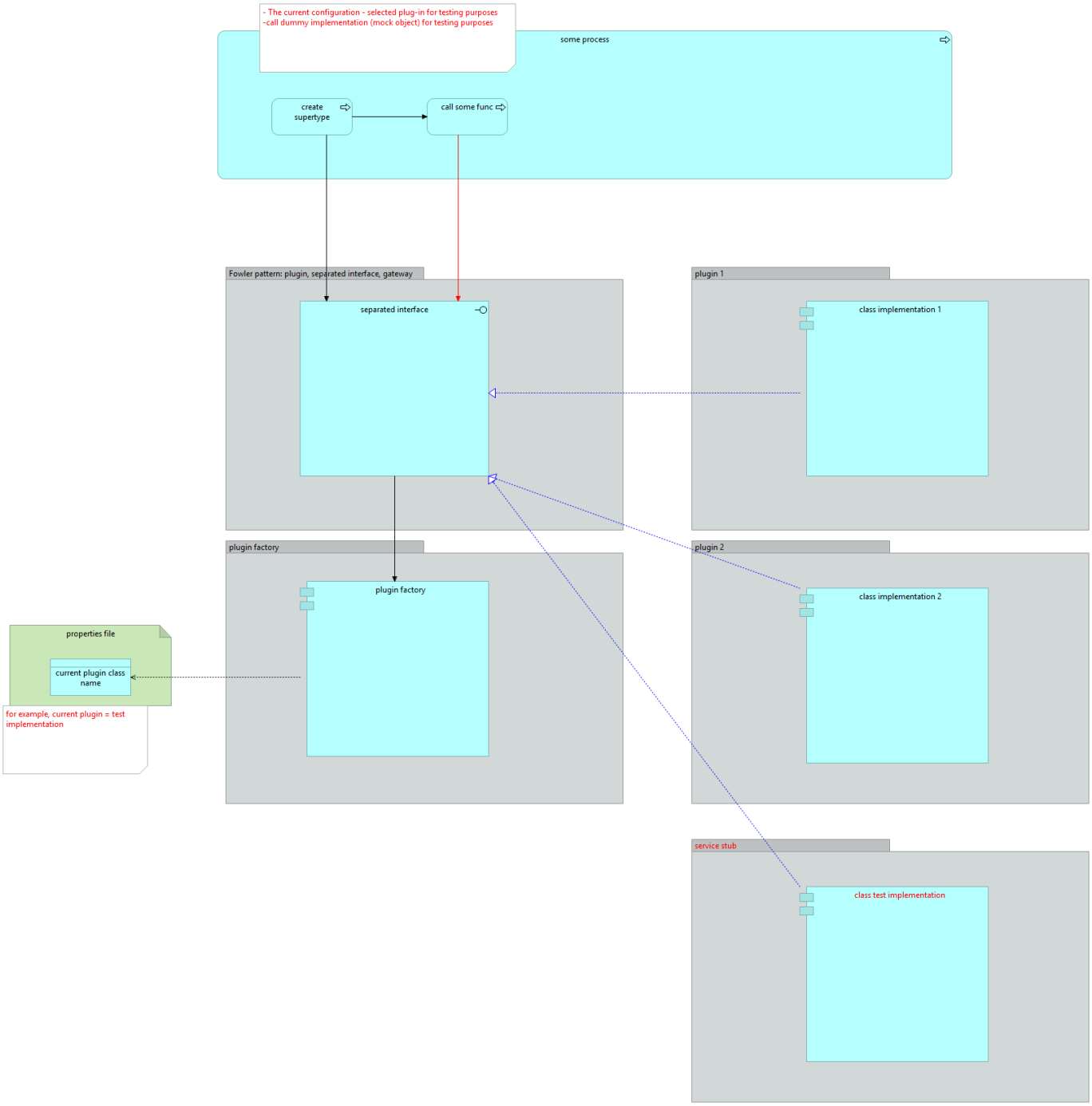
REGISTRY


some layer
find some objects of class 2
get registry
(one for all)
get finder
find
GoF patern: singleton
Registry
ref to finder 1
ref to finder 2
ref to finder 3
class1 finder
Find (ID)
return object
of class 1
class2 finder
Find (ID)
return object
of class 2
class3 finder
Find (ID)
return object
of class 3
class 1
class 2
class 3
Access to the global data and
functions (for example, find domain
objects)

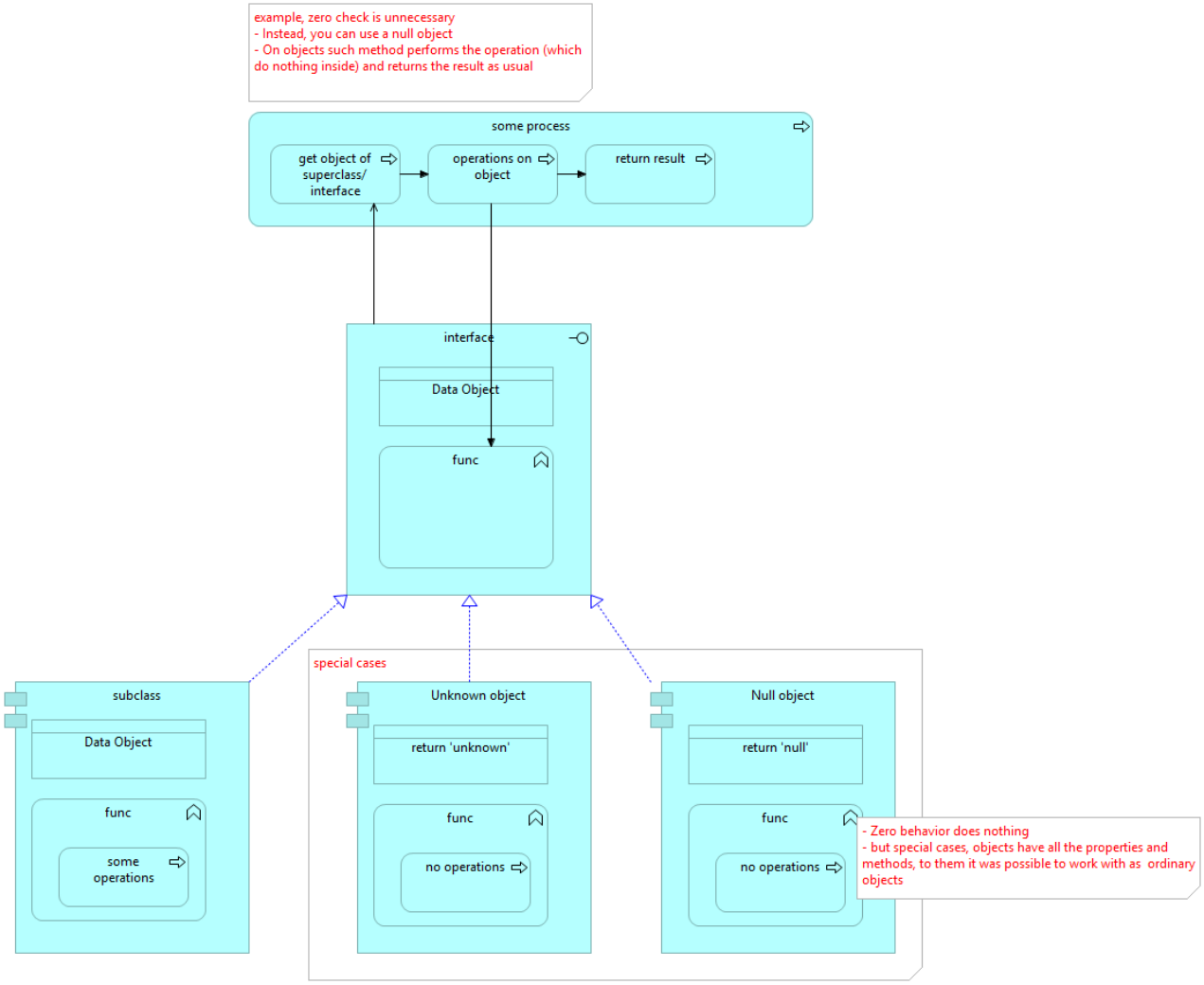
SEPARATED INTERFACE



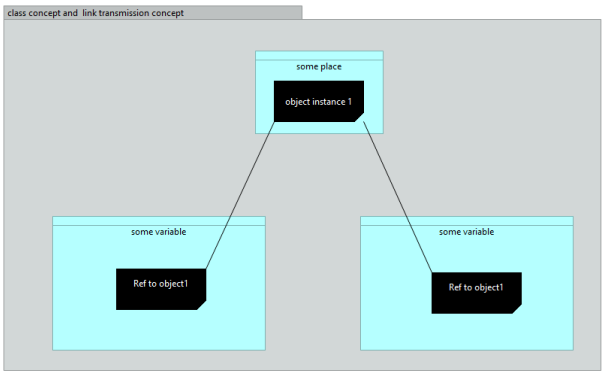
SERVICE STUB

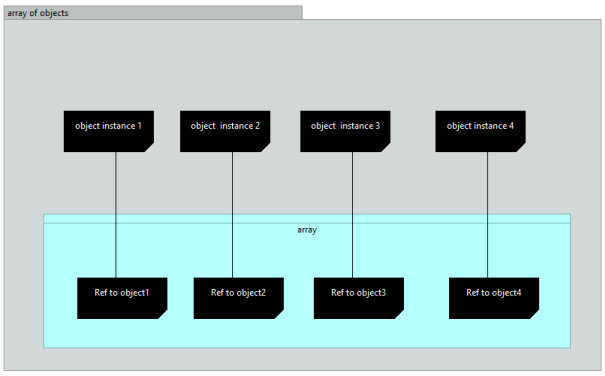


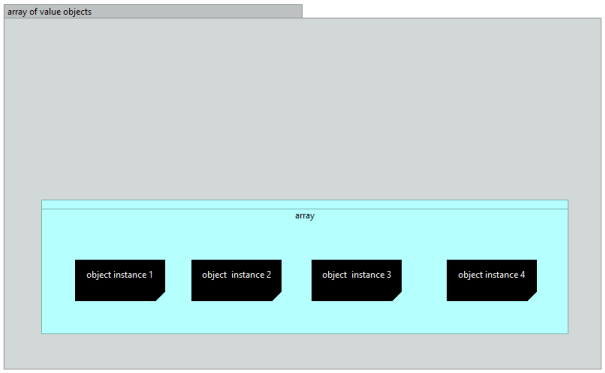
SPECIAL CASE



VALUE OBJECT







03

Analysis Patterns

Reusable Object Models

MARTIN FOWLER

1 Accountability

2 Observations and Measurements

3 Observations for Corporate Finance

4 Referring to Objects

5 Inventory and Accounting

6 Planning

7 Trading

8 Derivative Contracts

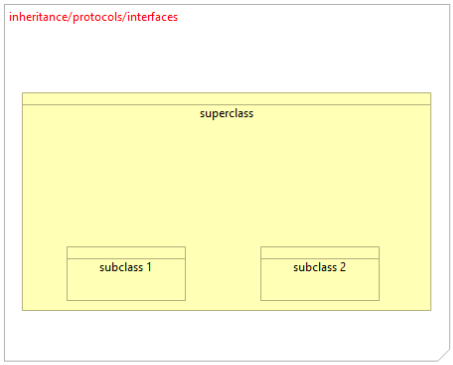
9 Trading Packages

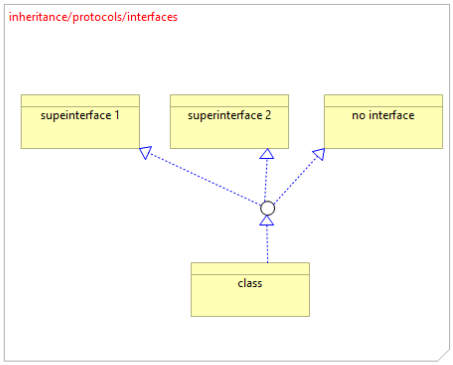
10 Layered Architecture for Information Systems

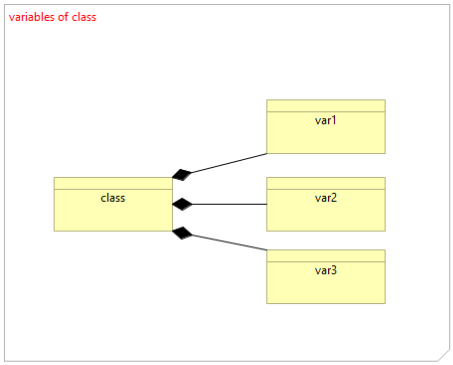
11 Patterns for Type Model Design

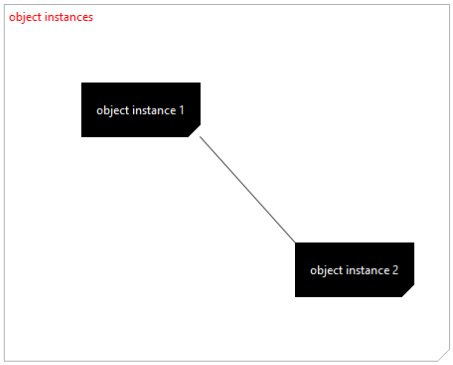
12 Association Patterns

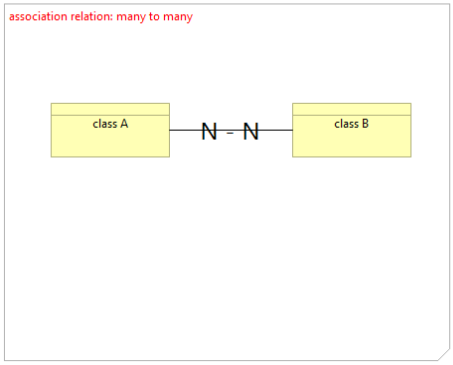
USED NOTATION

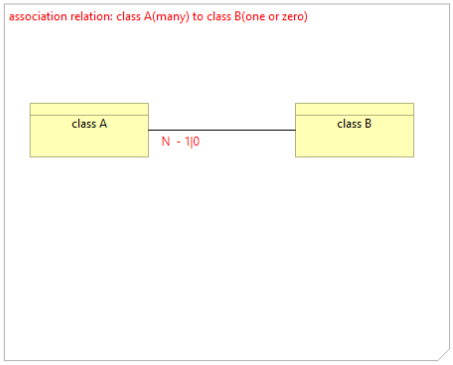


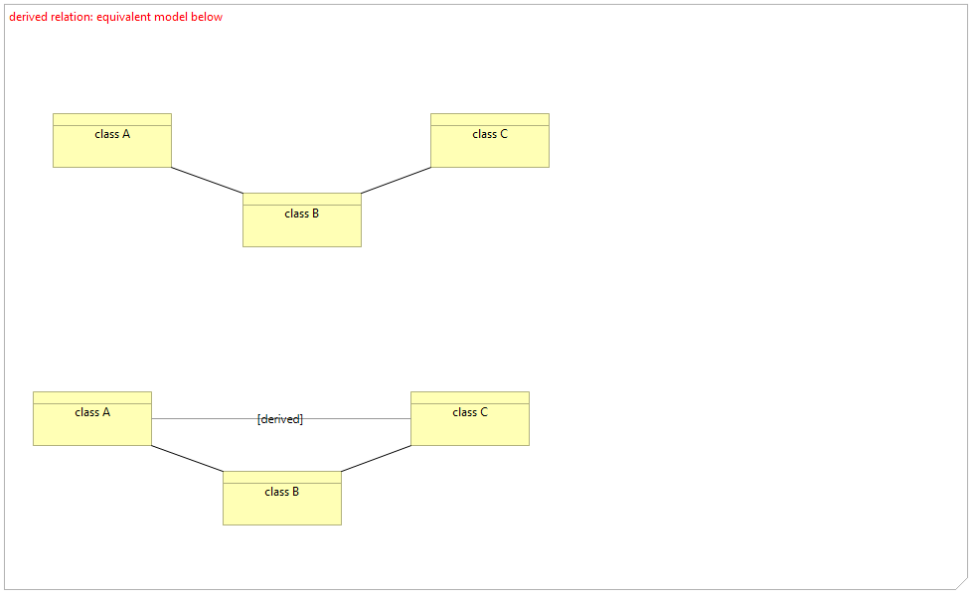








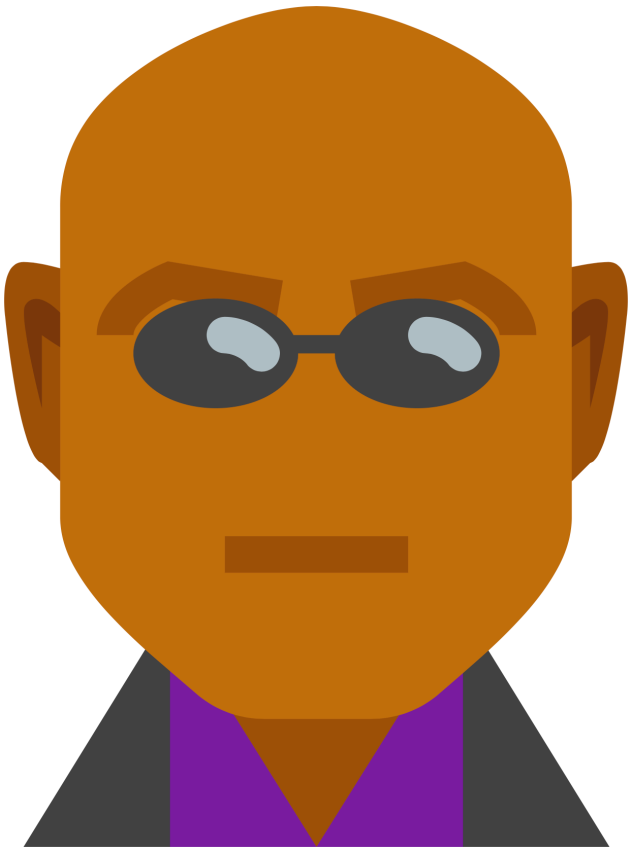




ACCOUNTABILITY

ANALYSIS PATTERNS

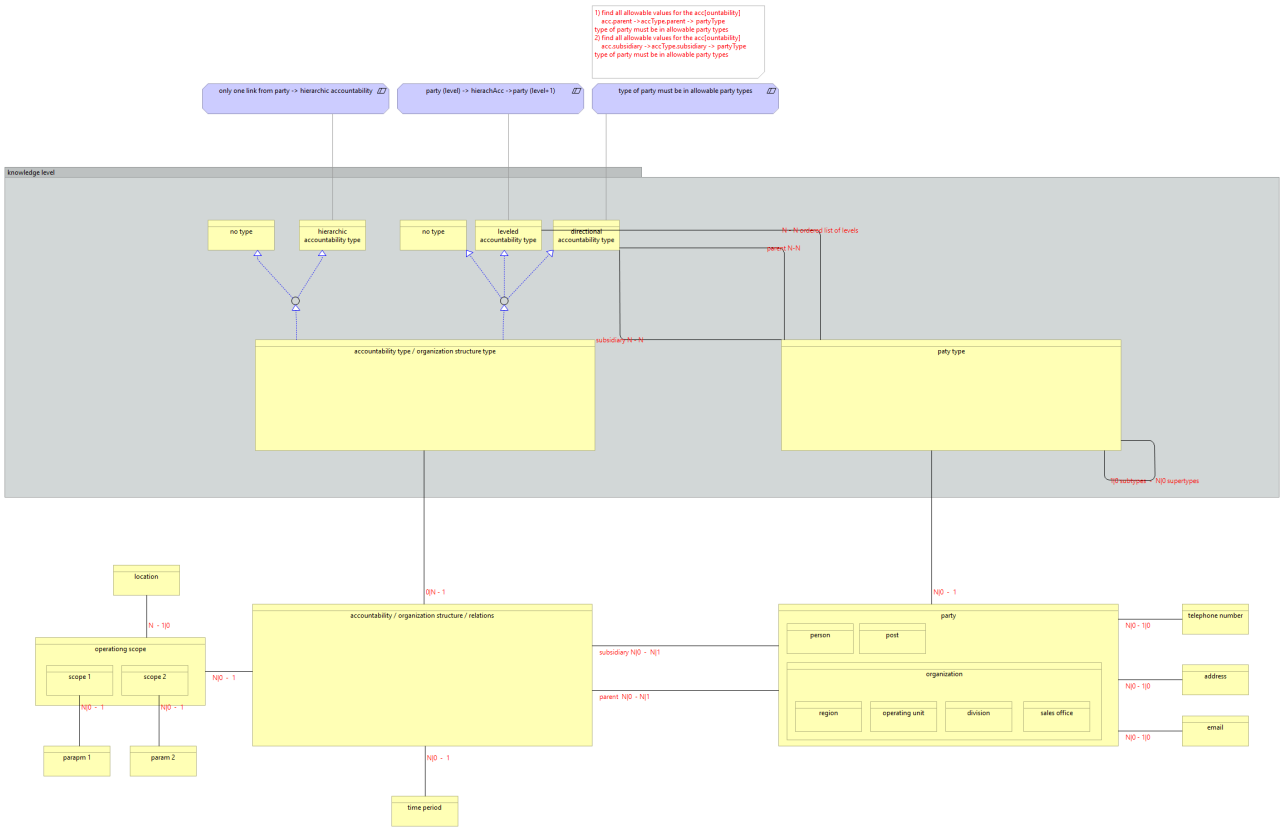
Martin Fowler

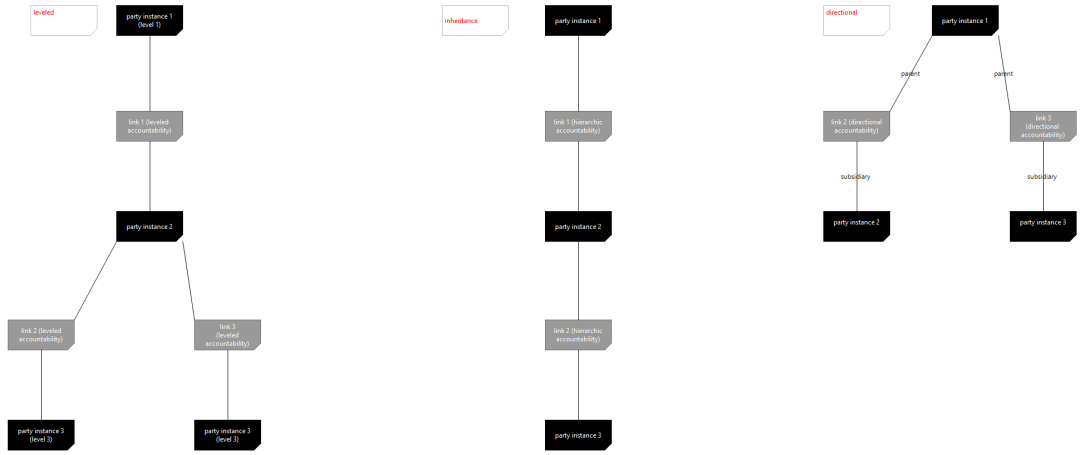


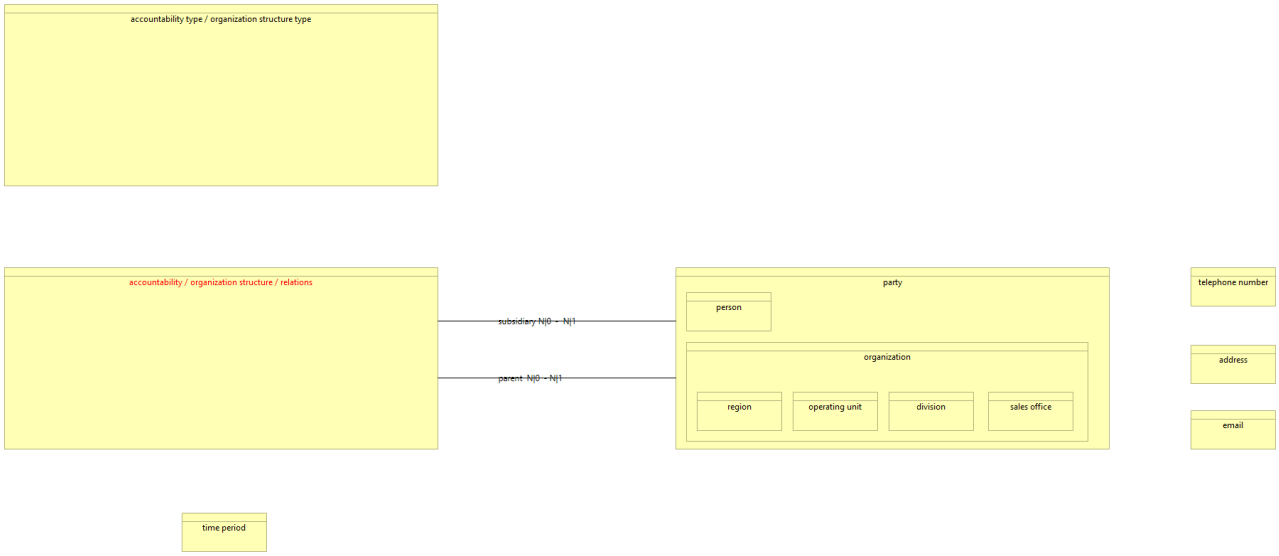
PARTY

party
person
organization
telephone number
address
email

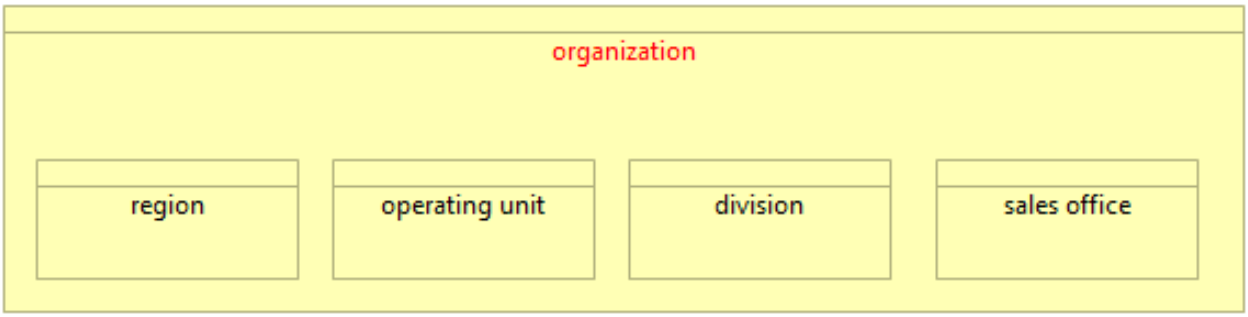
ACCOUNTABILITY



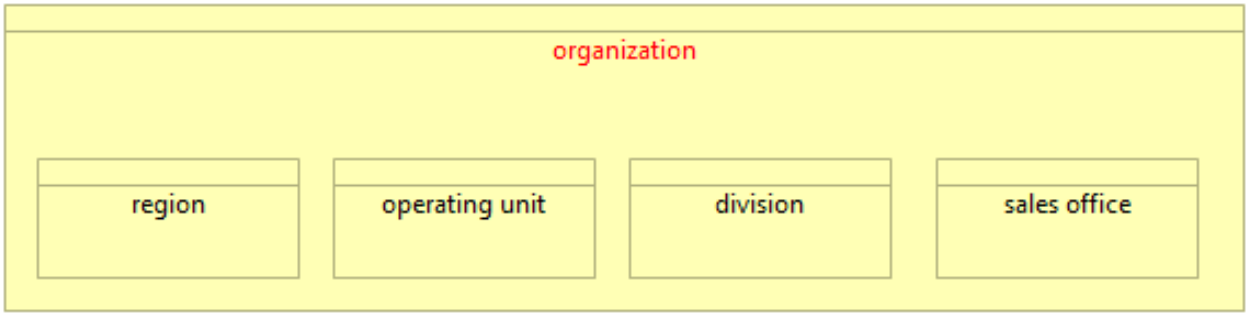



accountability type / organization structure type
accountability / organization structure / relations
subsidiary N|0 - N|1
parent N|0 - N|1
party
person
organization
region
operating unit
division
sales office
telephone number
address
email
time period

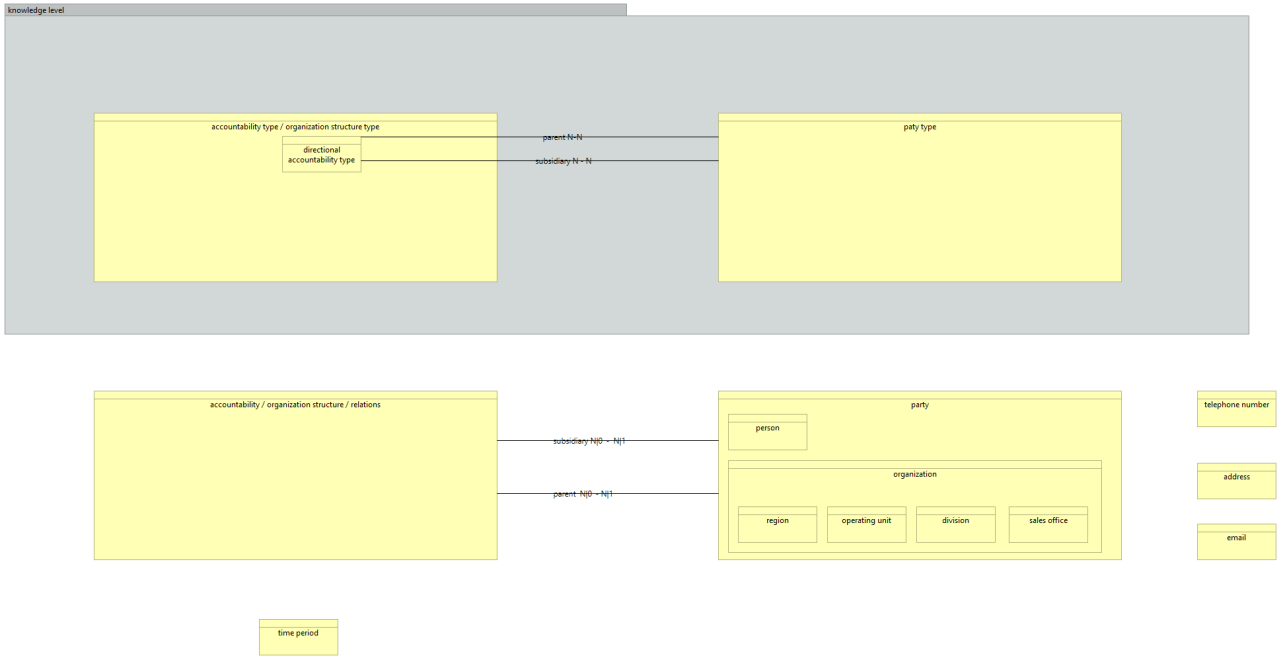
ORGANIZATION HIERARCHIES



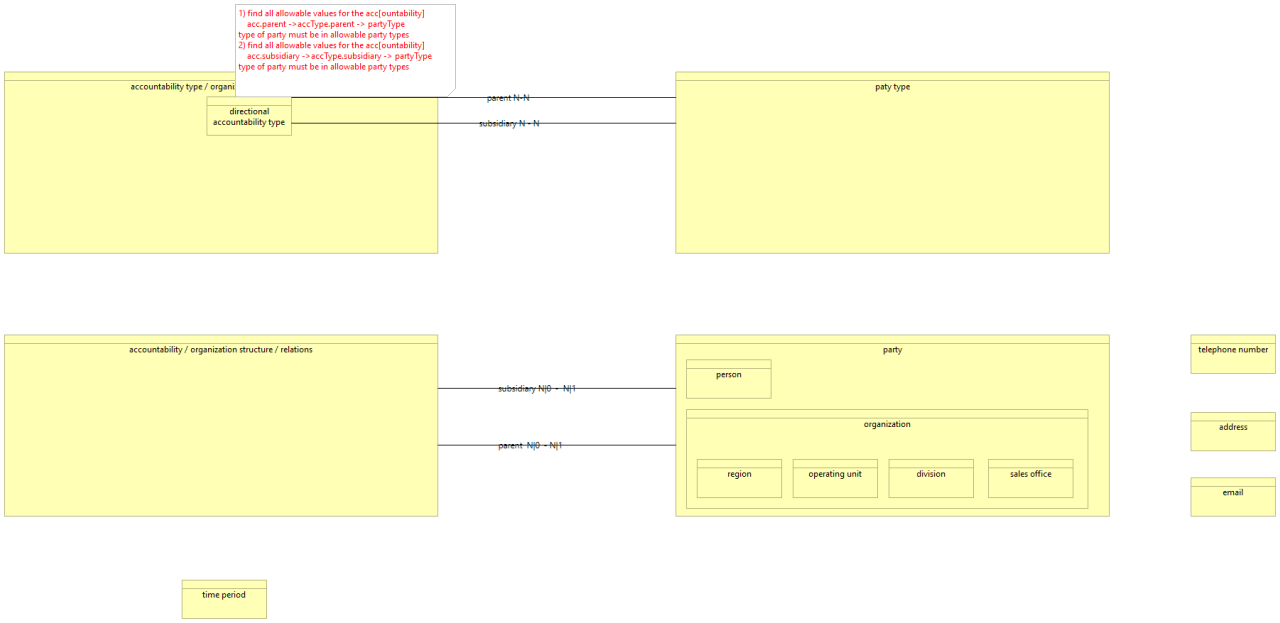
ORGANIZATION STRUCTURE



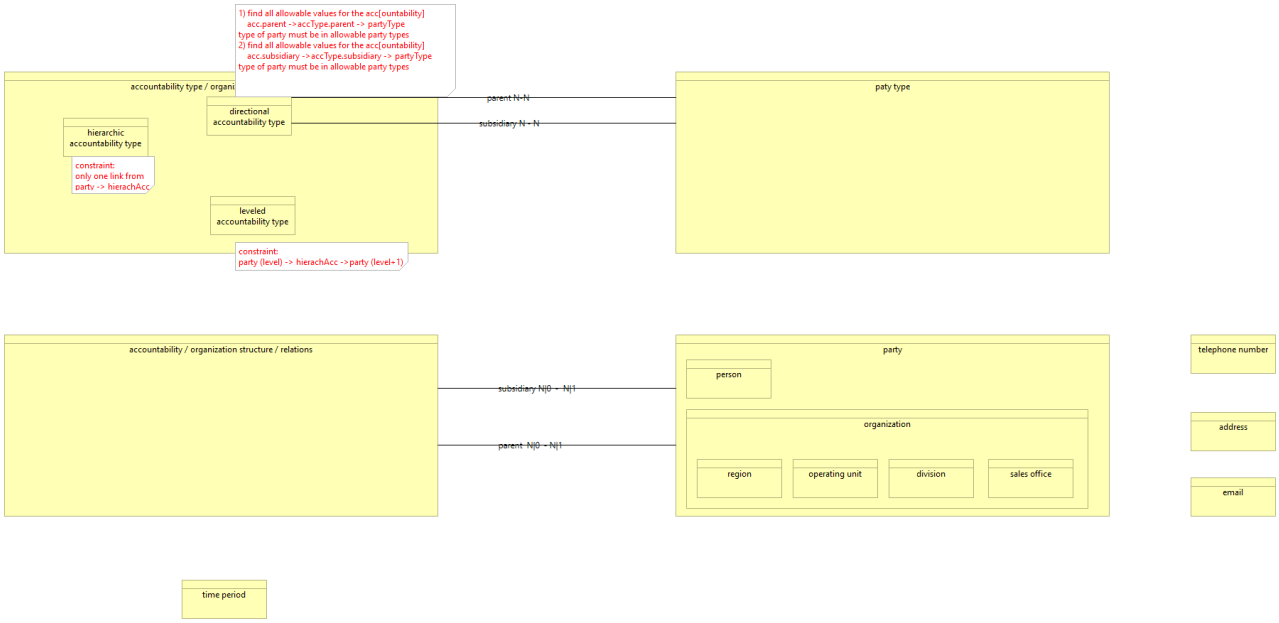
ACCOUNTABILITY KNOWLEDGE LEVEL



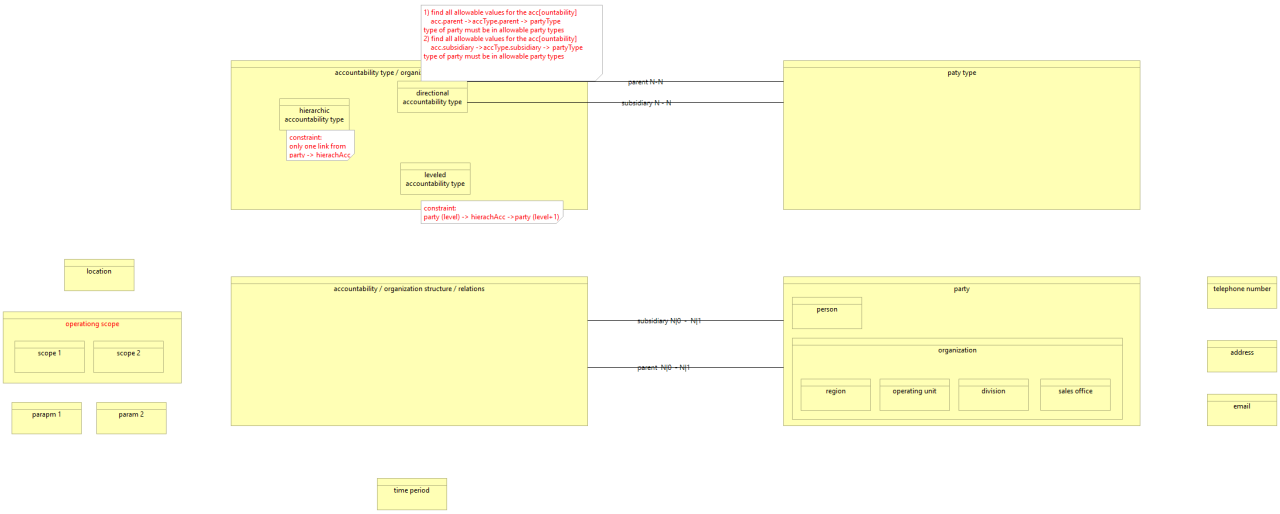
PARTY TYPE GENERALIZATIONS



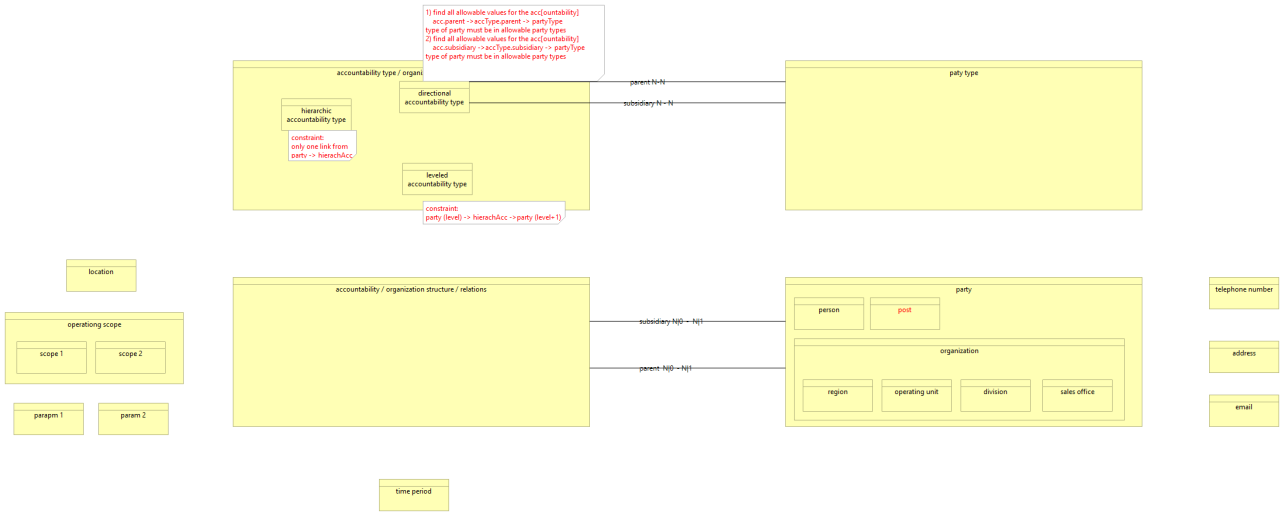
HIERARCHIC ACCOUNTABILITY



OPERATING SCOPES



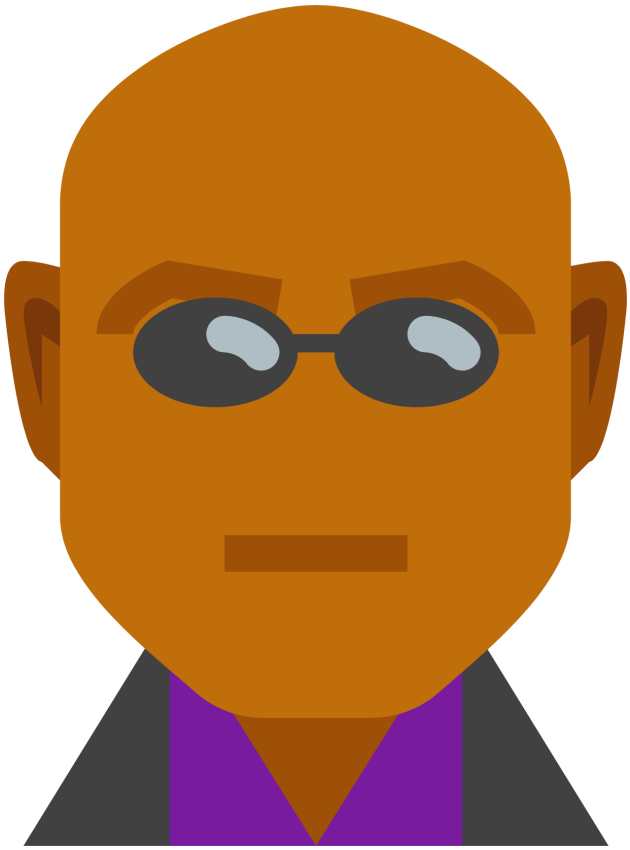
POST


1) find all allowable values for the acc[ountability]
acc.parent -> accType.parent -> partyType
type of party must be in allowable party types
2) find all allowable values for the acc[ountability]
acc.subsidiary -> accType.subsidiary -> partyType
type of party must be in allowable party types
accountability type / organi
directional
accountability type
hierarchic
accountability type
constraint:
only one link from
party -> hierachAcc
leveled
accountability type
constraint:
party (level) -> hierachAcc -> party (level+1)
parent N-N
subsidiary N - N
paty type
location
operationg scope
scope 1
scope 2
paraprn 1
param 2
accountability / organization structure / relations
subsidiary N0 - N1
parent N0 - N1
party
person
post
organization
region
operating unit
division
sales office
telephone number
address
email
time period

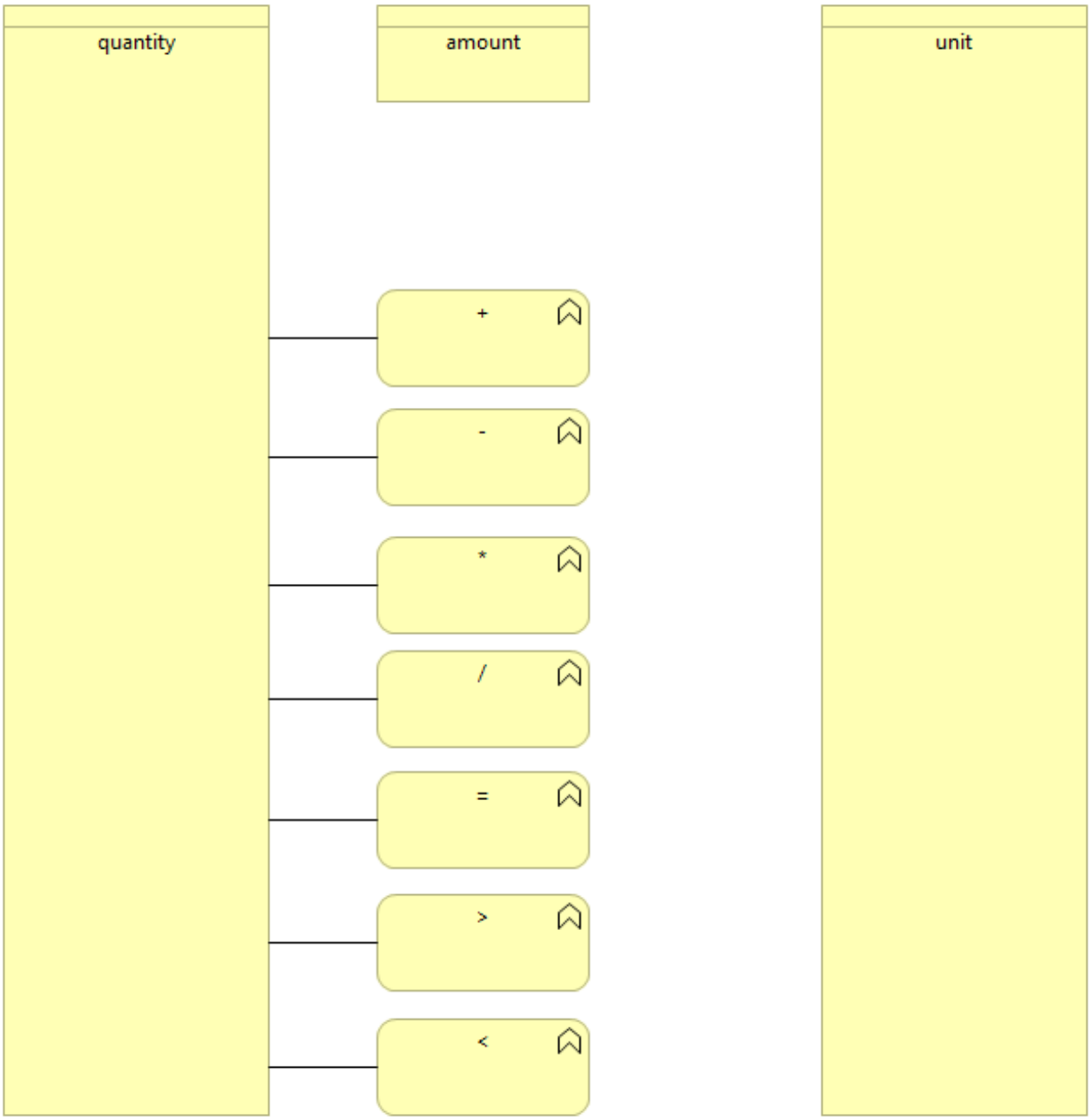
OBSERVATIONS AND MEASUREMENTS

ANALYSIS PATTERNS

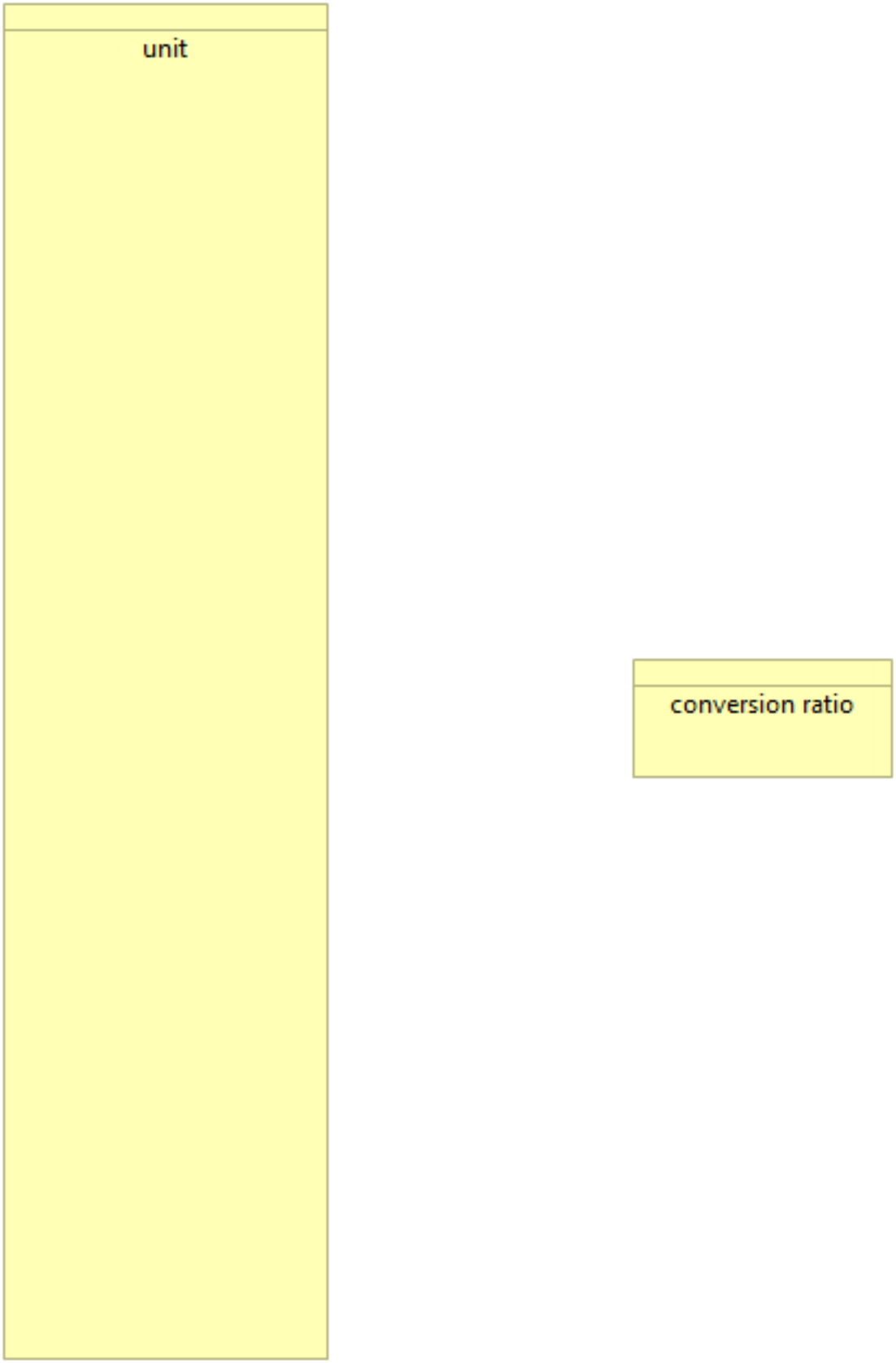
Martin Fowler



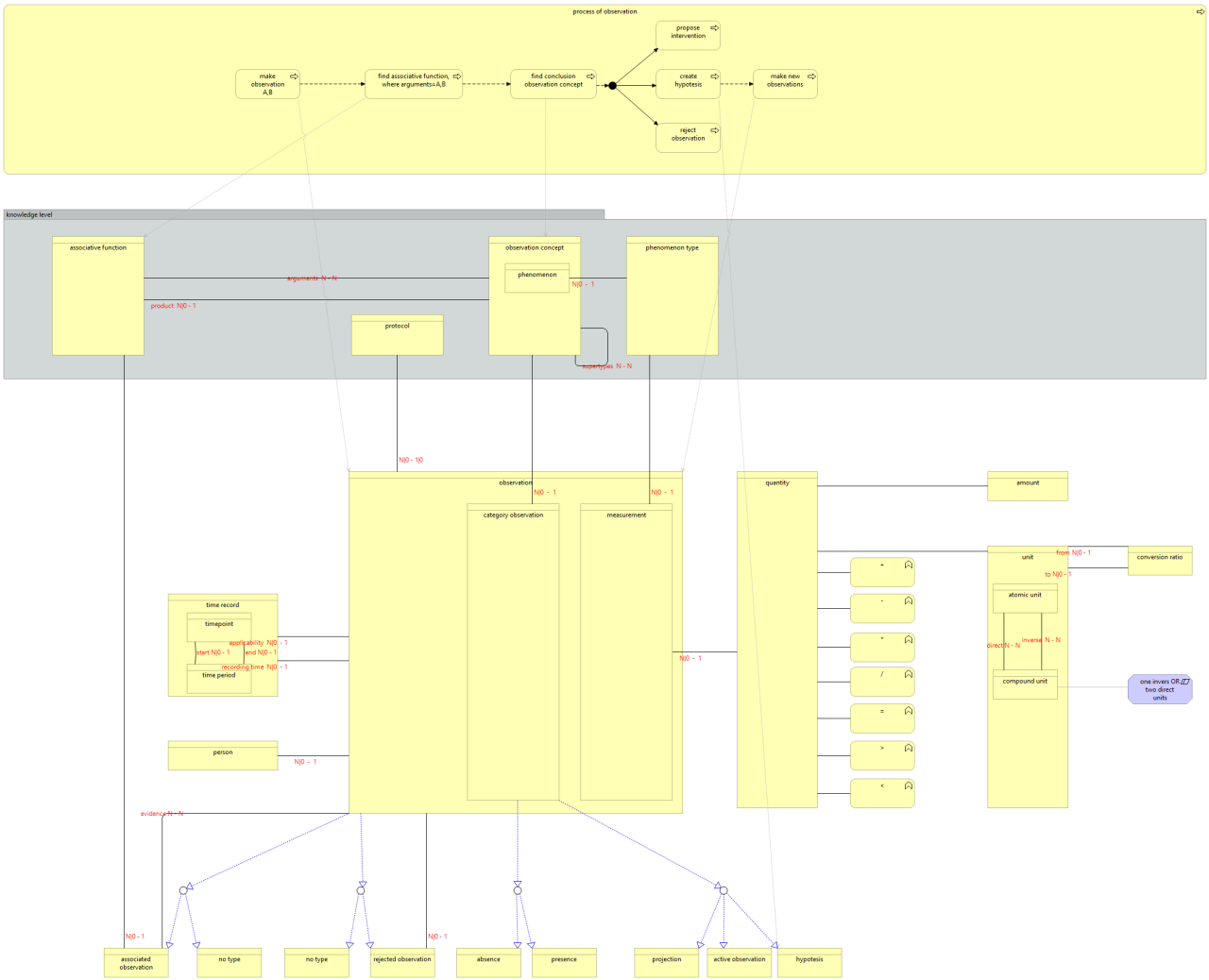
QUANTITY

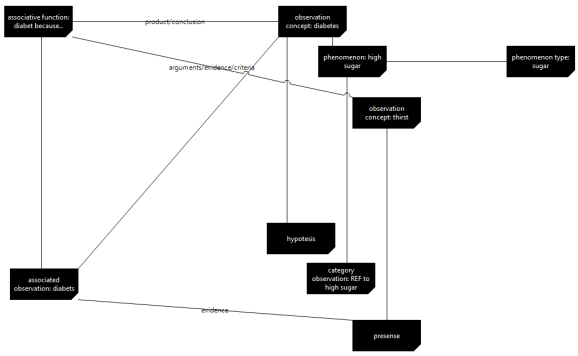


CONVERSION RATIO

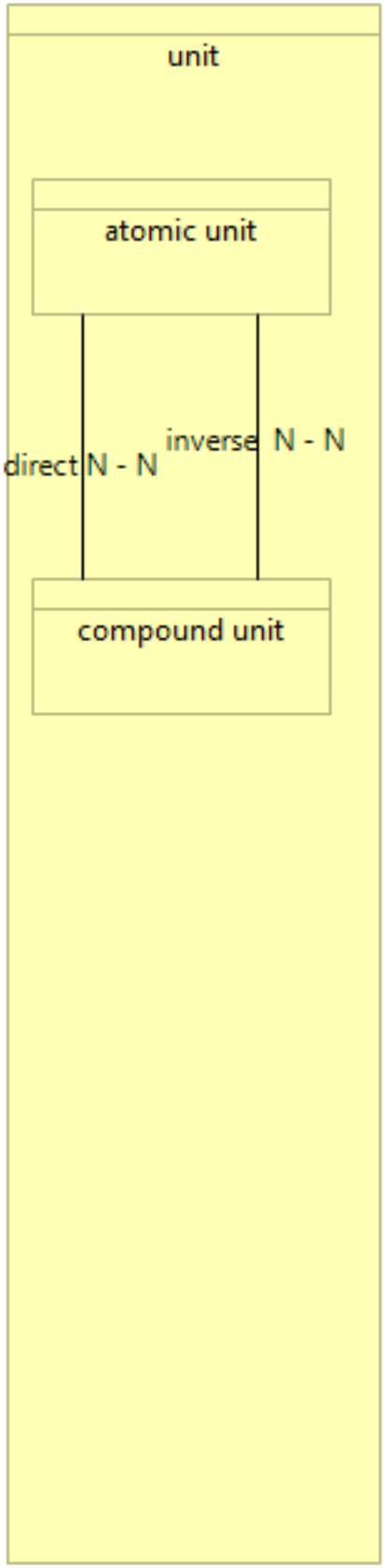


OBSERVATIONS AND MEASUREMENTS



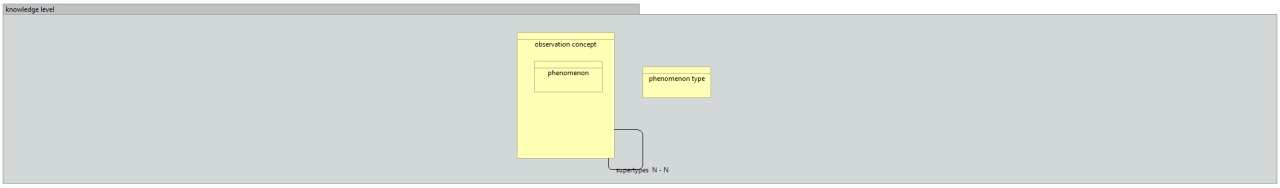


COMPOUND UNITS

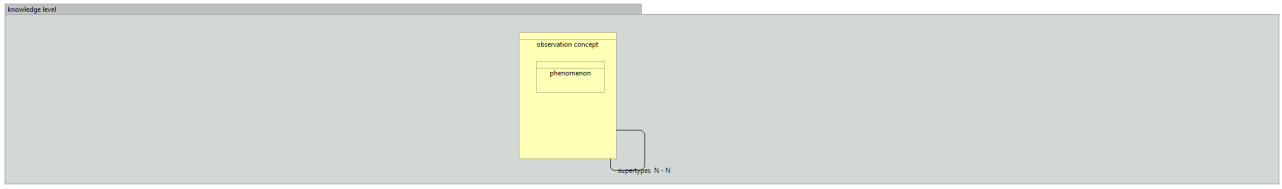


MEASUREMENT

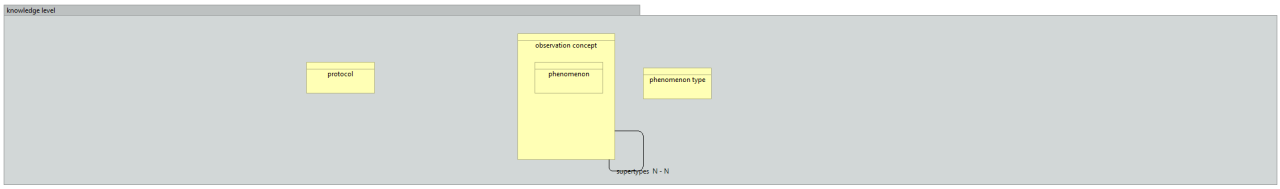
OBSERVATION



SUBTYPING OBSERVATION CONCEPTS

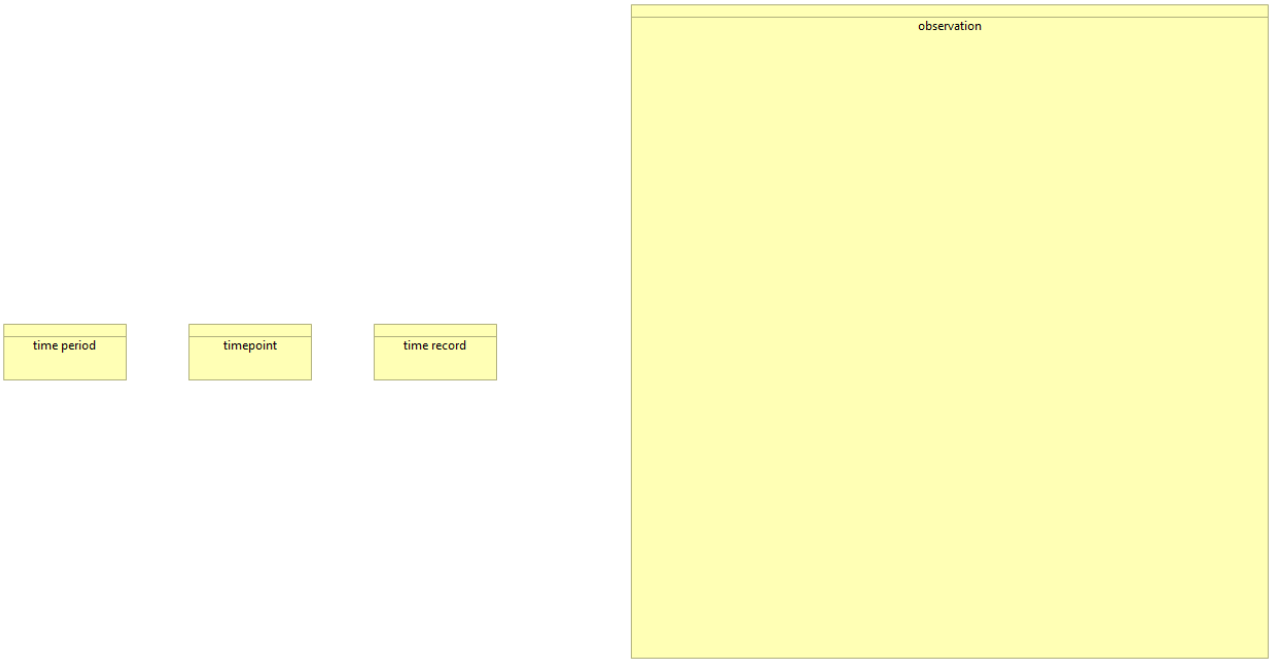


PROTOCOL


knowledge level
protocol
observation concept
phenomenon
phenomenon type
supertypes N - N

observation
category observation
absence
presence
measurement
person
quantity
amount
+
-
*
/
=
>
<

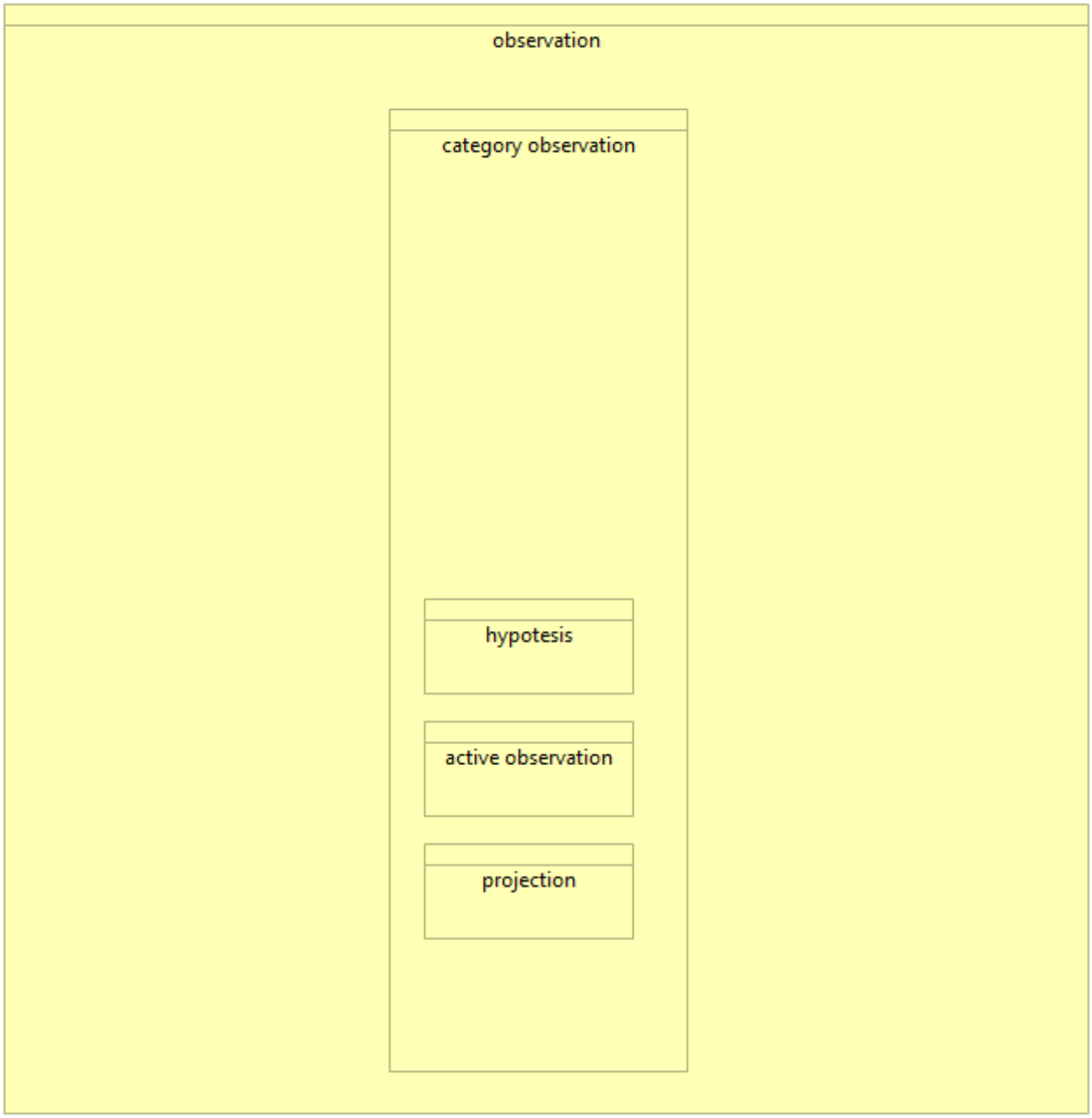
DUAL TIME RECORD



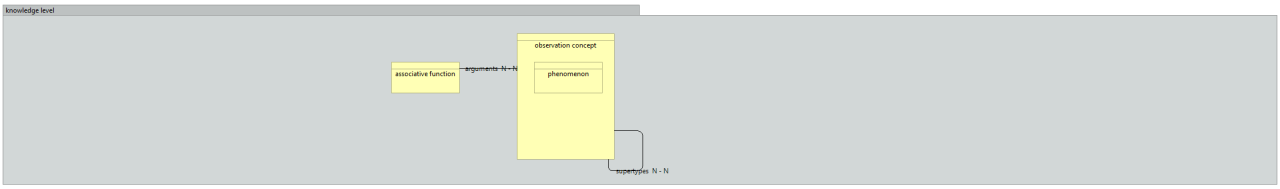
REJECTED OBSERVATION

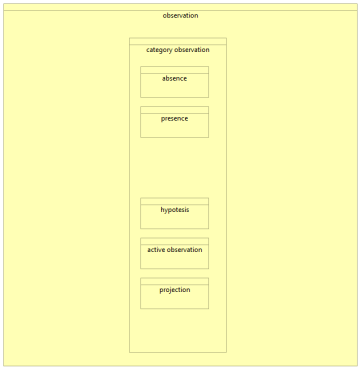
observation

ACTIVE OBSERVATION, HYPOTHESIS, AND PROJECTION

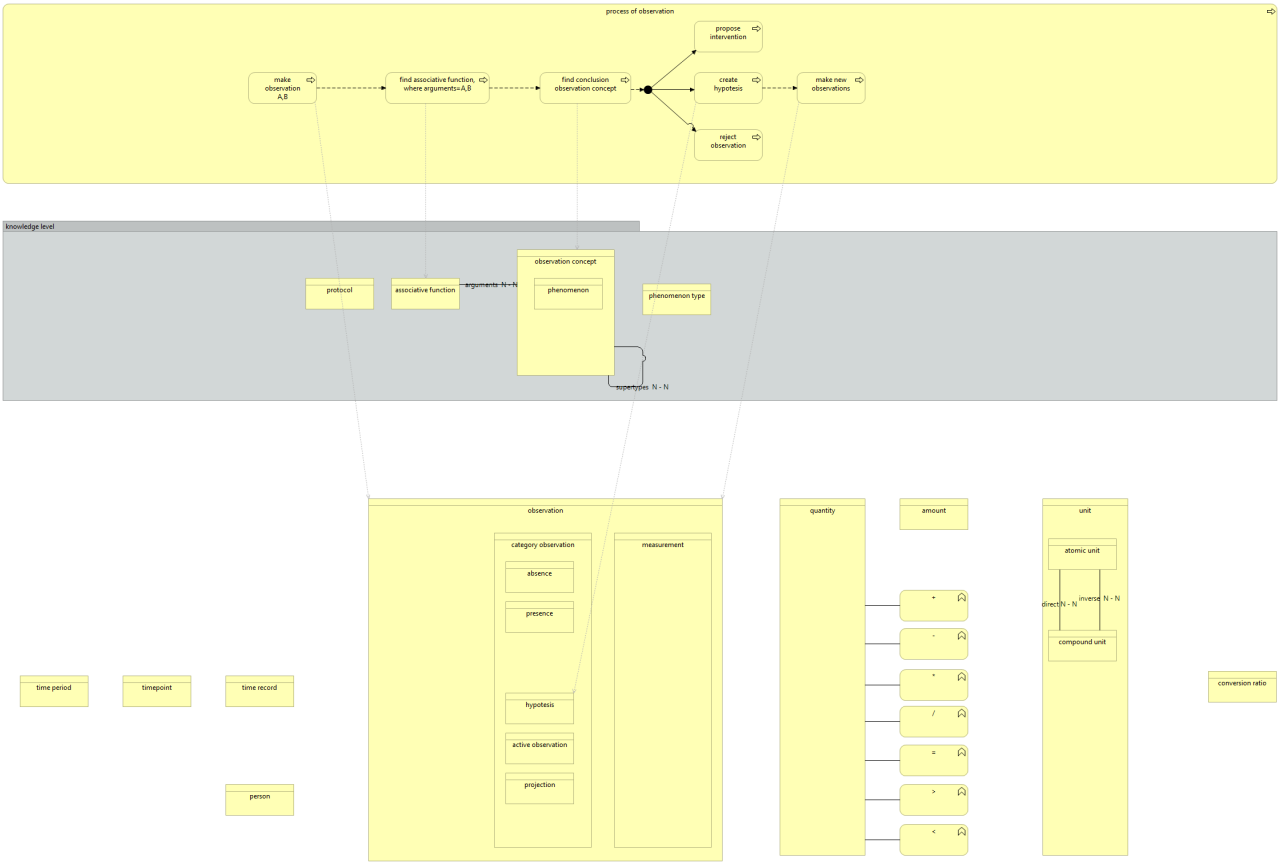


ASSOCIATED OBSERVATION





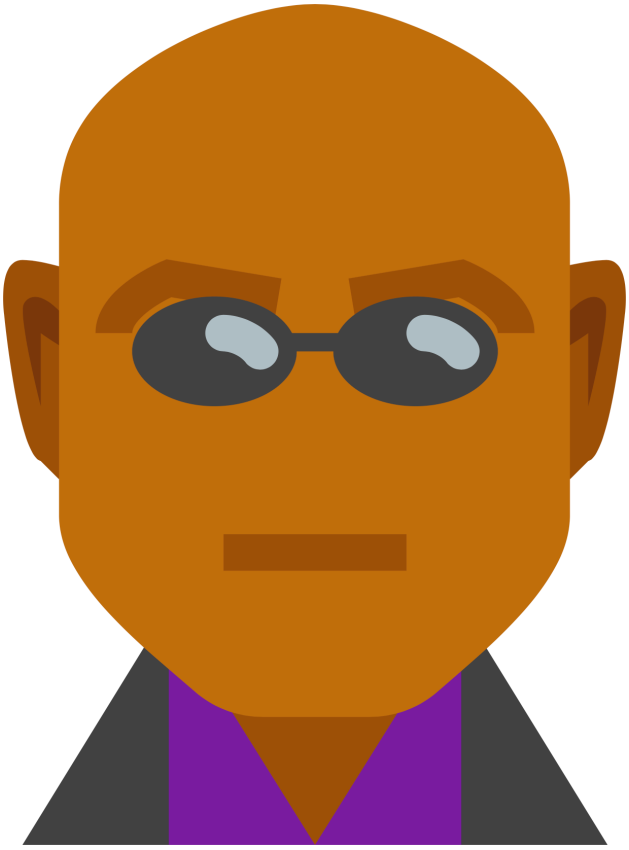
PROCESS OF OBSERVATION



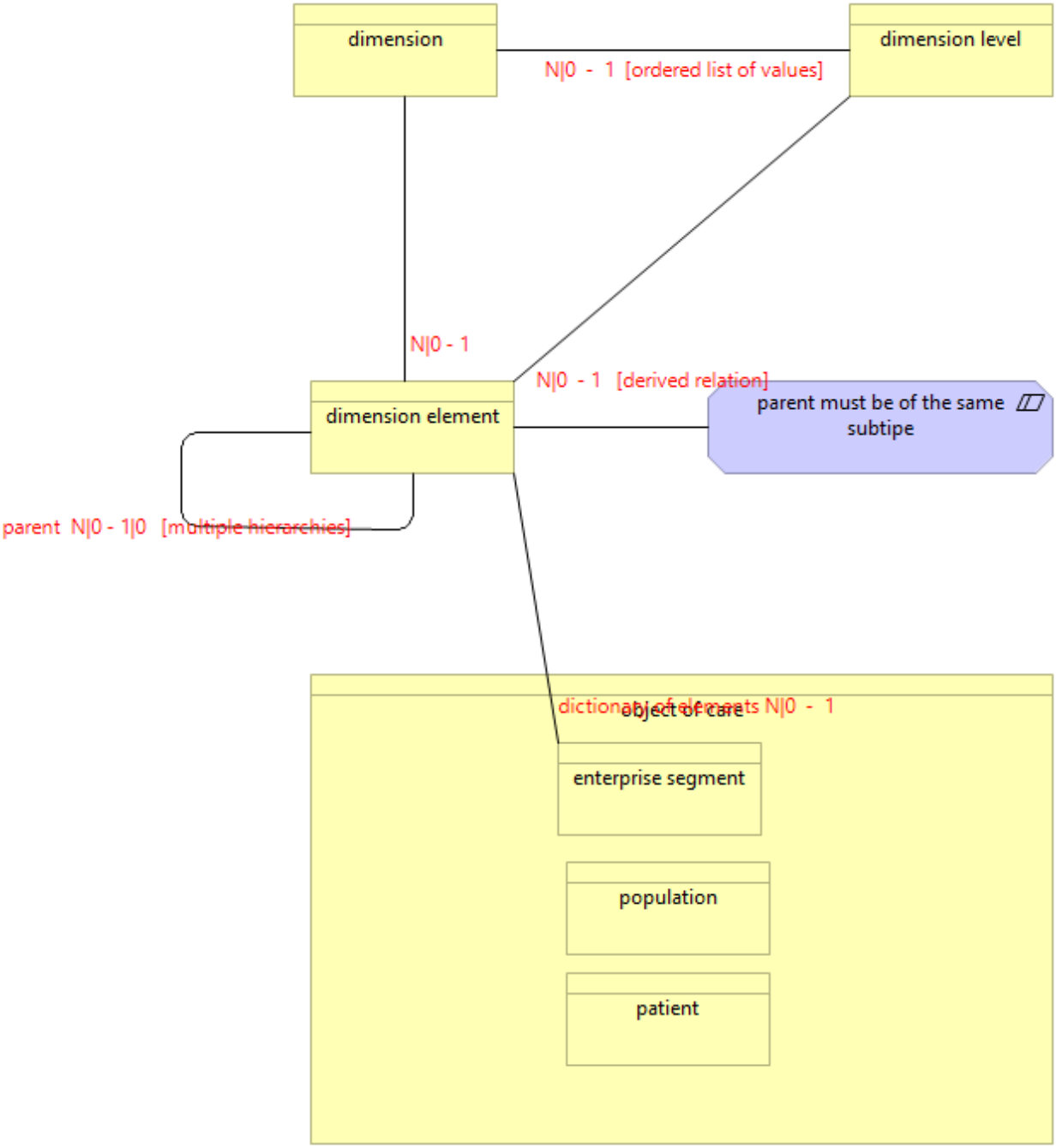
OBSERVATIONS FOR CORPORATE FINANCE

ANALYSIS PATTERNS

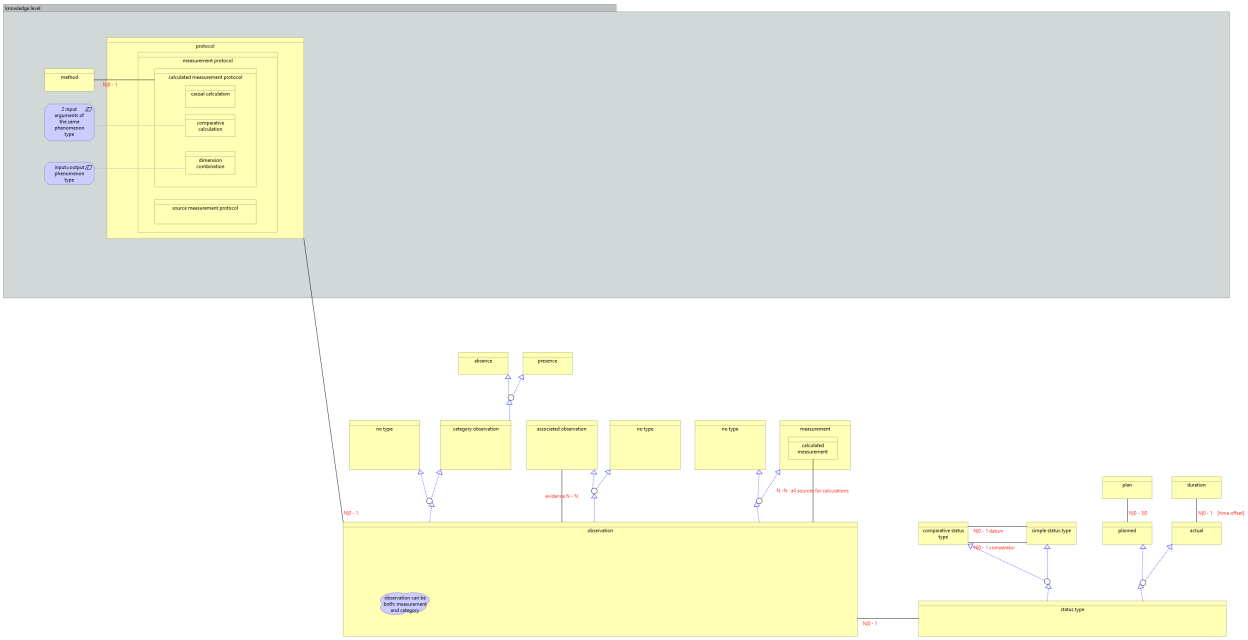
Martin Fowler



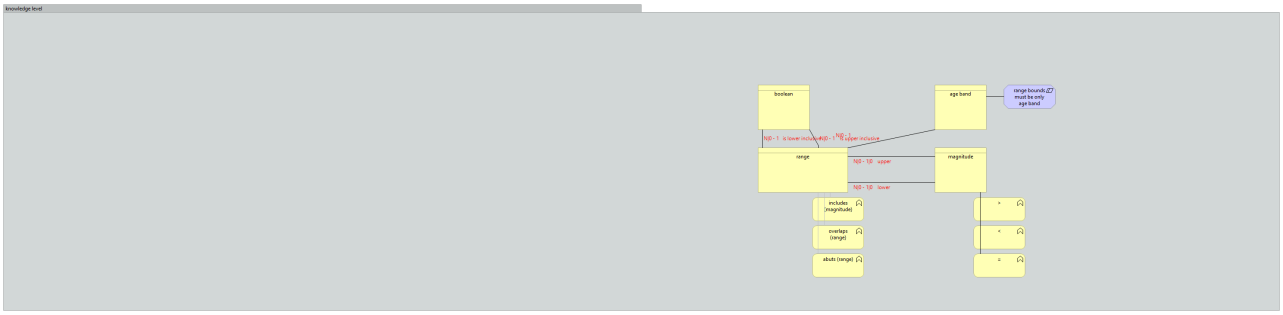
ENTERPRISE SEGMENT



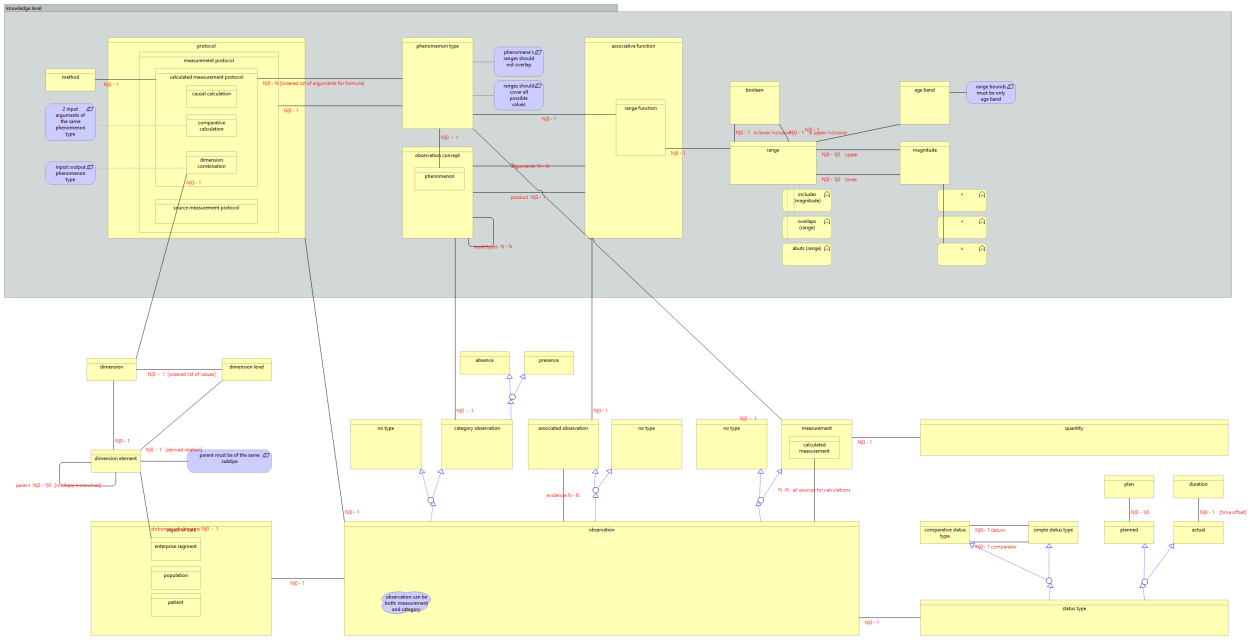
MEASUREMENT PROTOCOL

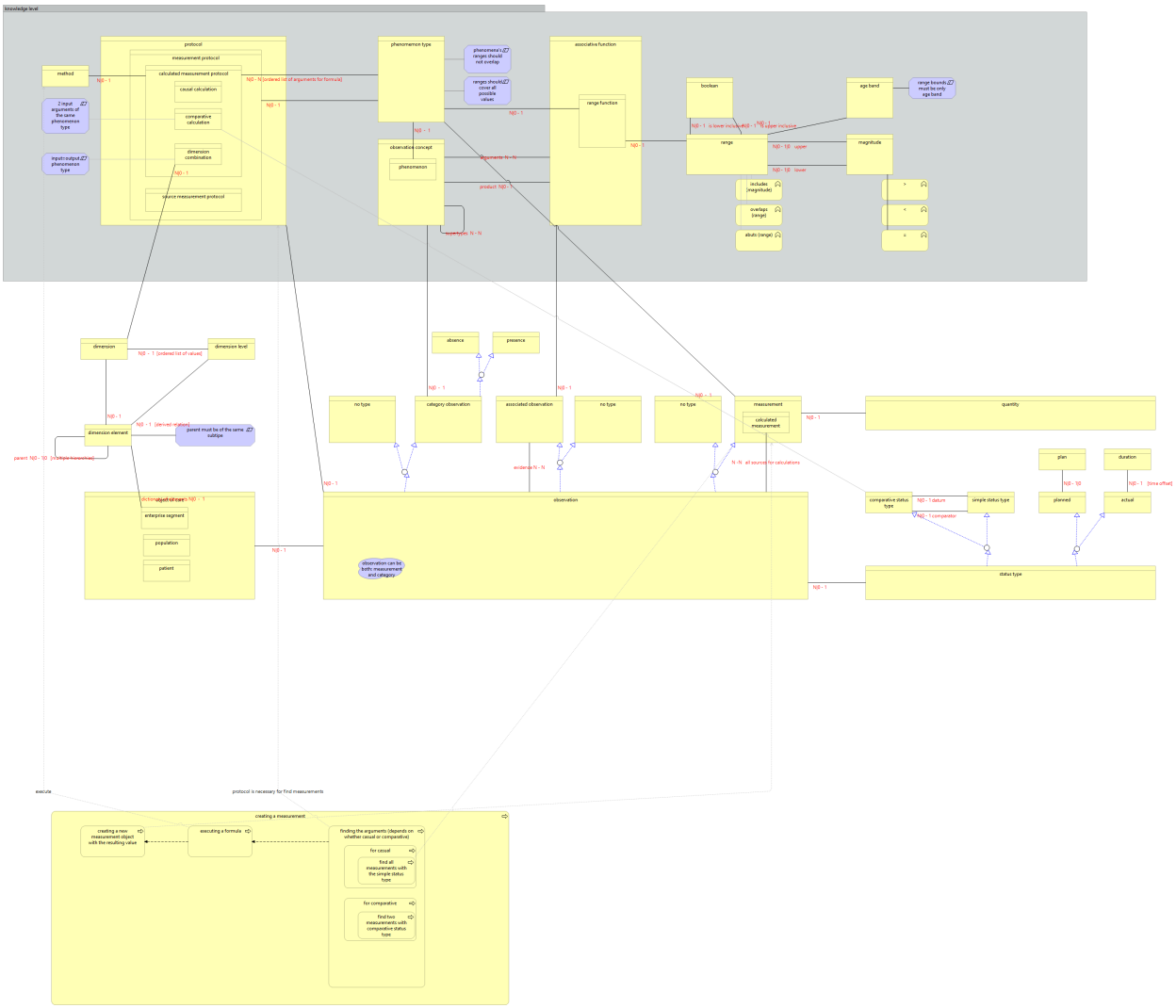


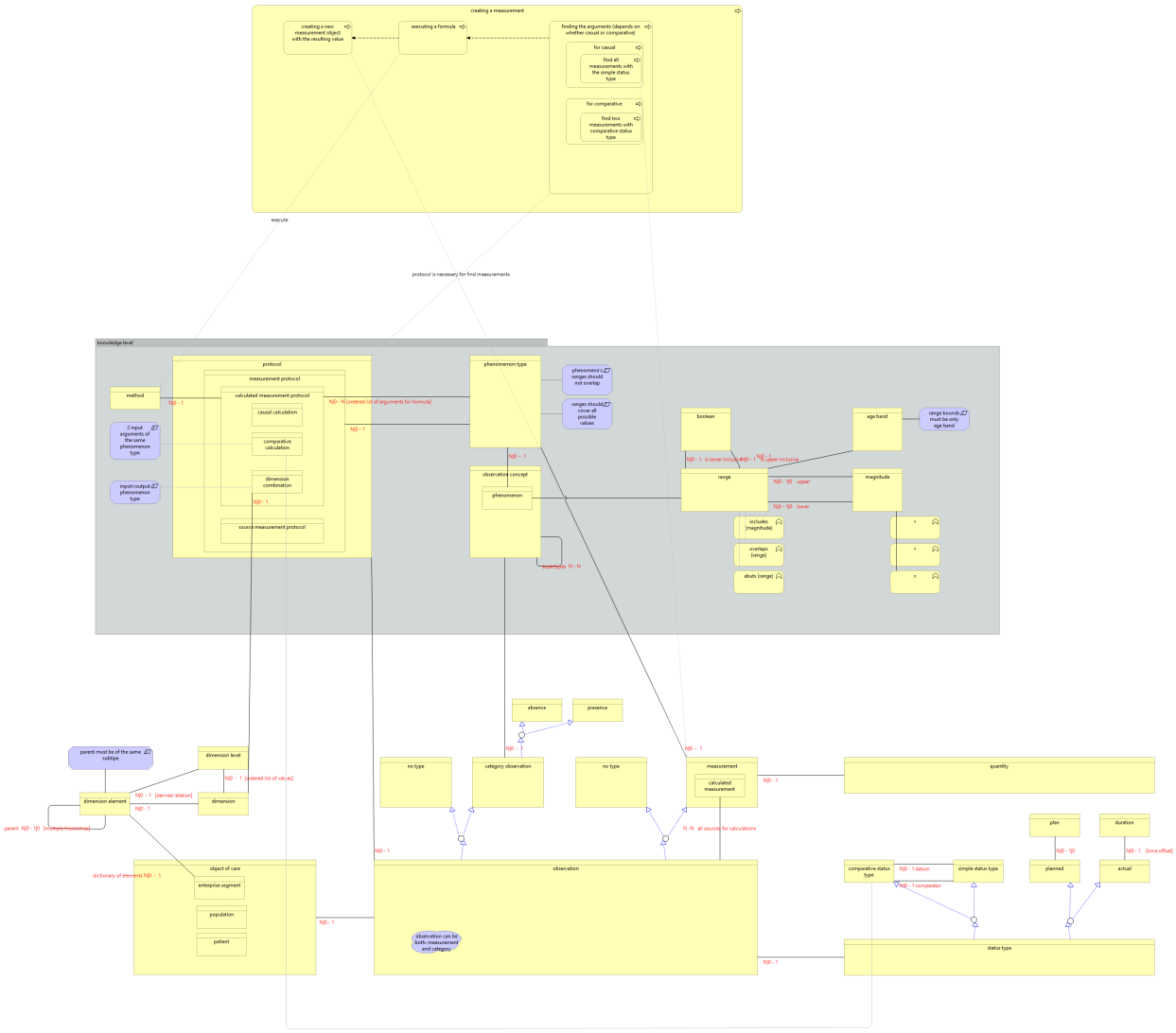
RANGE


knowledge level
boolean
age band
range bounds must be only age band
N2 - 1 is lower inclusive
N2 - 1 is upper inclusive
range
N2 - 1:0 upper
N2 - 1:0 lower
magnitude
includes (magnitude)
overlaps (range)
abuts (range)
>
<
=

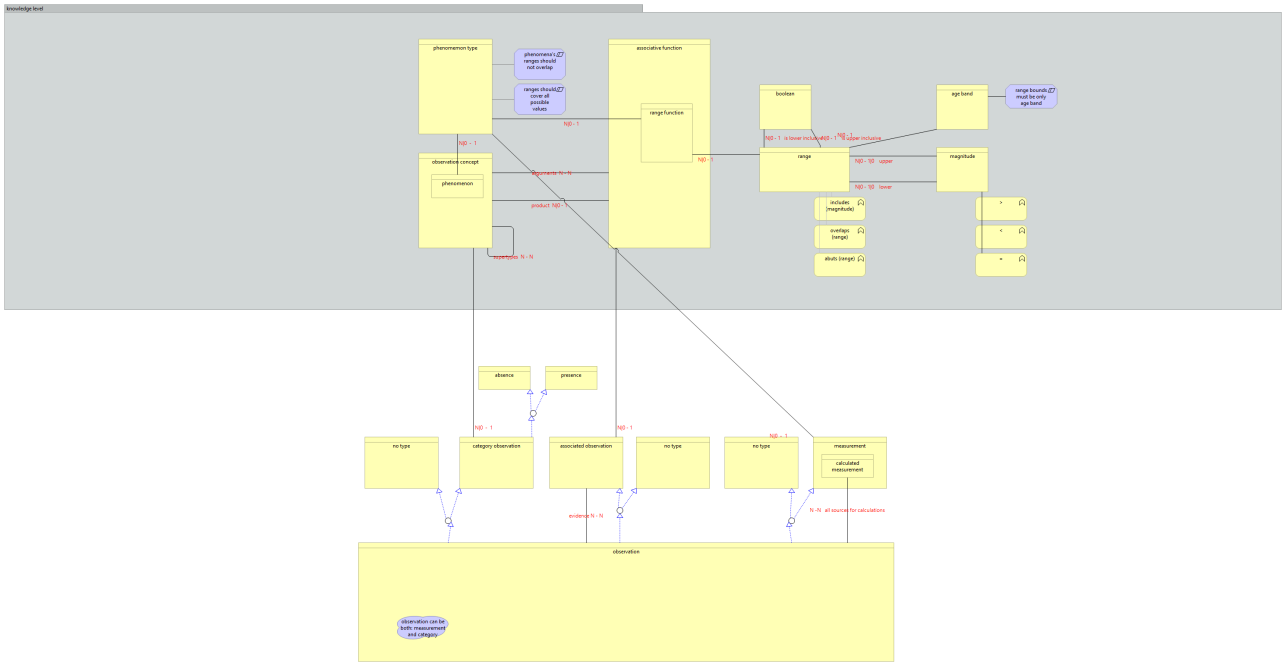
OBSERVATIONS FOR CORPORATE FINANCE




knowledge level
protocol
measurement protocol
calculated measurement protocol
causal calculation
comparative calculation
dimension combination
source measurement protocol
method
1 input - arguments of the same phenomenon type
input-output phenomenon type
N|0 - 1
N|0 - N [ordered list of arguments for formula]
phenomenon type
phenomena ranges should not overlap
ranges should cover all possible values
observation concept
phenomenon
arguments N - N
product N|0 - 1
supertypes N - N
associative function
range function
boolean
is lower inclusive
is upper inclusive
range
upper
lower
includes (magnitude)
overlaps (range)
abuts (range)
age band
range bounds must be only age band
magnitude
>
<
≥
dimension
dimension level
N|0 - 1 [ordered list of values]
dimension element
N|0 - 1 (derived relation)
parent must be of the same subtype
parent N|0 - 1[0..1 simple hierarchy]
actor or role
N|0 - 1
enterprise segment
population
patient
absence
presence
no type
category observation
associated observation
no type
no type
measurement
calculated measurement
quantity
evidence N - N
N - N all sources for calculations
observation
observation can be both measurement and category
comparative status type
N|0 - 1 datum
N|0 - 1 comparator
simple status type
plan
duration
planned
actual
N|0 - 1[0]
N|0 - 1 [time offset]
status type
execute
protocol is necessary for find measurements
creating a measurement
creating a new measurement object with the resulting value
executing a formula
finding the arguments (depends on whether casual or comparative)
for casual
find all measurements with the simple status type
for comparative
find two measurements with comparative status type


creating a measurement
creating a new measurement object with the resulting value
executing a formula
finding the arguments (depends on whether casual or comparative)
for casual
find all measurements with the simple status type
for comparative
find two measurements with comparative status type
execute
protocol is necessary for find measurements
knowledge level
method
N(0 - 1
2 input arguments of the same phenomenon type
input/output phenomenon type
protocol
measurement protocol
calculated measurement protocol
causal calculation
comparative calculation
dimension combination
N(0 - 1
source measurement protocol
N(0 - N (ordered list of arguments for formula)
N(0 - 1
phenomenon type
phenomena's ranges should not overlap
ranges should cover all possible values
N(0 - 1
observation concept
phenomenon
subclass N - N
boolean
N(0 - 1 is lower inclusive
N(0 - 1 is upper inclusive
age band
range bounds must be only age band
range
N(0 - 1)0 upper
N(0 - 1)0 lower
magnitude
includes (magnitude)
overlaps (range)
abuts (range)
>
<
=
absence
presence
N(0 - 1
N(0 - 1
parent must be of the same subtype
dimension level
N(0 - 1 (ordered list of values)
dimension element
N(0 - 1 (derived relation)
N(0 - 1
dimension
parent N(0 - 1)0 (simple measurement)
no type
category observation
no type
measurement
calculated measurement
N(0 - 1
quantity
N - N all sources for calculations
plan
duration
N(0 - 1)0
N(0 - 1 (time offset)
planned
actual
N(0 - 1
dictionary of elements N(0 - 1
object of care
enterprise segment
population
patient
N(0 - 1
observation
observation can be both: measurement and category
comparative status type
N(0 - 1 datum
N(0 - 1 comparator
simple status type
N(0 - 1
status type

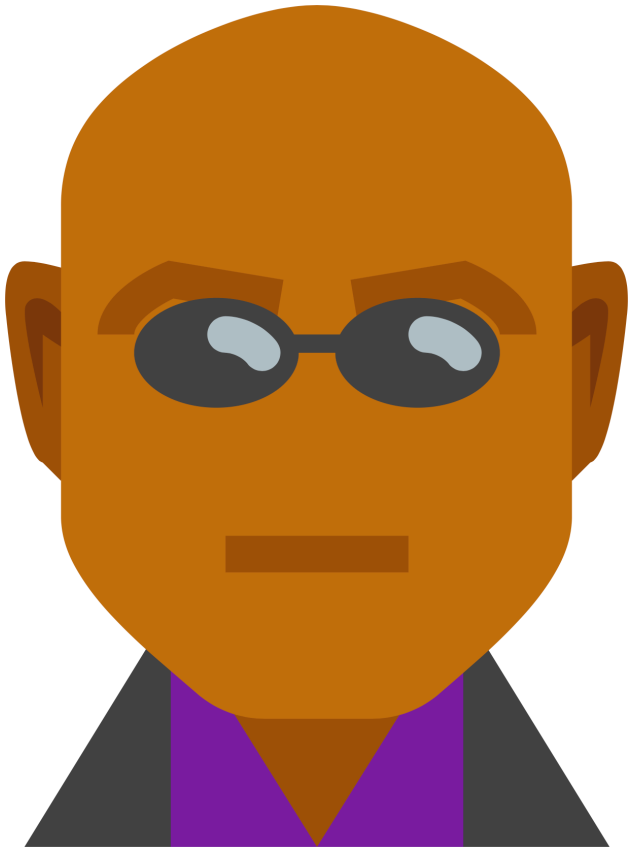
PHENOMENON WITH RANGE



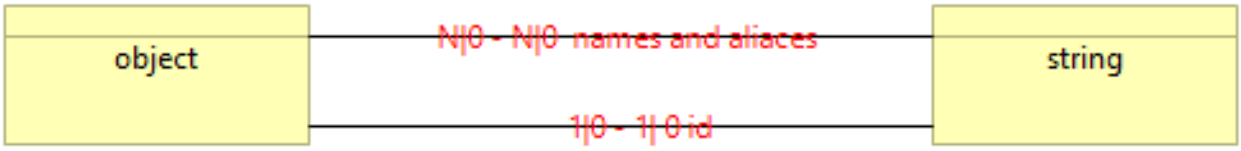
REFERRING TO OBJECTS

ANALYSIS PATTERNS

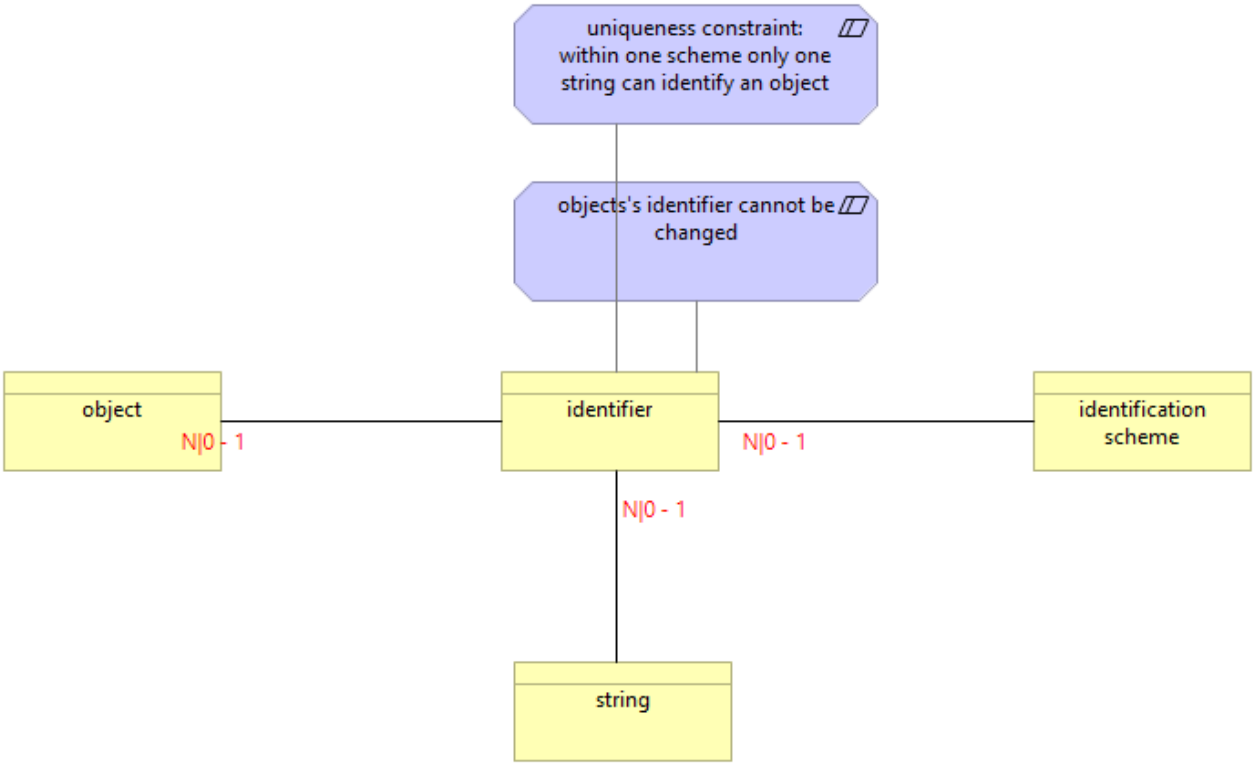
Martin Fowler



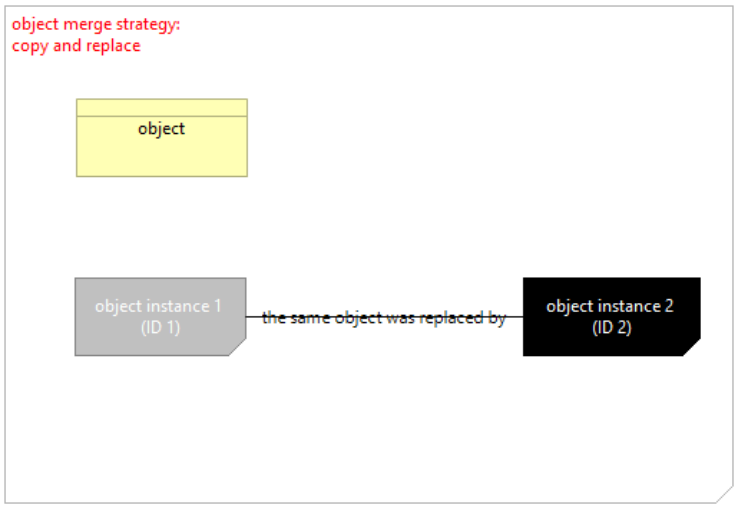
NAME

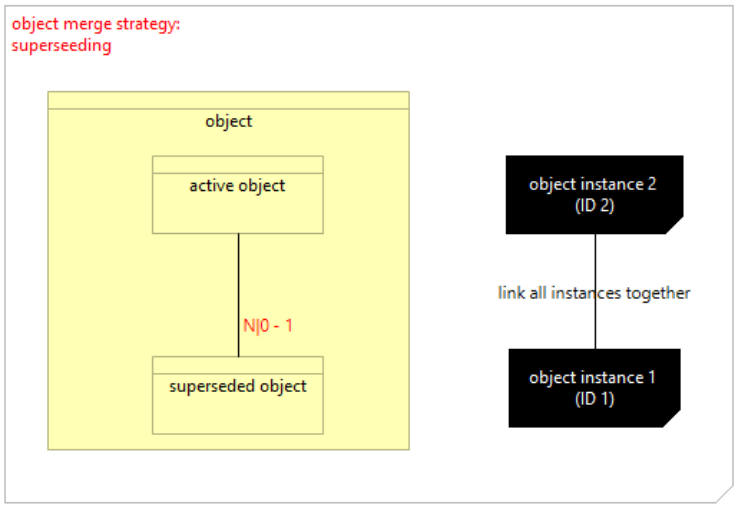


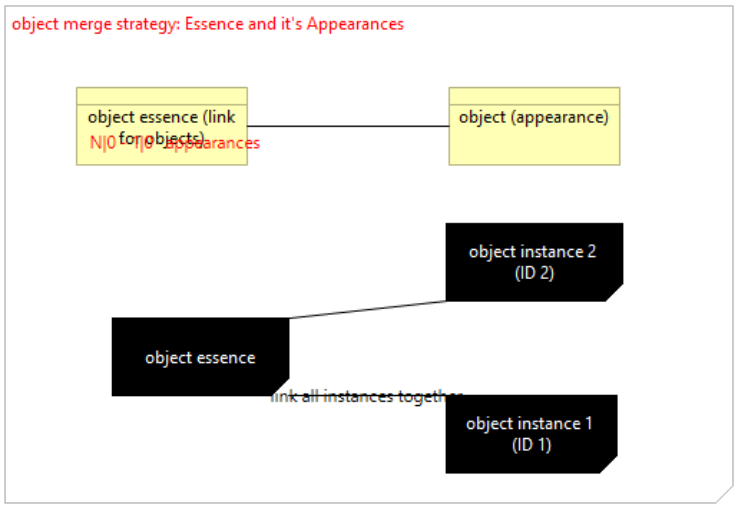
IDENTIFICATION SCHEME



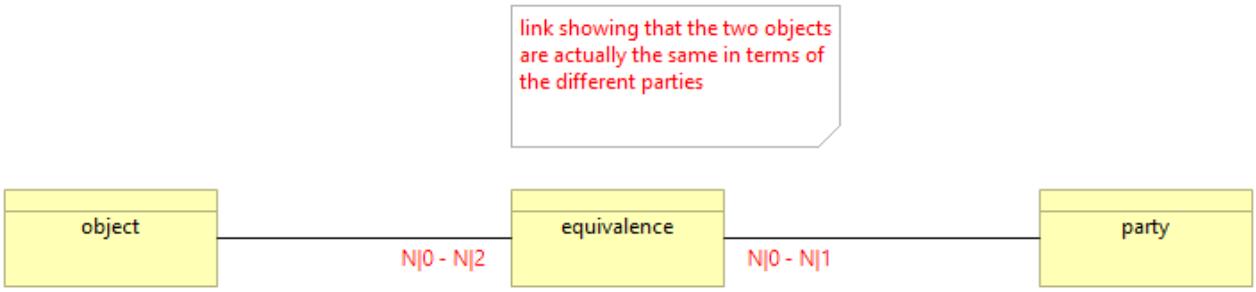
OBJECT MERGE



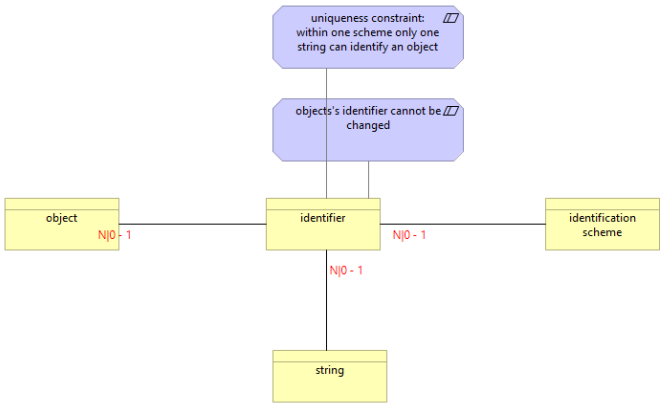


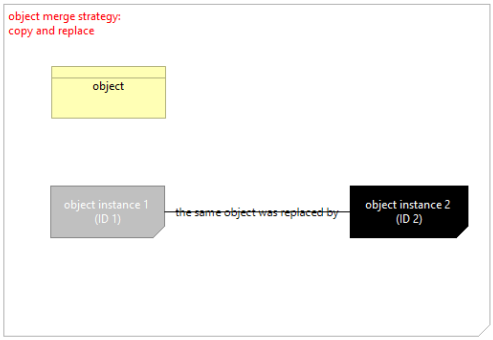


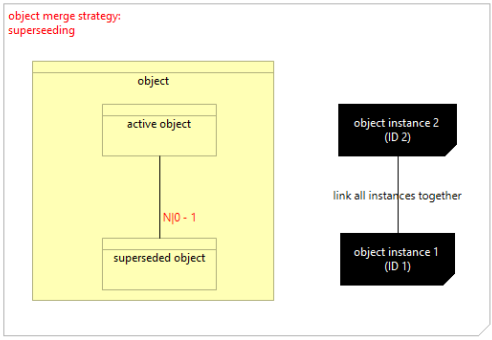
OBJECT EQUIVALENCE

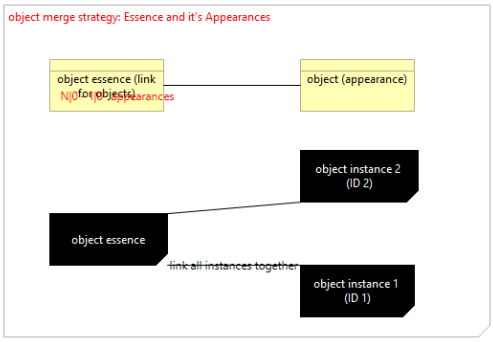


REFERRING TO OBJECTS





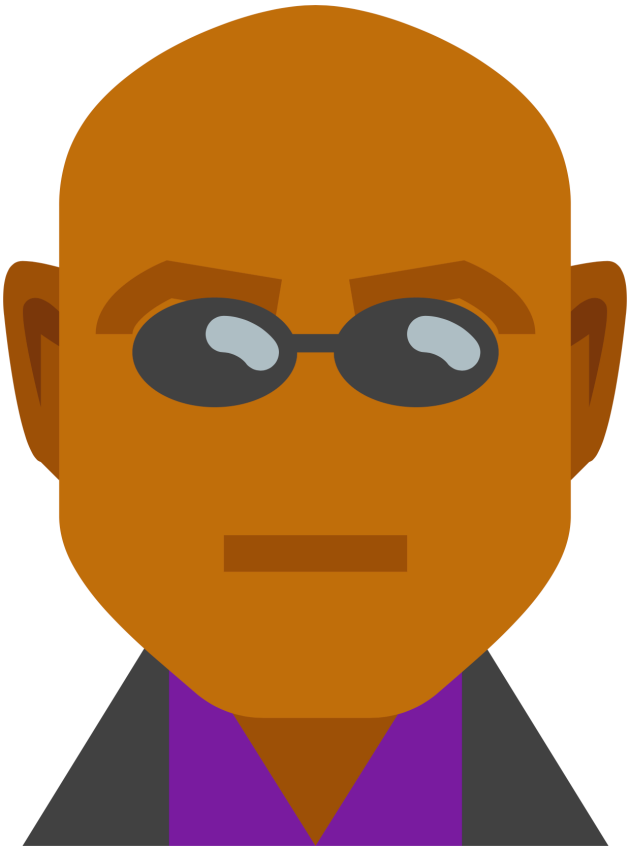




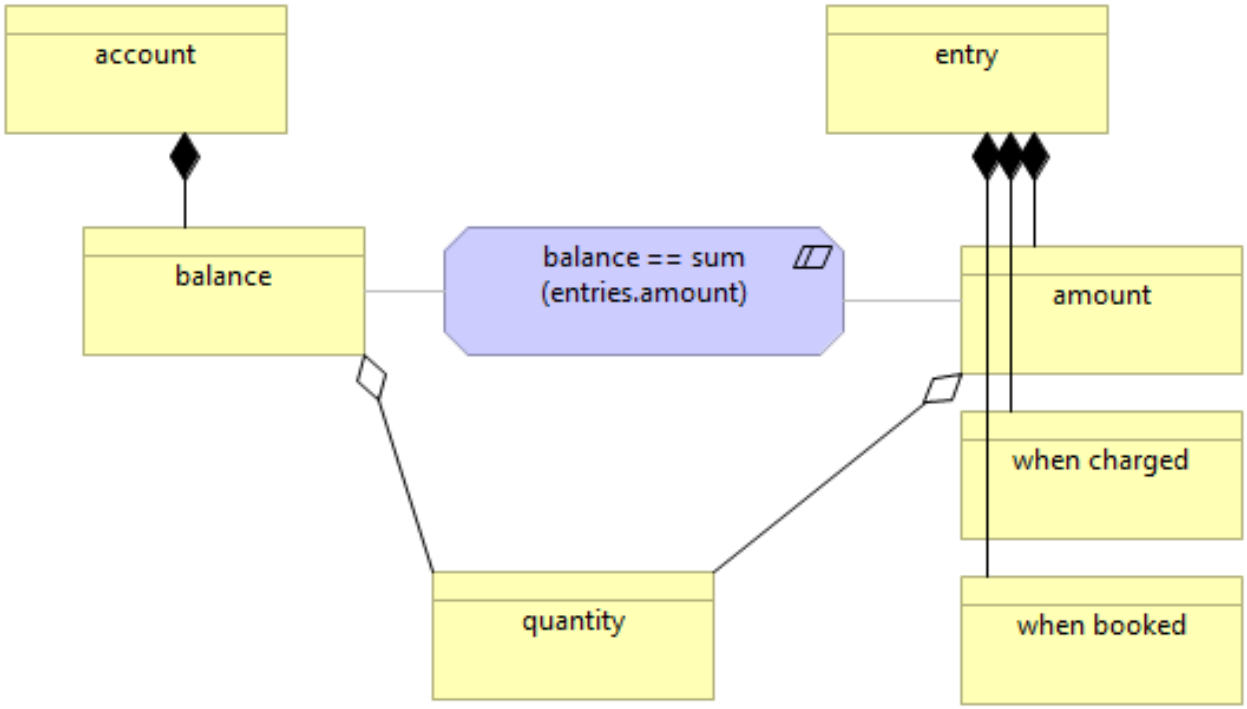
INVENTORY AND ACCOUNTING

ANALYSIS PATTERNS

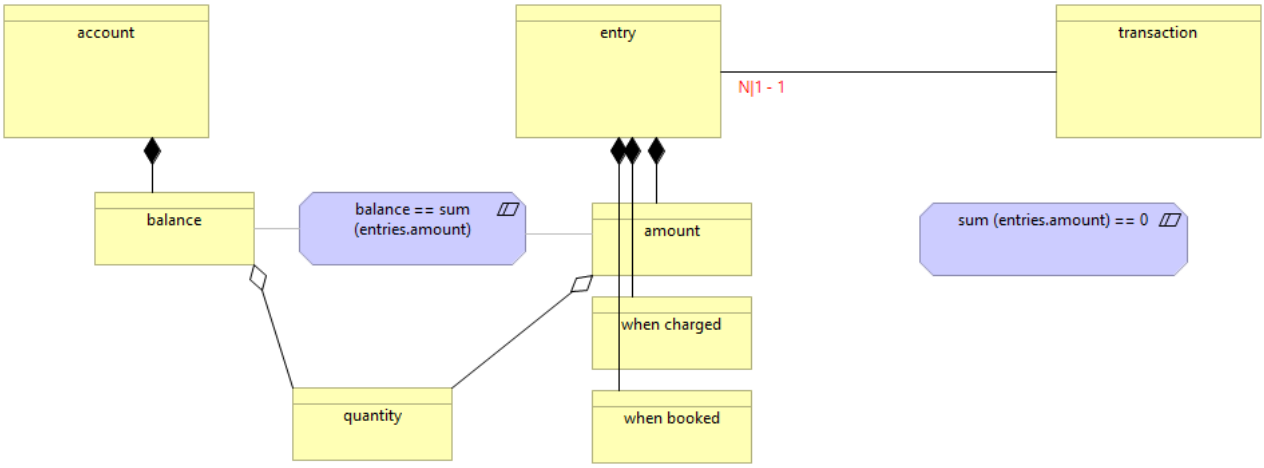
Martin Fowler



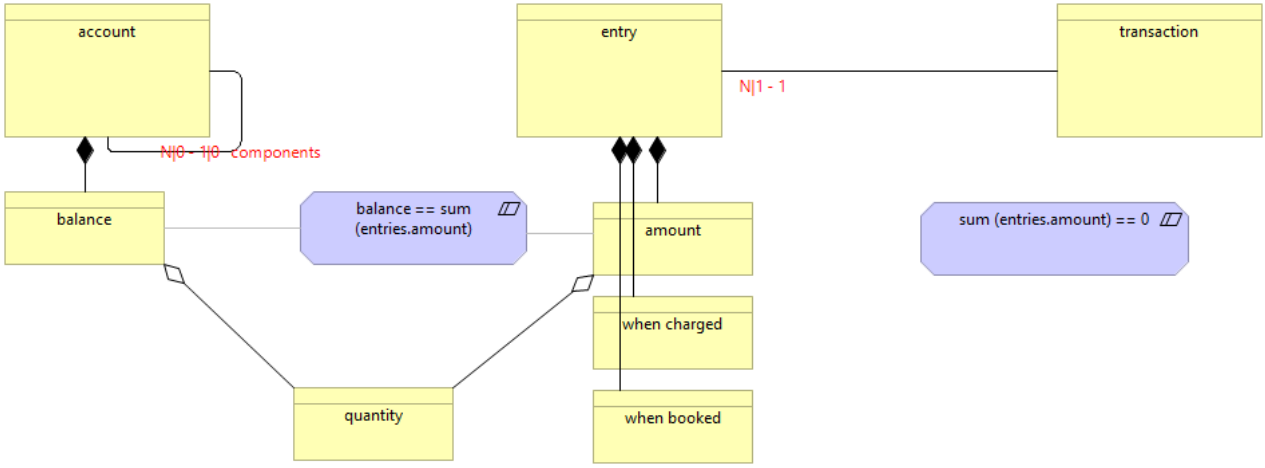
ACCOUNT


account
entry
balance
balance == sum
(entries.amount)
amount
when charged
quantity
when booked

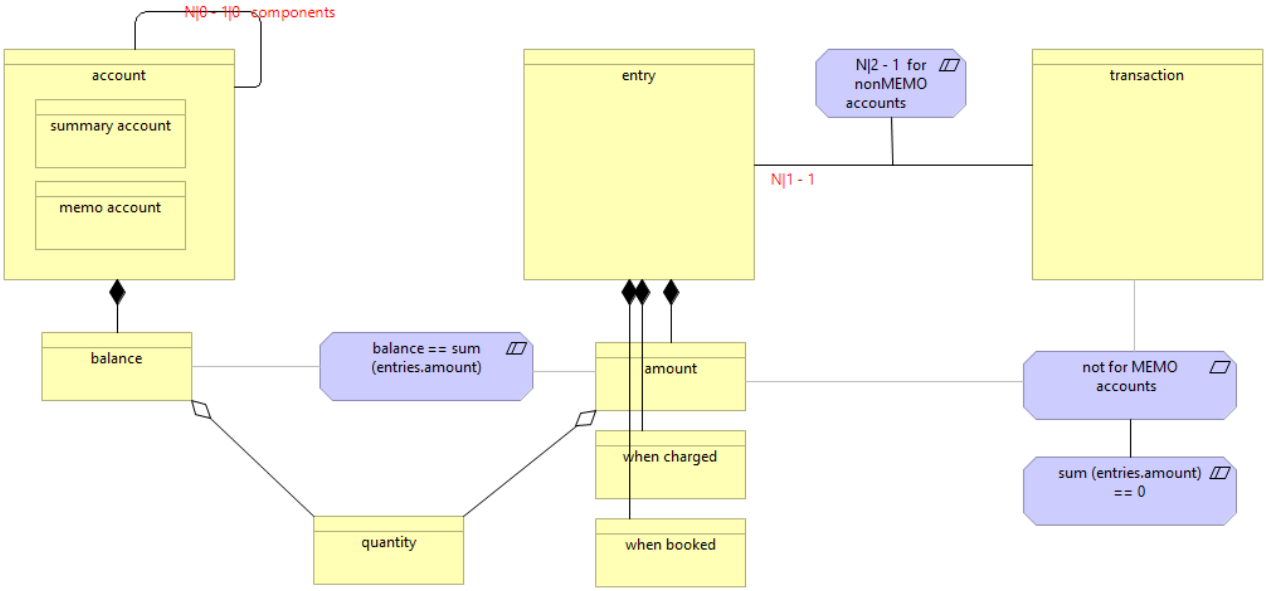
TRANSACTIONS



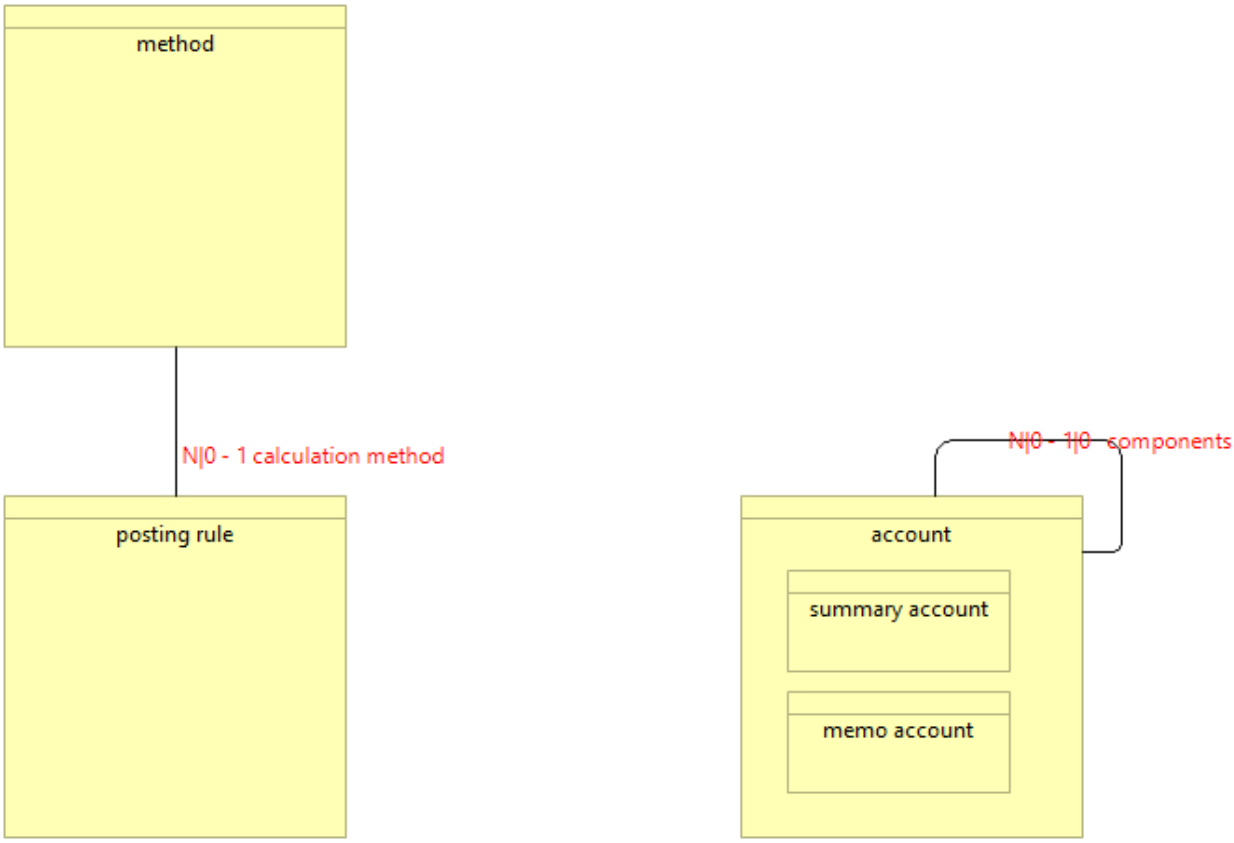
SUMMARY ACCOUNT



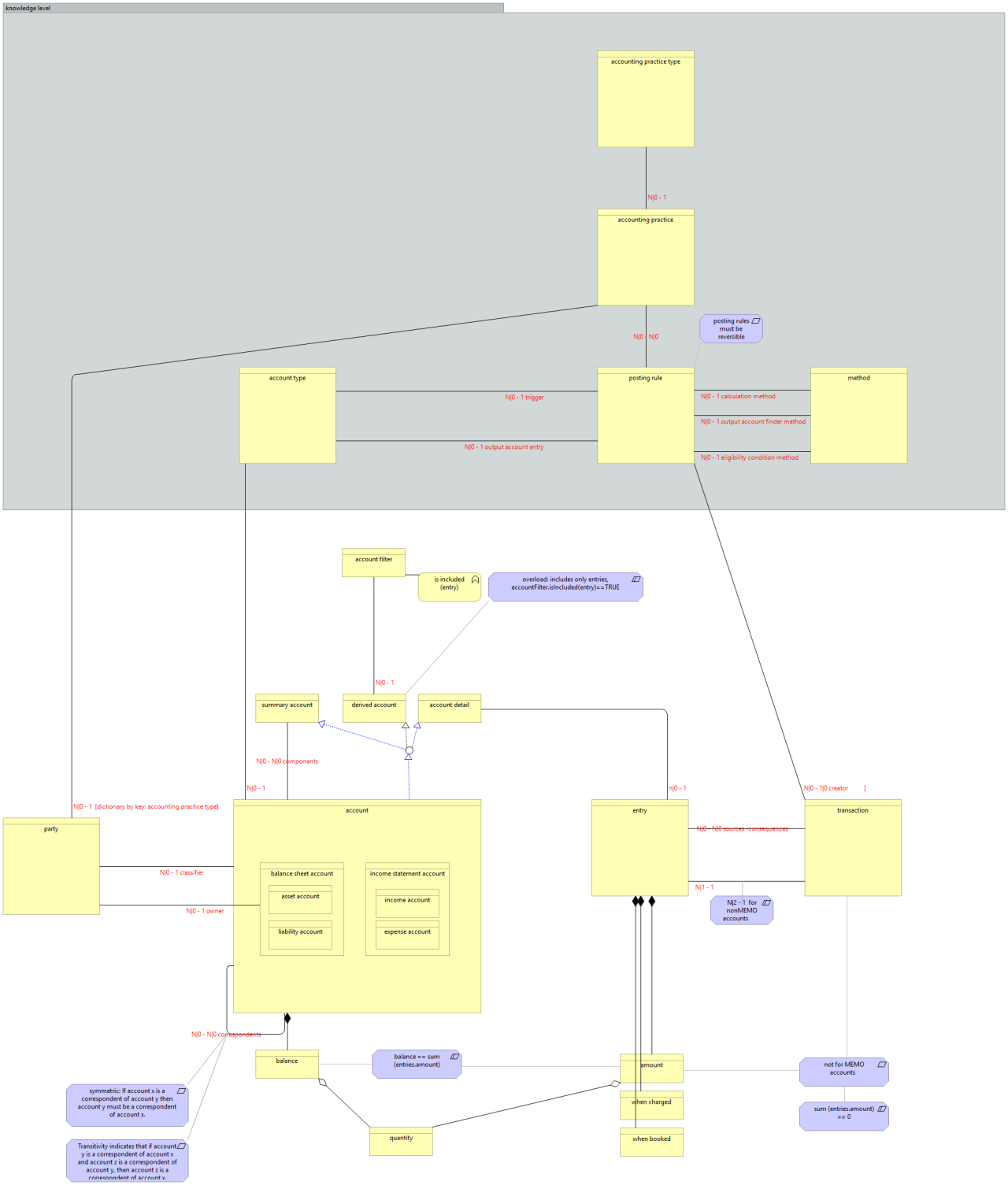
MEMO ACCOUNT



POSTING RULES

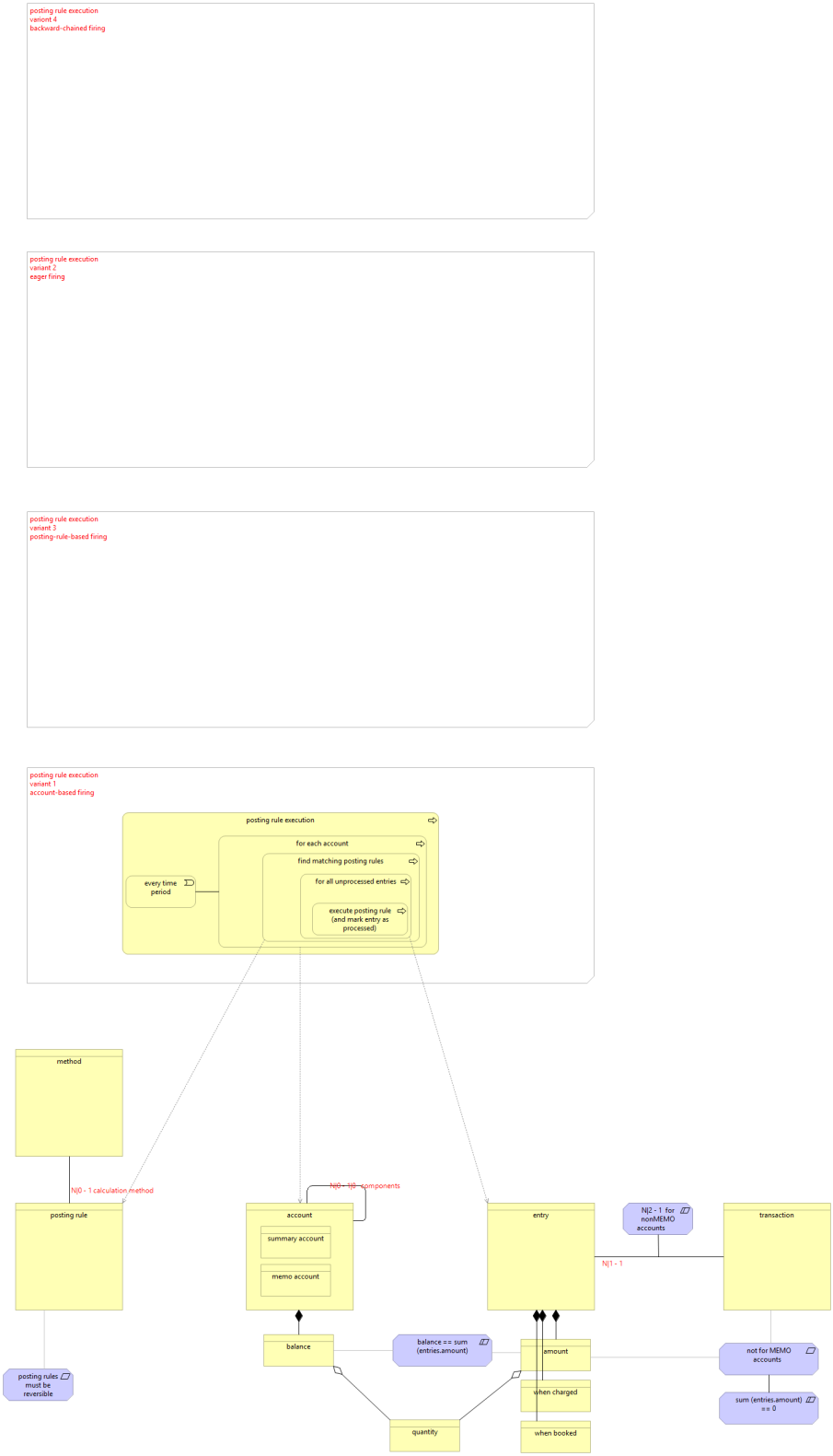


INVENTORY AND ACCOUNTING

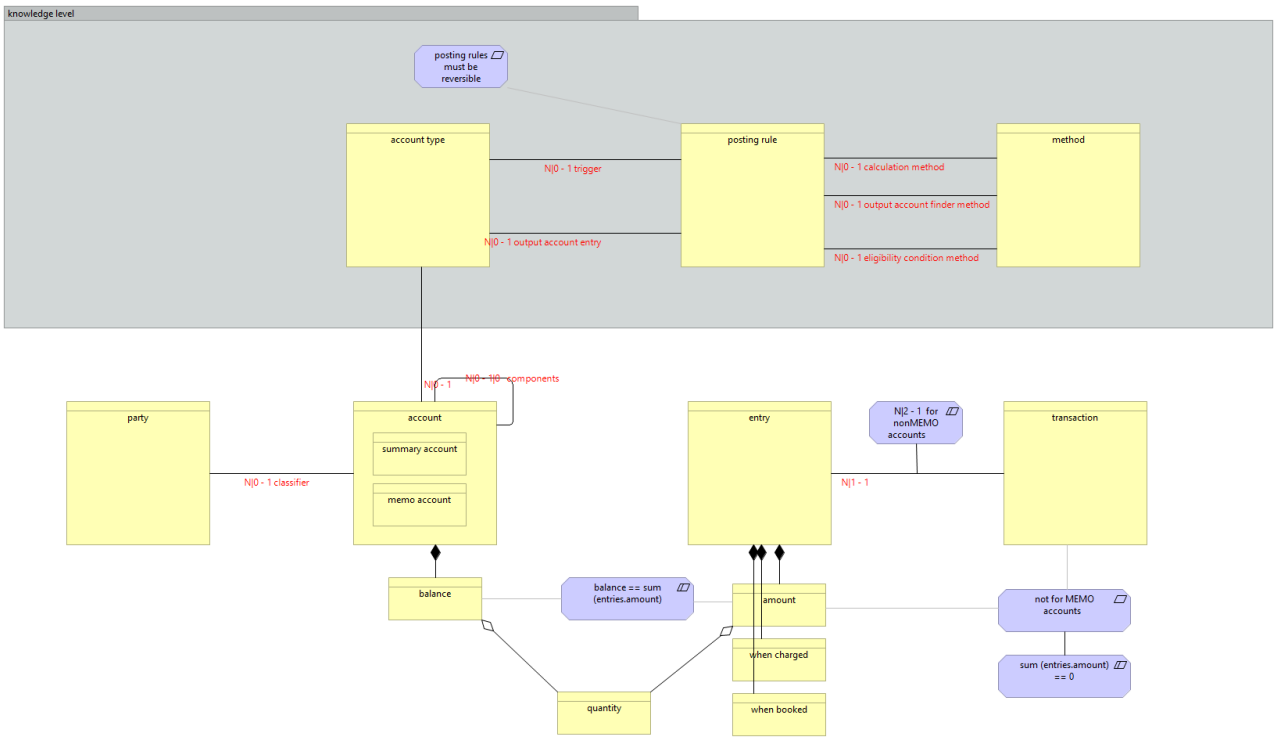


INDIVIDUAL INSTANCE METHOD

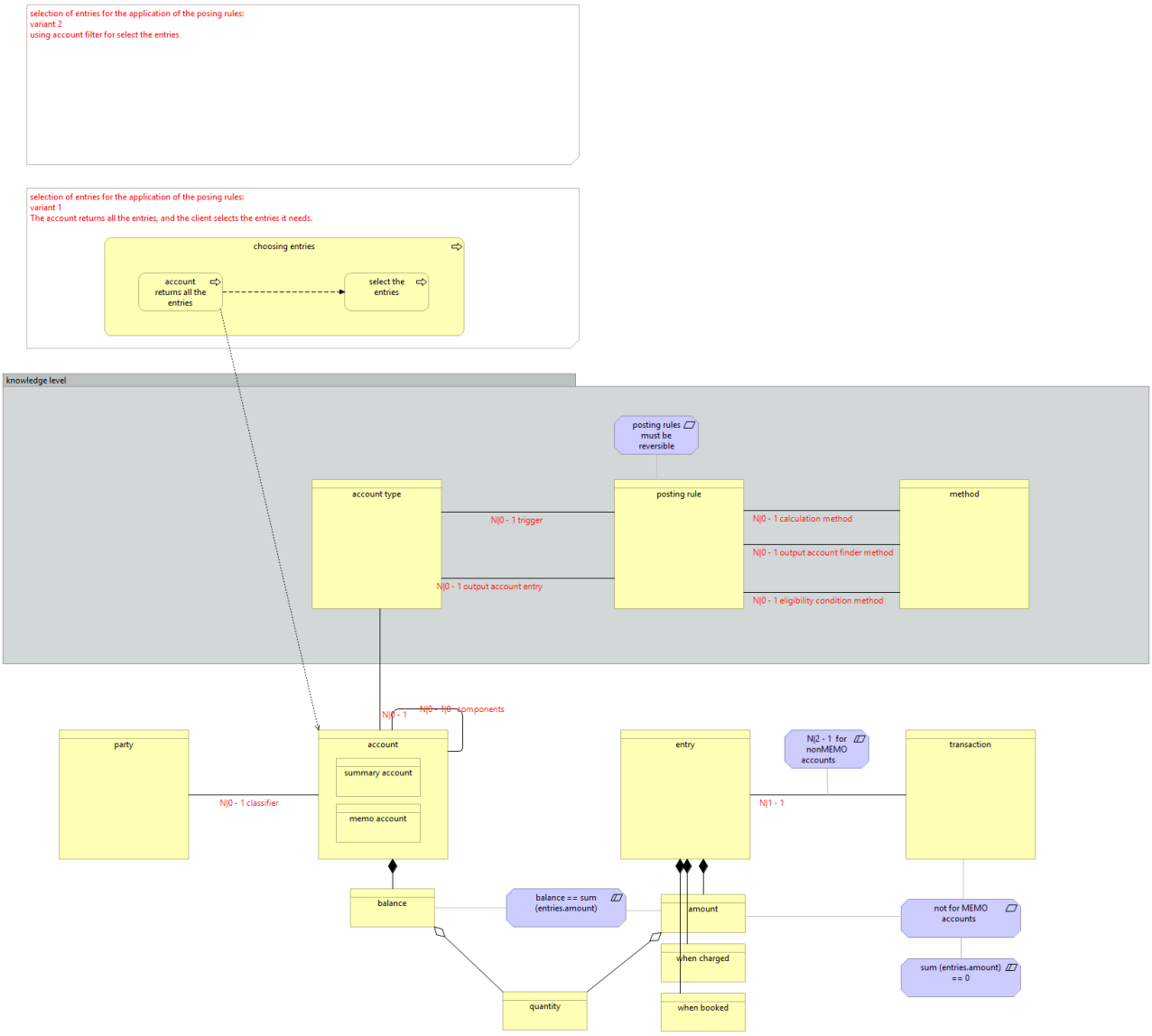
POSTING RULE EXECUTION



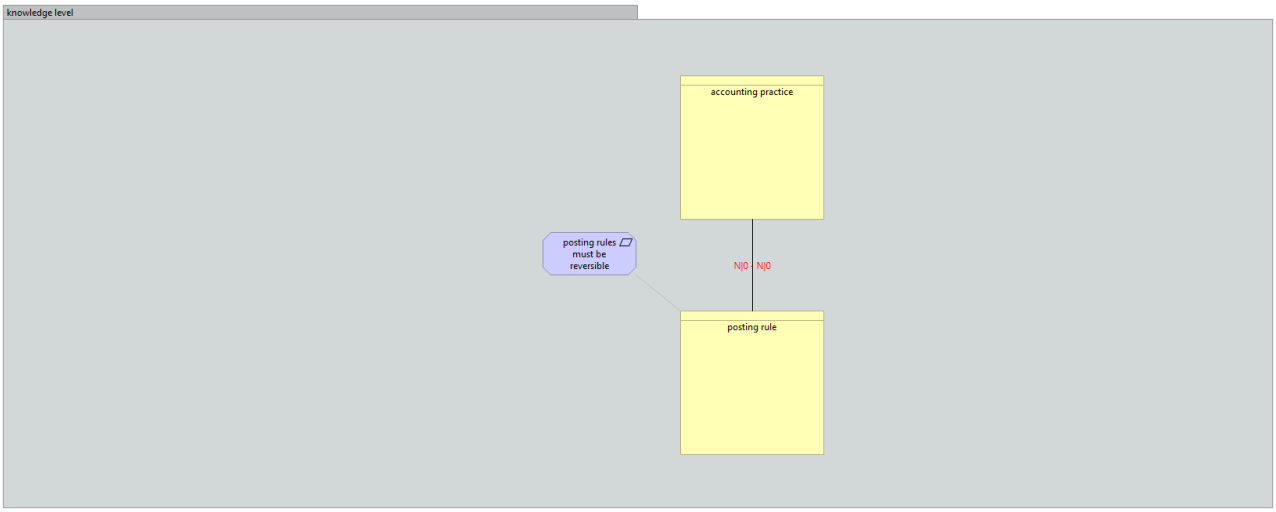
POSTING RULES FOR MANY ACCOUNTS



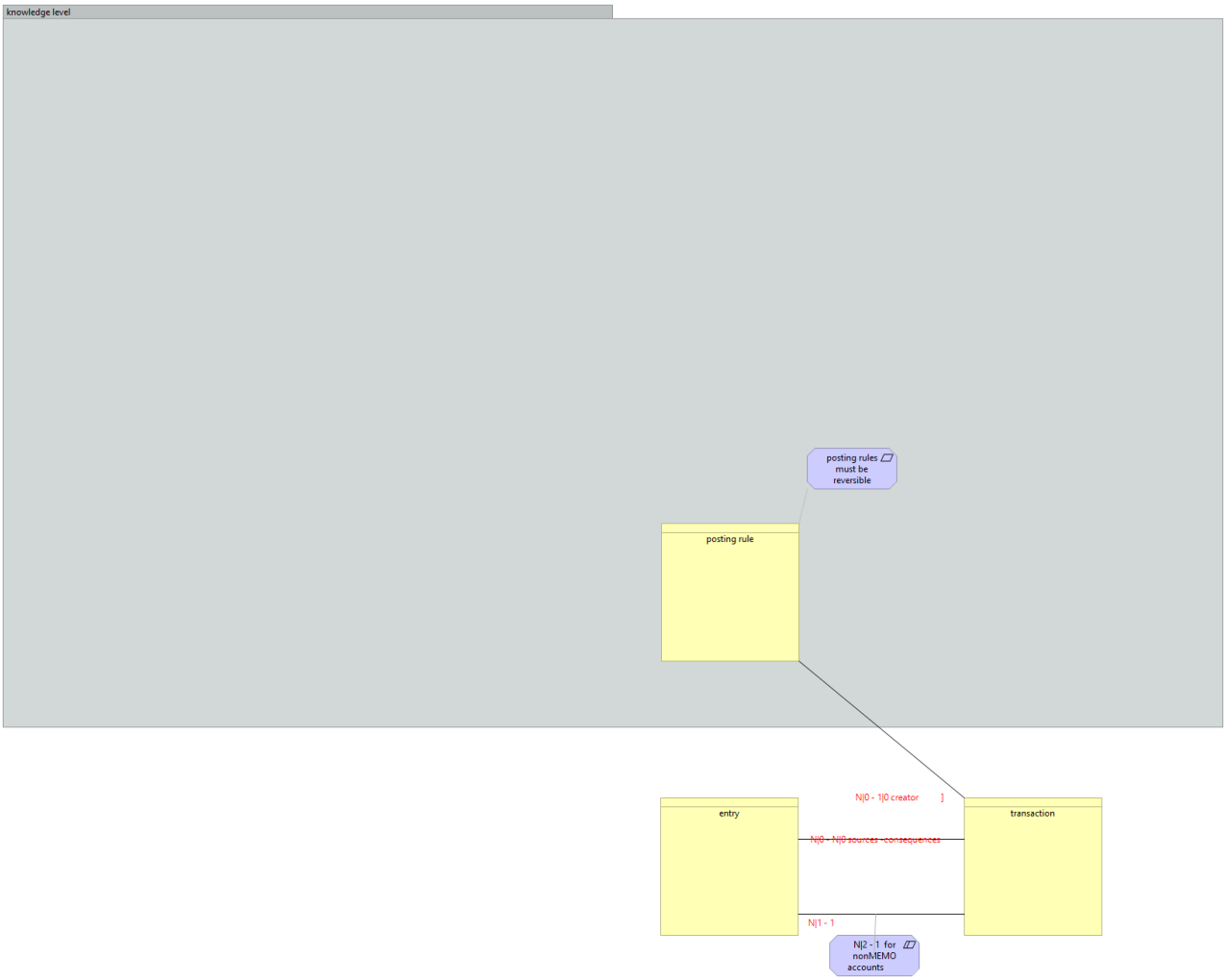
CHOOSING ENTRIES



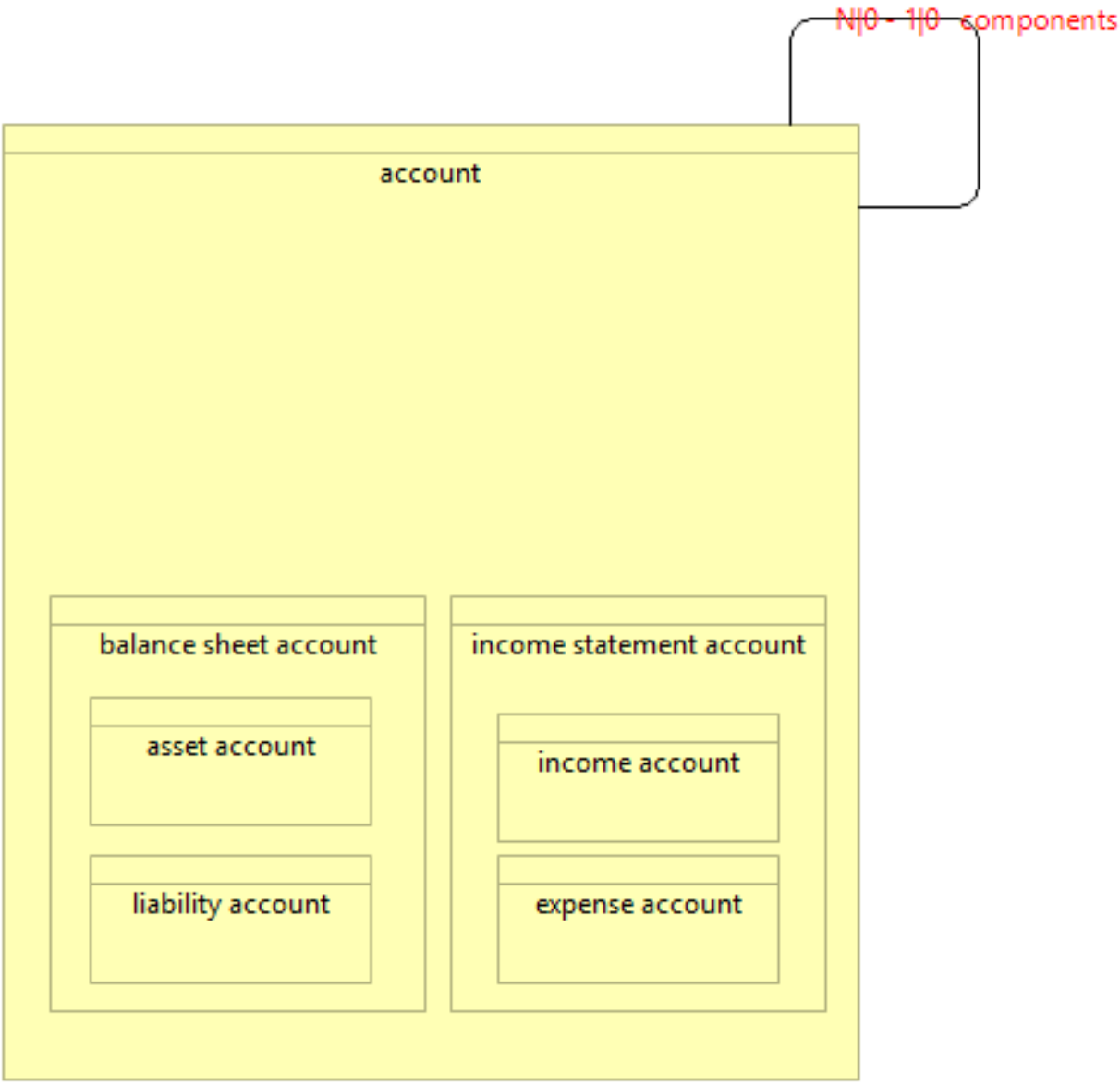
ACCOUNTING PRACTICE



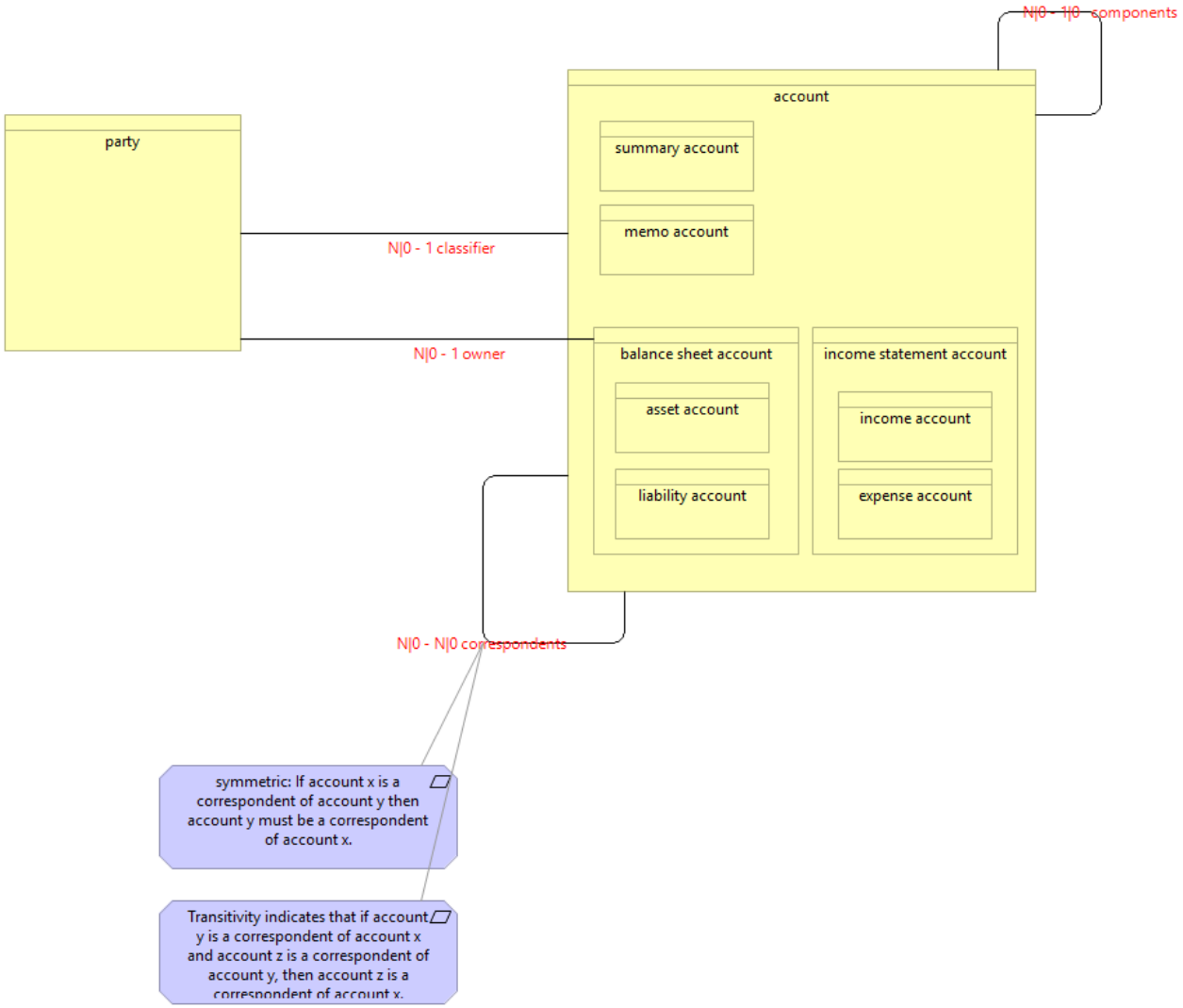
SOURCES OF AN ENTRY



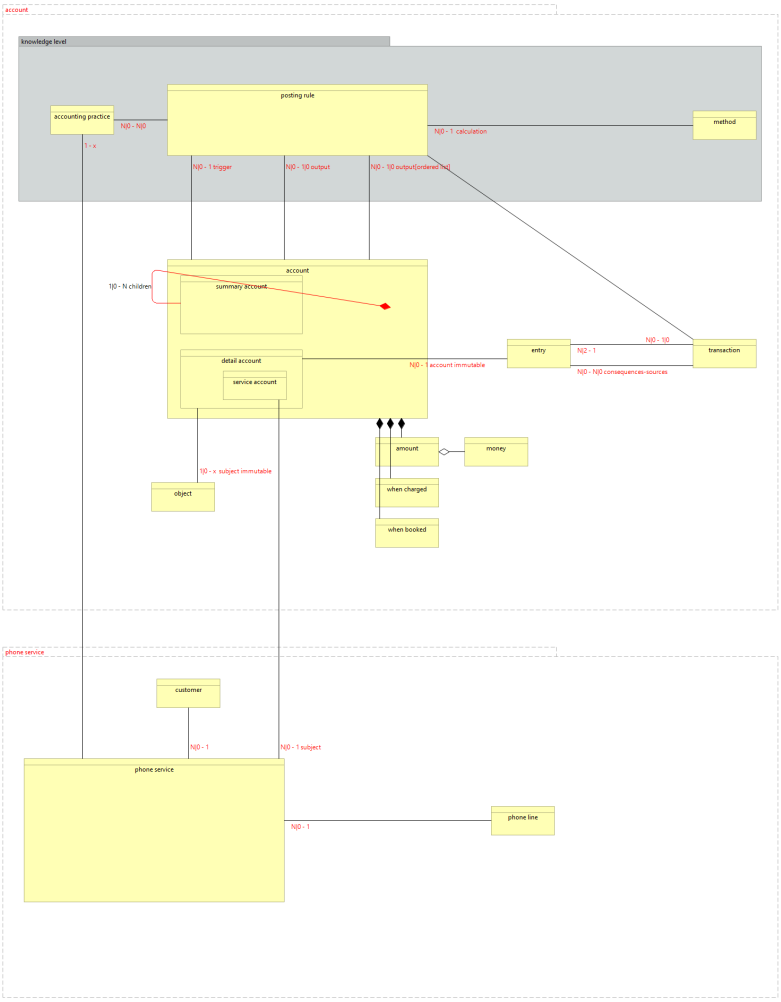
BALANCE SHEET AND INCOME STATEMENT

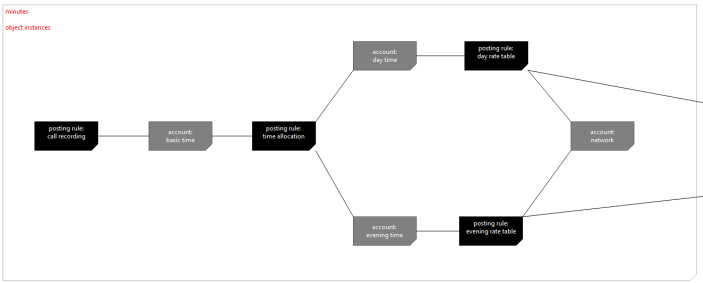


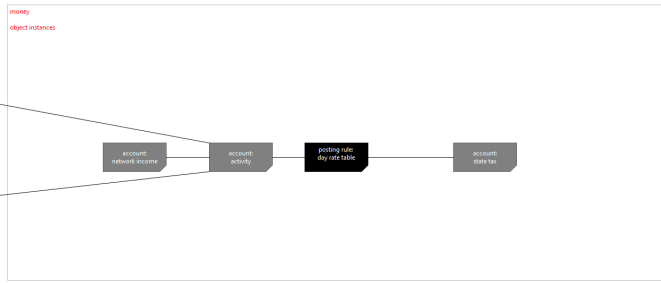
CORRESPONDING ACCOUNT



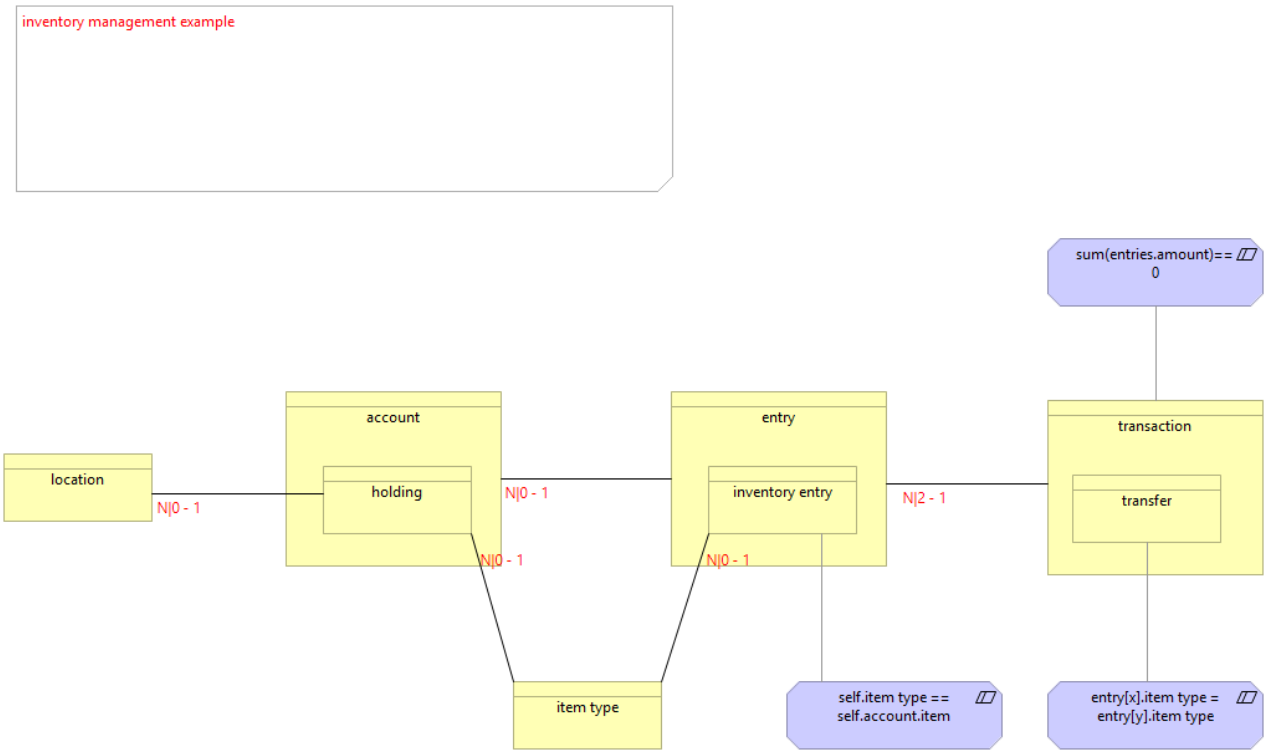
SPECIALIZED ACCOUNT MODEL (BILLING EXAMPLE)



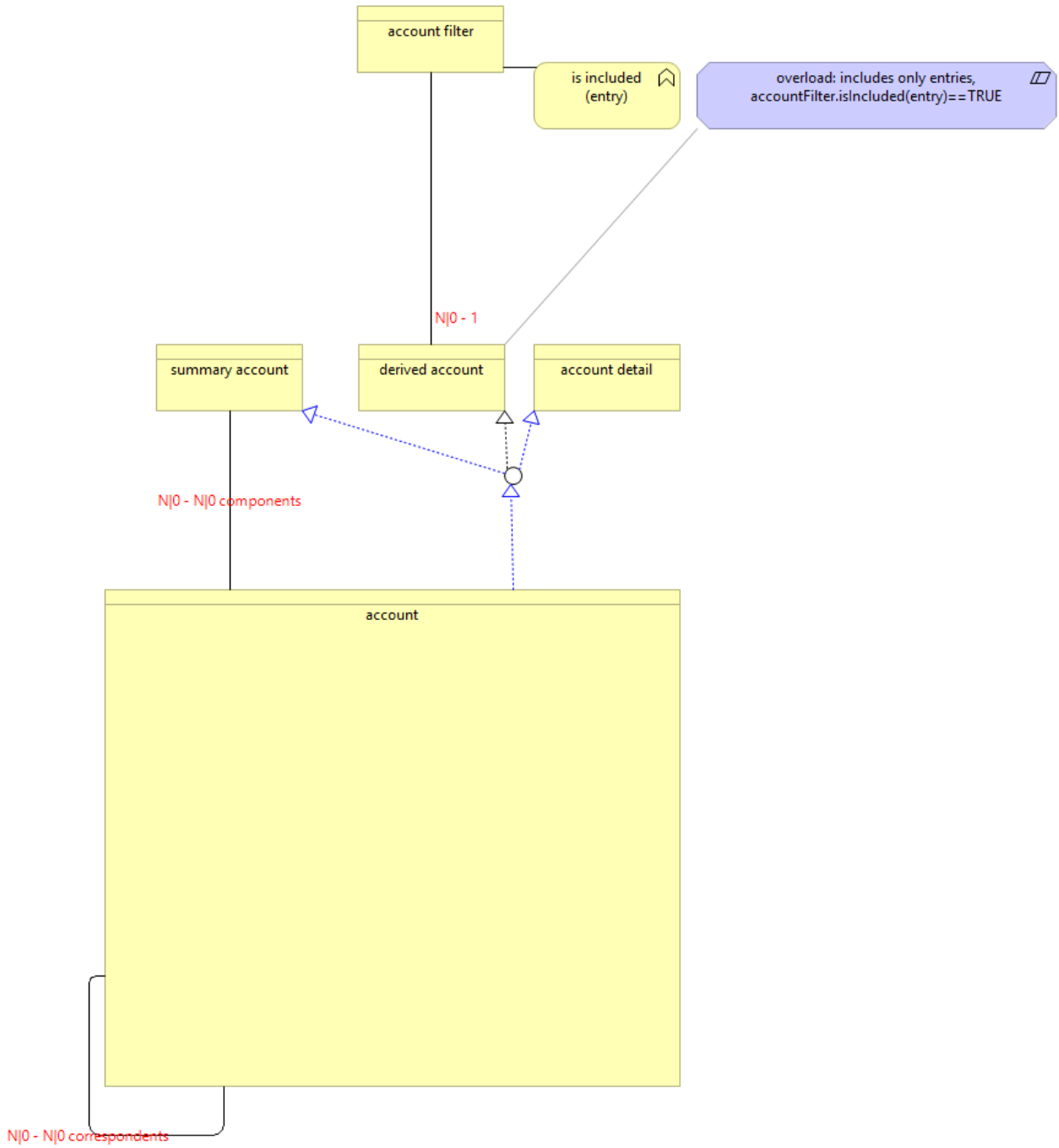




SPECIALIZED ACCOUNT MODEL (INVENTORY EXAMPLE)



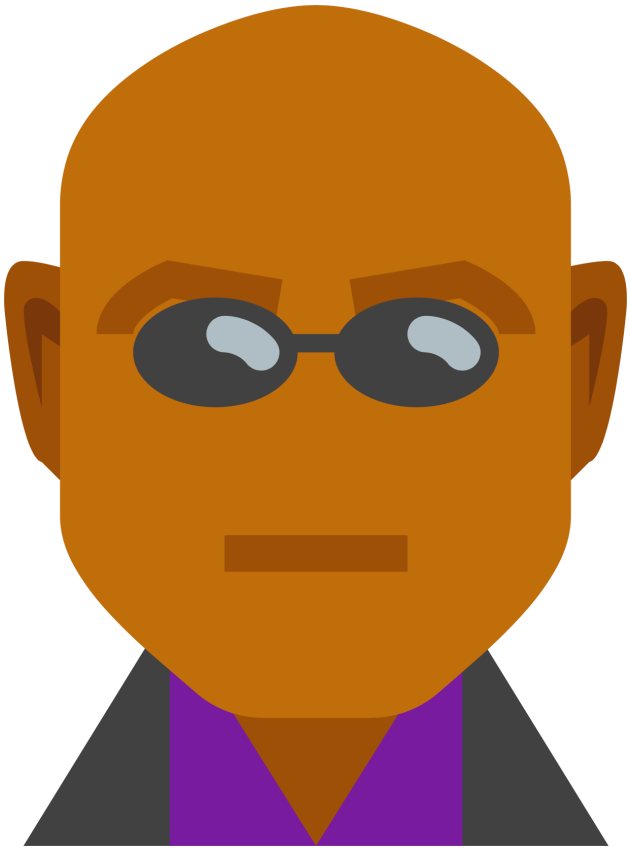
BOOKING ENTRIES TO MULTIPLE ACCOUNTS



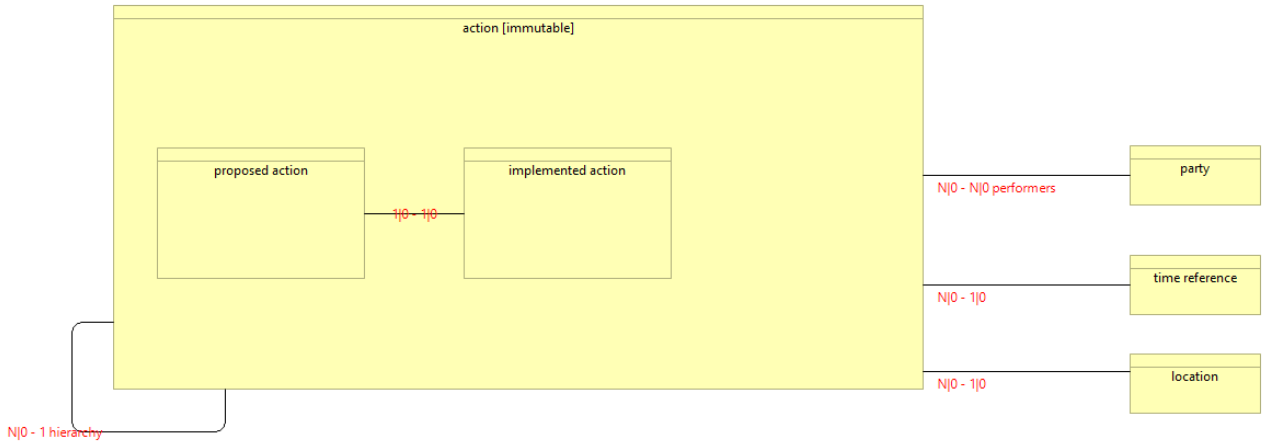
PLANNING

ANALYSIS PATTERNS

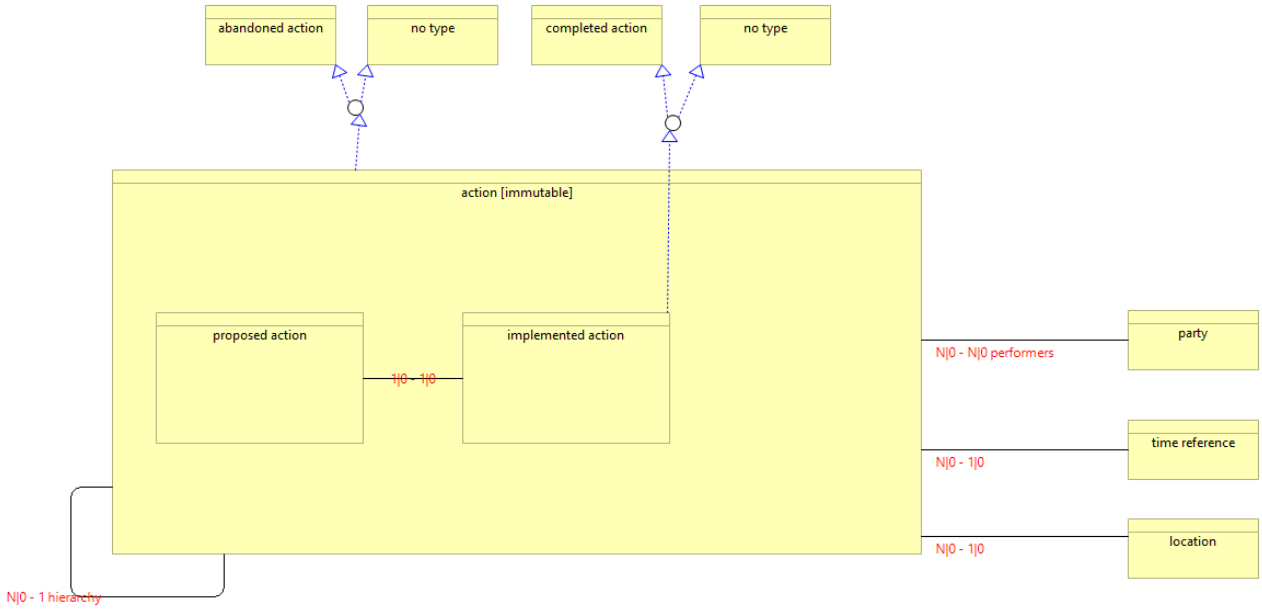
Martin Fowler



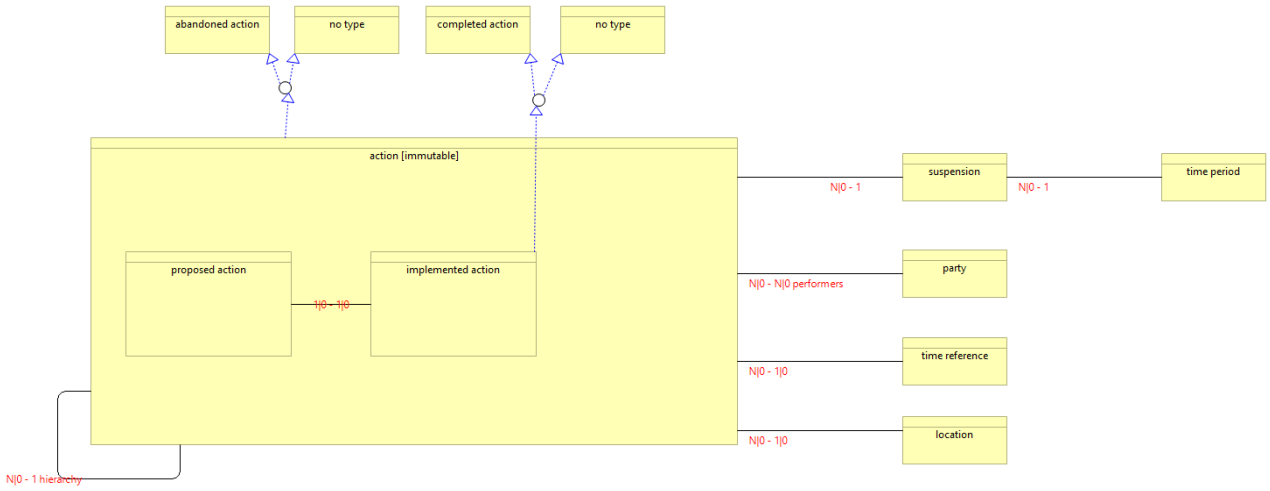
PROPOSED AND IMPLEMENTED ACTION



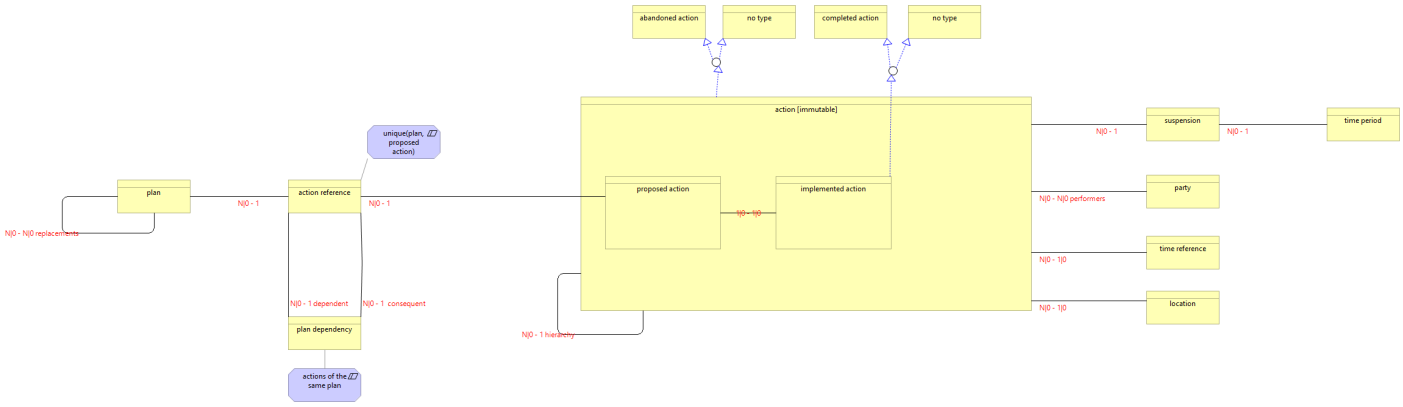
COMPLETED AND ABANDONED ACTIONS



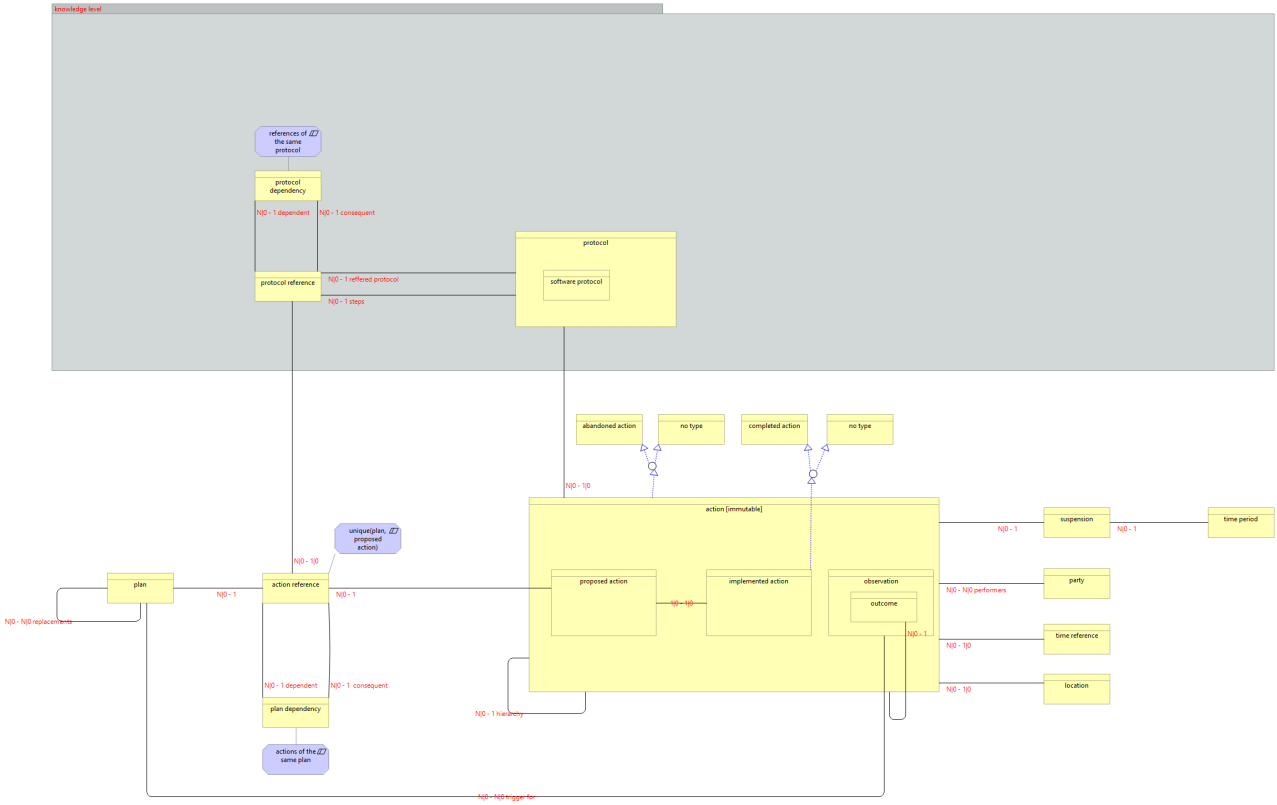
SUSPENSION


abandoned action
no type
completed action
no type
action [immutable]
proposed action
implemented action
1|0 - 1|0
N|0 - 1 hierarchy
N|0 - 1
suspension
N|0 - 1
time period
N|0 - N|0 performers
party
N|0 - 1|0
time reference
N|0 - 1|0
location

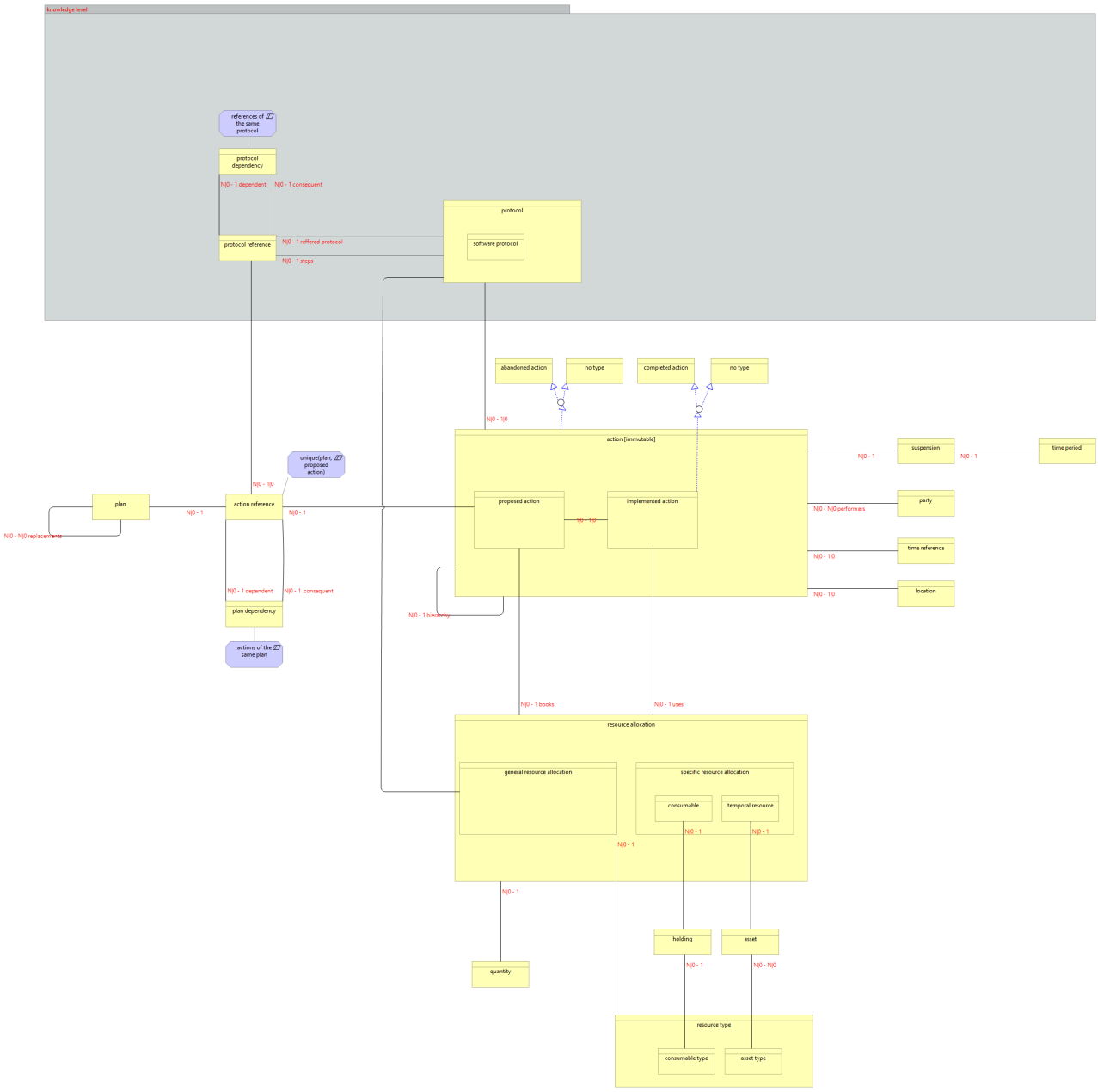
PLAN


abandoned action
no type
completed action
no type
action [immutable]
suspension
time period
N|0 - 1
N|0 - 1
unique(plan, proposed action)
plan
N|0 - 1
action reference
N|0 - 1
proposed action
implemented action
N|0 - N|0
party
N|0 - N|0 performers
N|0 - N|0 replacements
time reference
N|0 - 1|0
location
N|0 - 1|0
N|0 - 1 dependent
N|0 - 1 consequent
plan dependency
N|0 - 1 hierarchy
actions of the same plan

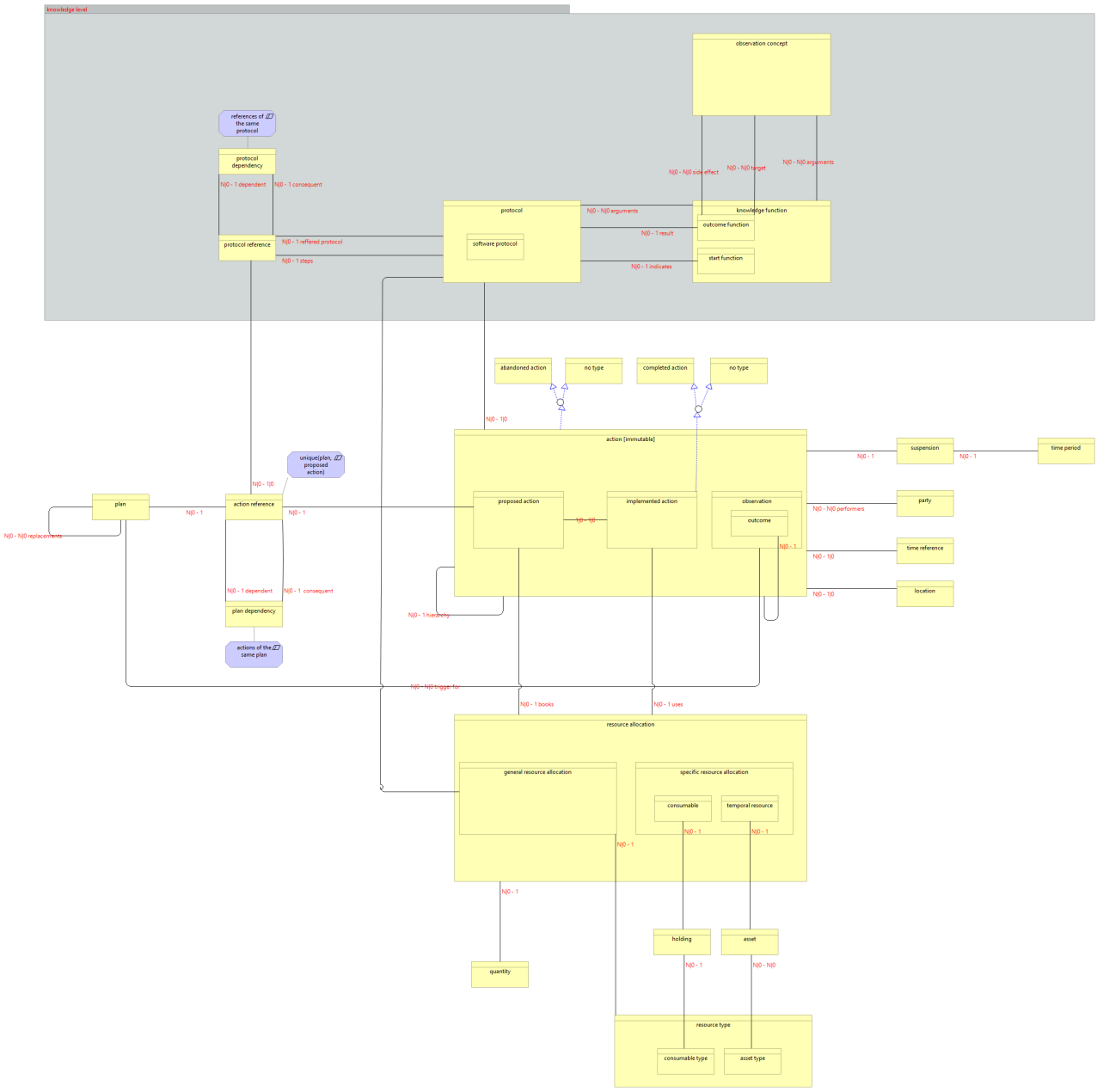
PROTOCOL


knowledge level
references of the same protocol
protocol dependency
N|0 - 1 dependent
N|0 - 1 consequent
protocol
software protocol
protocol reference
N|0 - 1 reffered protocol
N|0 - 1 steps
abandoned action
no type
completed action
no type
N|0 - 1|0
action [immutable]
suspension
N|0 - 1
N|0 - 1
time period
unique(plan, proposed action)
N|0 - 1|0
plan
N|0 - 1
action reference
N|0 - 1
proposed action
N|0 - 1|0
implemented action
observation
outcome
N|0 - N|0 performers
party
N|0 - 1
time reference
N|0 - 1|0
location
N|0 - 1|0
N|0 - N|0 replacements
N|0 - 1 dependent
N|0 - 1 consequent
plan dependency
N|0 - 1 hierarchy
actions of the same plan
N|0 - N|0 trigger for

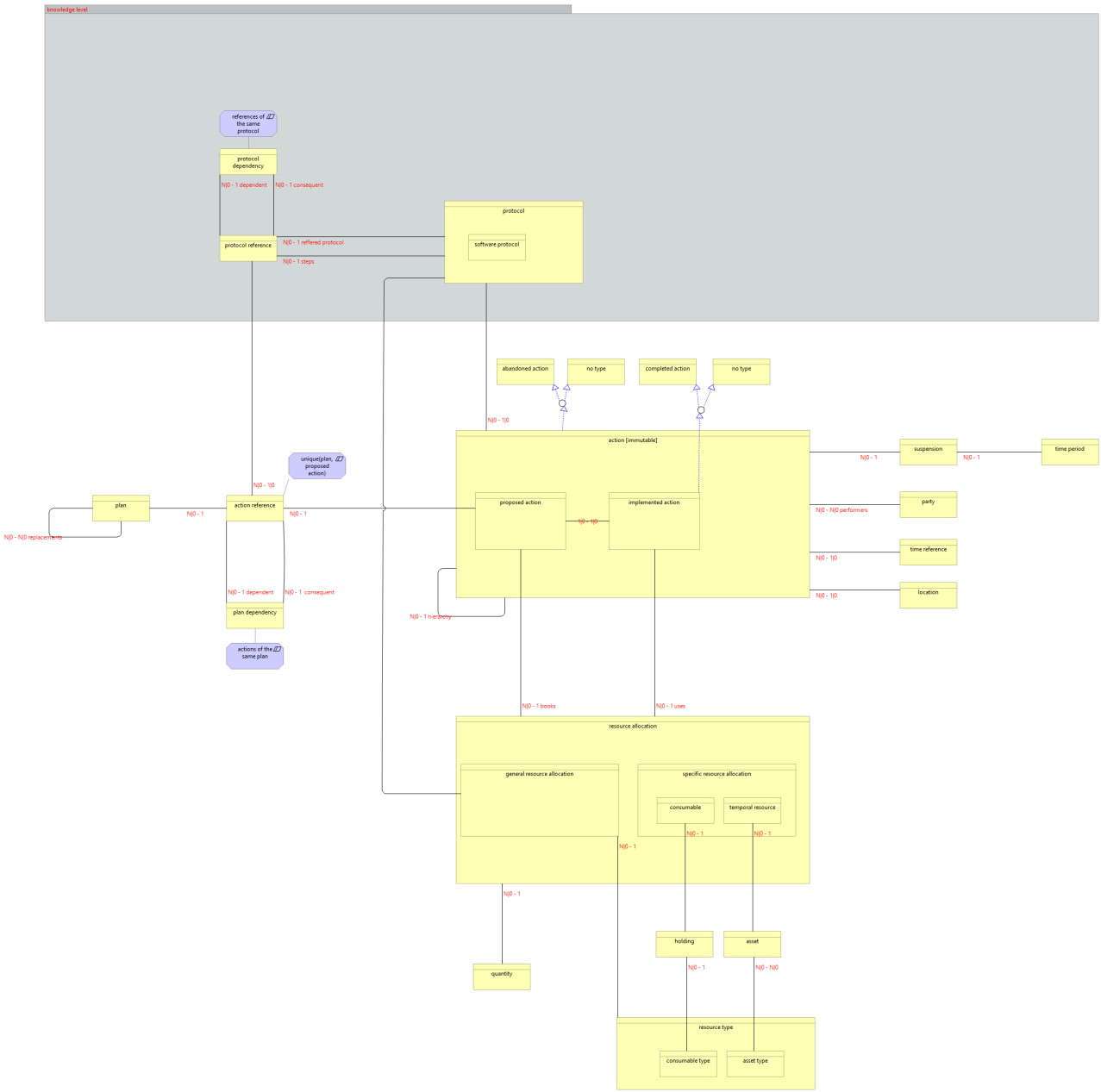
RESOURCE ALLOCATION



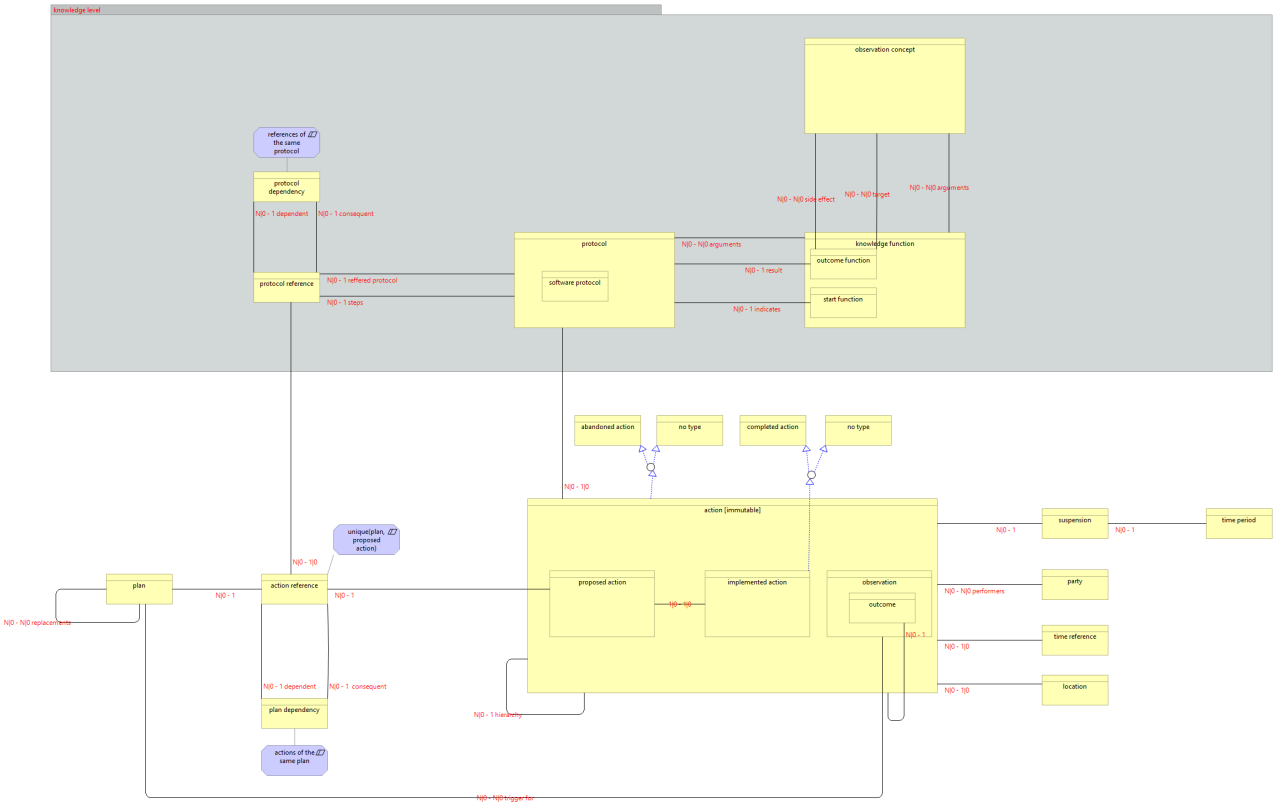
PLANNING


knowledge level
observation concept
references of the same protocol
protocol dependency
N|0 - 1 dependent
N|0 - 1 consequent
protocol reference
N|0 - 1 reffered protocol
N|0 - 1 steps
protocol
software protocol
N|0 - N|0 arguments
N|0 - 1 result
N|0 - 1 indicates
N|0 - N|0 side effect
N|0 - N|0 target
N|0 - N|0 arguments
knowledge function
outcome function
start function
abandoned action
no type
completed action
no type
N|0 - 1|0
action [immutable]
proposed action
implemented action
observation
outcome
N|0 - 1|0
N|0 - 1
N|0 - 1
suspension
N|0 - 1
time period
N|0 - N|0 performers
party
N|0 - 1|0
time reference
N|0 - 1|0
location
uniqueplan, proposed action)
N|0 - 1|0
plan
N|0 - 1
action reference
N|0 - 1
N|0 - N|0 replacements
N|0 - 1 dependent
N|0 - 1 consequent
plan dependency
actions of the same plan
N|0 - 1 hierarchy
N|0 - N|0 trigger for
N|0 - 1 books
N|0 - 1 uses
resource allocation
general resource allocation
specific resource allocation
consumable
temporal resource
N|0 - 1
N|0 - 1
N|0 - 1
N|0 - 1
quantity
holding
asset
N|0 - 1
N|0 - N|0
resource type
consumable type
asset type

PLANNING (NO OUTCOME)



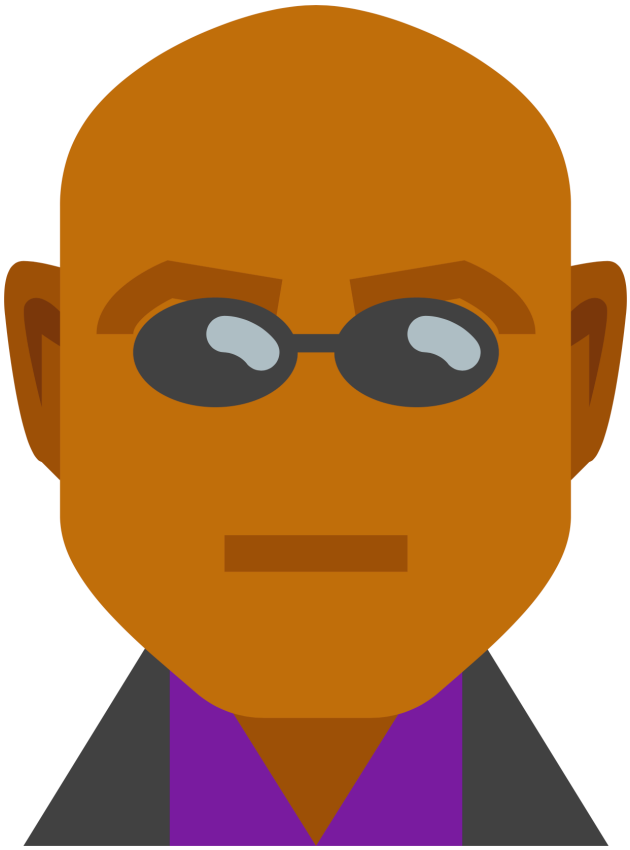
OUTCOME AND START FUNCTIONS



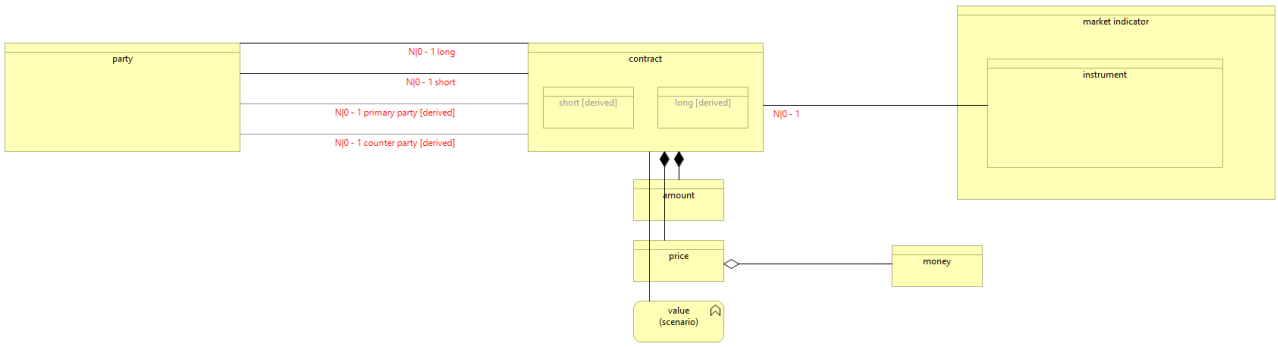
TRADING

ANALYSIS PATTERNS

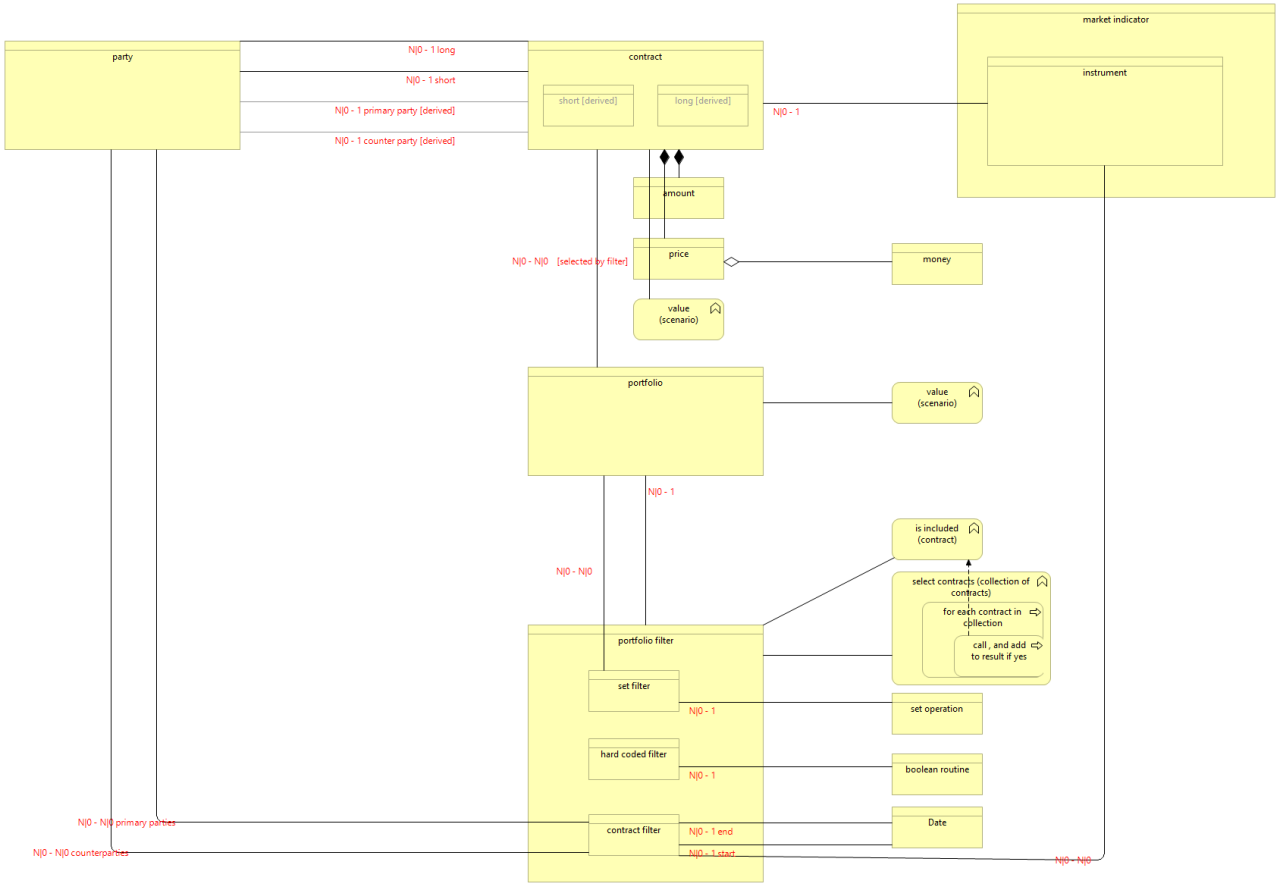
Martin Fowler



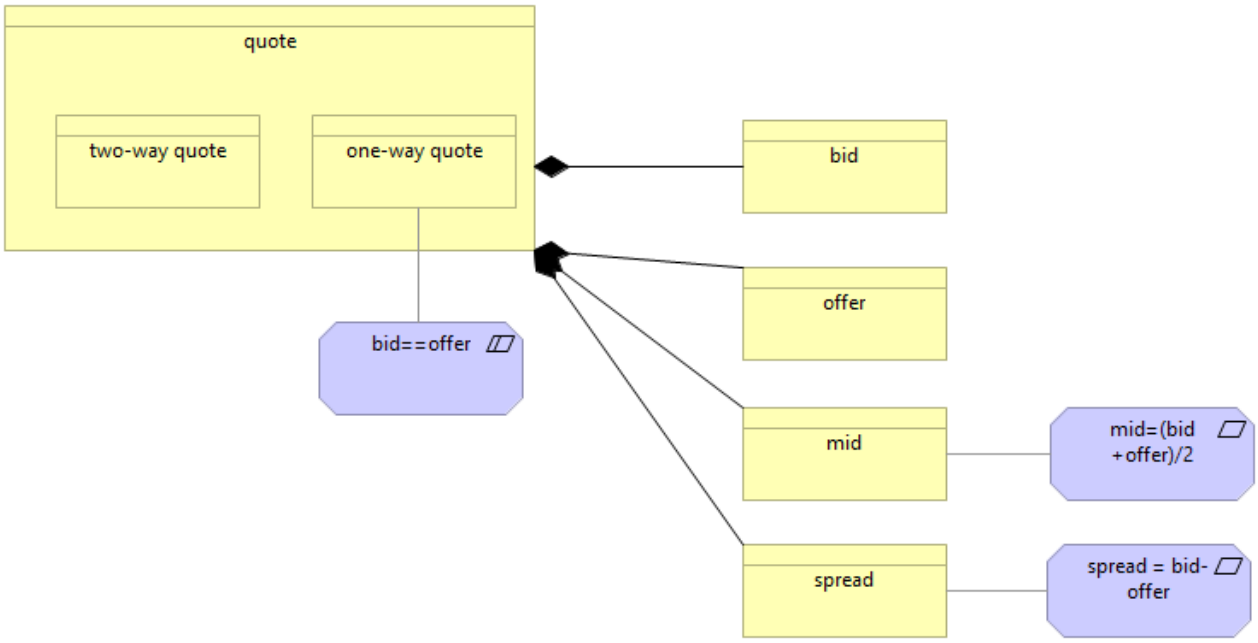
CONTRACT


party
N|0 - 1 long
N|0 - 1 short
N|0 - 1 primary party [derived]
N|0 - 1 counter party [derived]
contract
short [derived]
long [derived]
N|0 - 1
market indicator
instrument
amount
price
money
value
(scenario)

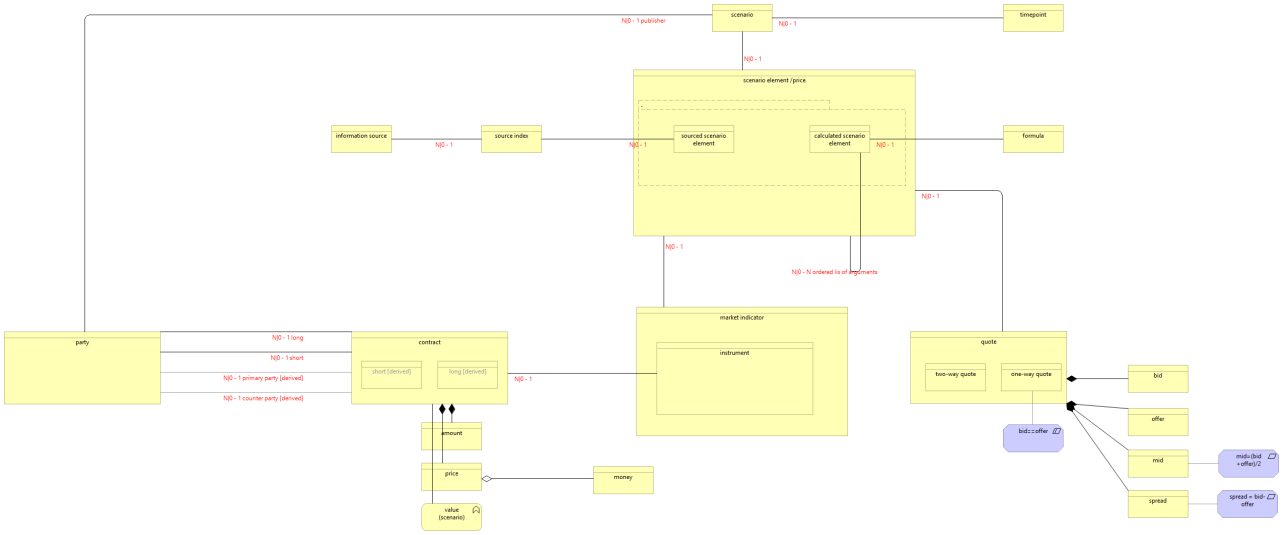
PORTFOLIO


party
N|0 - 1 long
N|0 - 1 short
N|0 - 1 primary party [derived]
N|0 - 1 counter party [derived]
contract
short [derived]
long [derived]
N|0 - 1
market indicator
instrument
amount
price
N|0 - N|0 [selected by filter]
money
value (scenario)
portfolio
value (scenario)
N|0 - 1
N|0 - N|0
is included (contract)
select contracts (collection of contracts)
for each contract in collection
call , and add to result if yes
portfolio filter
set filter
N|0 - 1
set operation
hard coded filter
N|0 - 1
boolean routine
N|0 - N|0 primary parties
N|0 - N|0 counterparties
contract filter
N|0 - 1 end
N|0 - 1 start
Date
N|0 - N|0

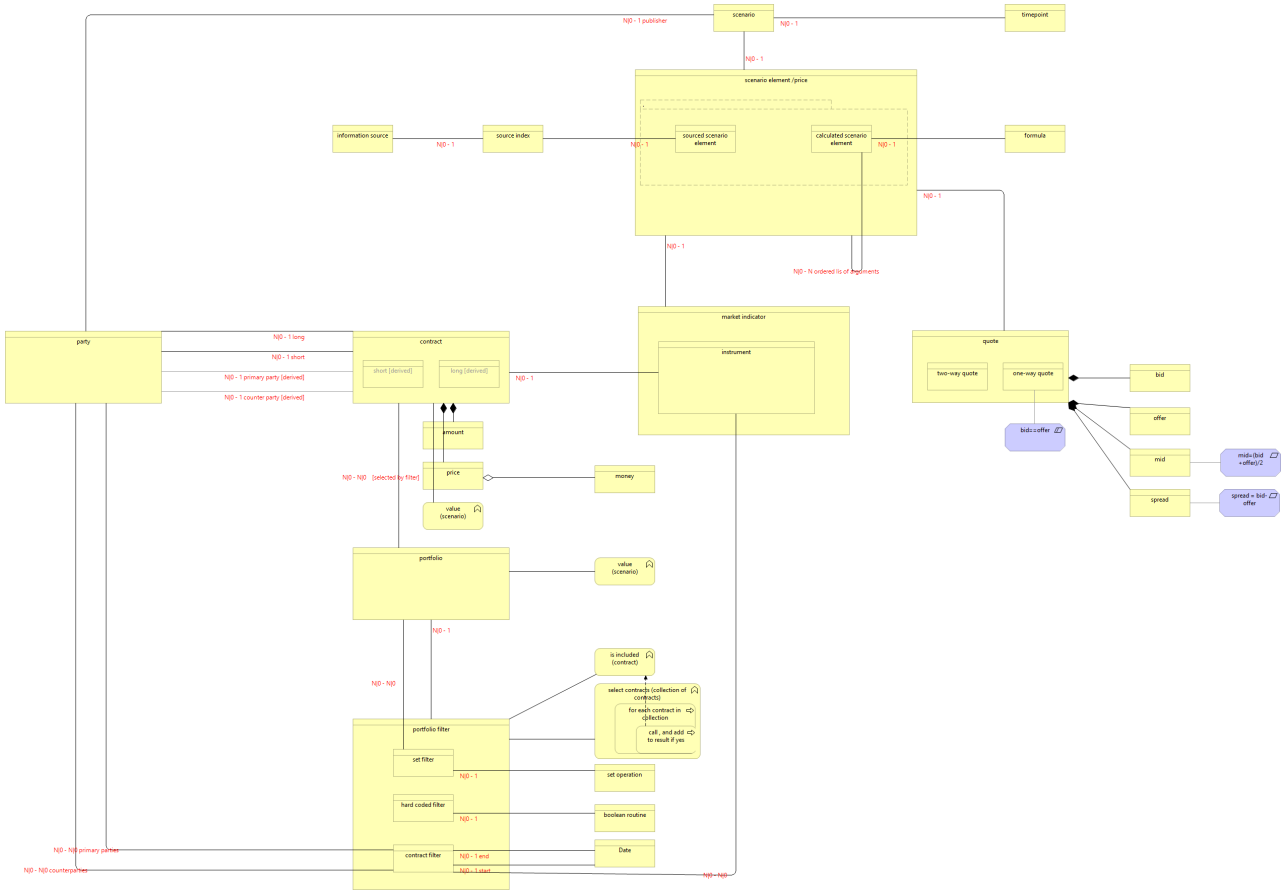
QUOTE


quote
two-way quote
one-way quote
bid
offer
mid
spread
bid==offer
mid=(bid +offer)/2
spread = bid- offer

SCENARIO


scenario
timepoint
N|0 - 1 publisher
N|0 - 1
N|0 - 1
scenario element /price
information source
source index
sourced scenario element
calculated scenario element
formula
N|0 - 1
N|0 - 1
N|0 - 1
N|0 - 1
N|0 - N ordered is of instruments
market indicator
instrument
party
N|0 - 1 long
N|0 - 1 short
N|0 - 1 primary party [derived]
N|0 - 1 counter party [derived]
contract
short [derived]
long [derived]
N|0 - 1
quote
two-way quote
one-way quote
bid
offer
bid::offer
mid
mid=(bid +offer)/2
spread
spread = bid- offer
amount
price
money
value (scenario)

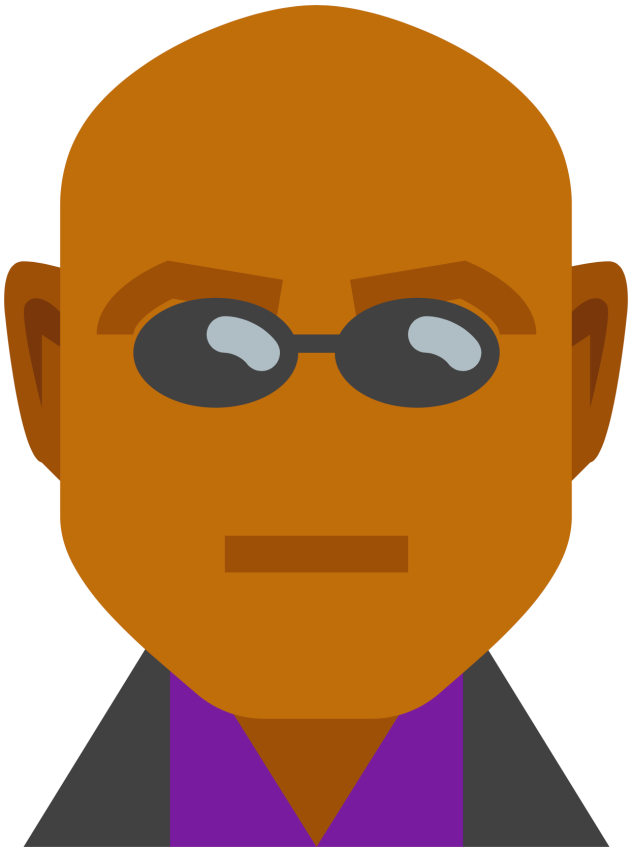
TRADING


scenario
timepoint
N|0 - 1 publisher
N|0 - 1
N|0 - 1
scenario element /price
information source
source index
sourced scenario element
calculated scenario element
formula
N|0 - 1
N|0 - 1
N|0 - 1
N|0 - 1
N|0 - N ordered lis of fragments
market indicator
instrument
party
N|0 - 1 long
N|0 - 1 short
N|0 - 1 primary party [derived]
N|0 - 1 counter party [derived]
contract
short [derived]
long [derived]
N|0 - 1
quote
two-way quote
one-way quote
bid
offer
bid::offer
mid
mid=(bid +offer)/2
spread
spread = bid- offer
amount
price
N|0 - N|0 [selected by filter]
money
value (scenario)
portfolio
value (scenario)
N|0 - 1
N|0 - N|0
is included (contract)
select contracts (collection of contracts)
for each contract in collection
call , and add to result if yes
portfolio filter
set filter
N|0 - 1
set operation
hard coded filter
N|0 - 1
boolean routine
N|0 - N|0 primary partner
N|0 - N|0 counterparties
contract filter
N|0 - 1 end
N|0 - 1 start
Date
N|0 - N|0

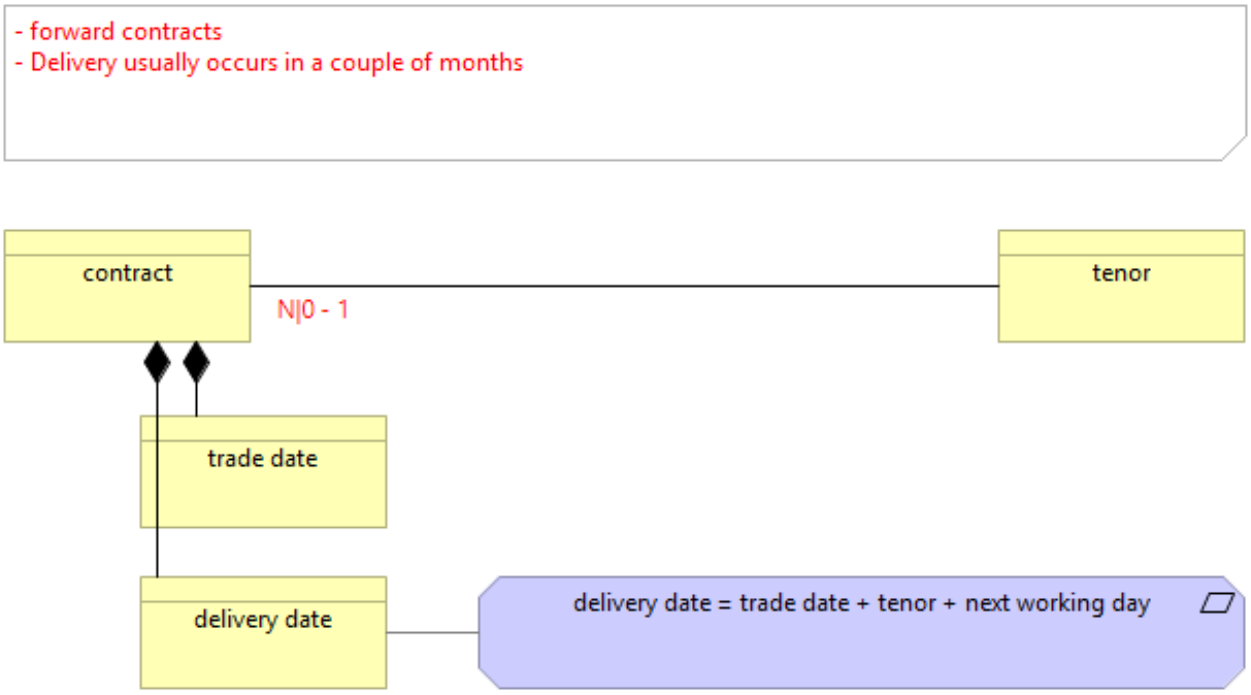
DERIVATIVE CONTRACTS

ANALYSIS PATTERNS

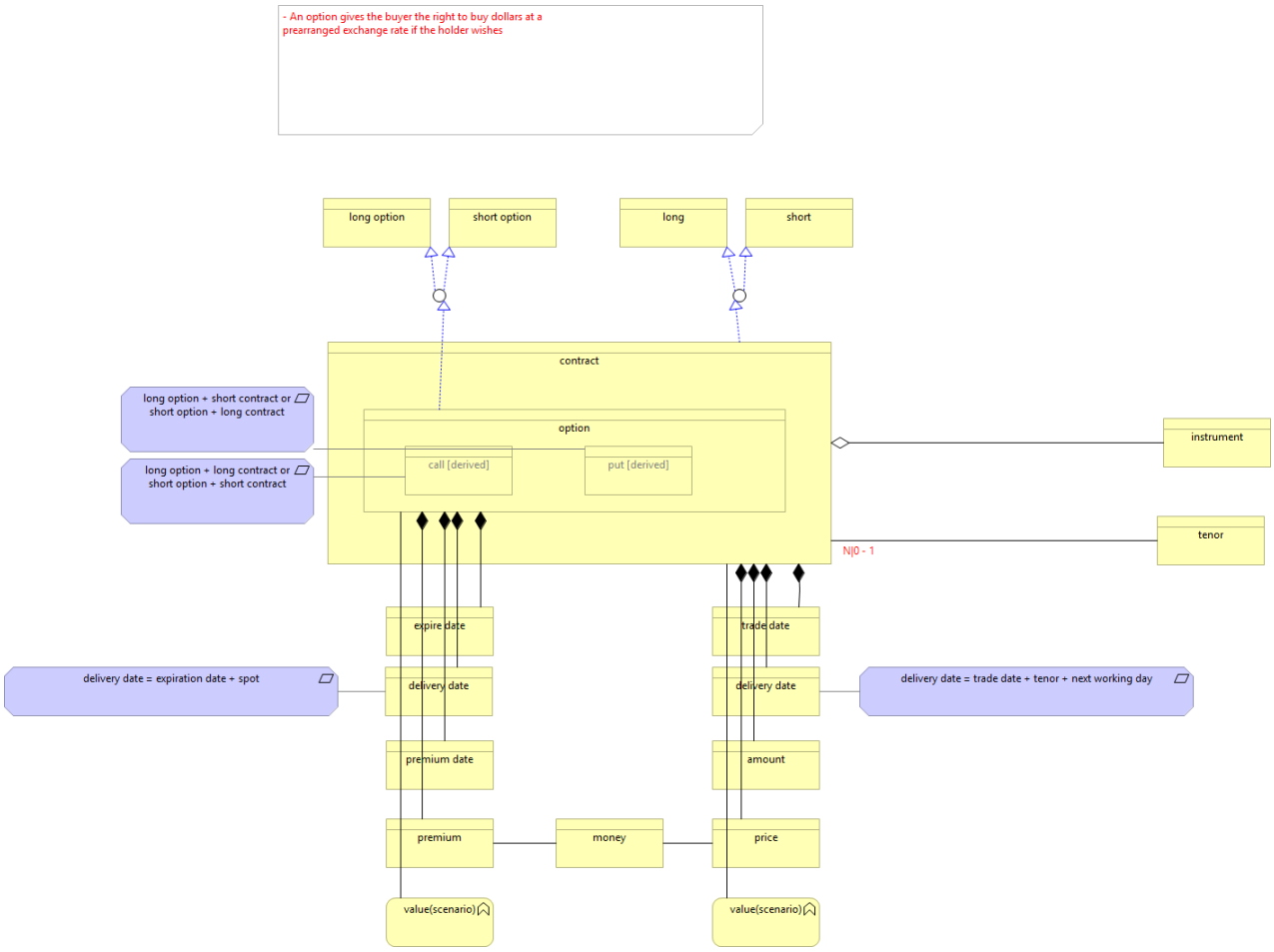
Martin Fowler

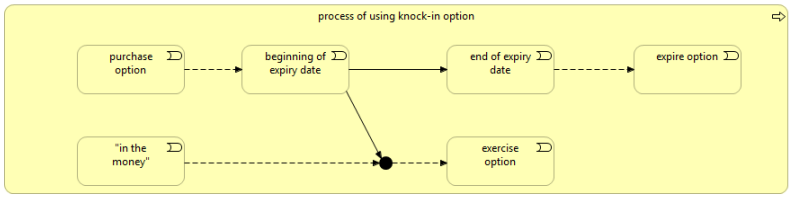


FORWARD CONTRACTS

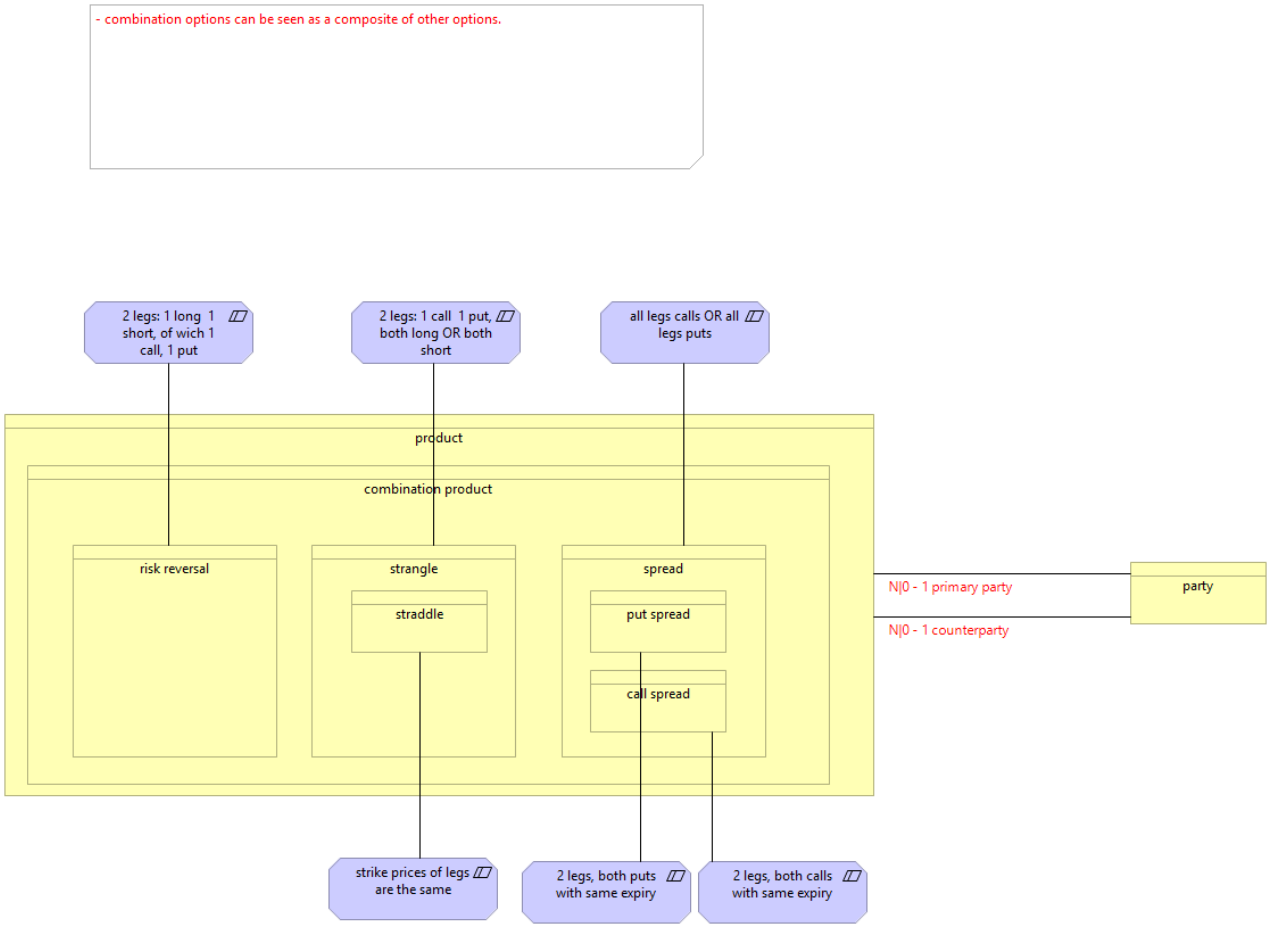


OPTIONS

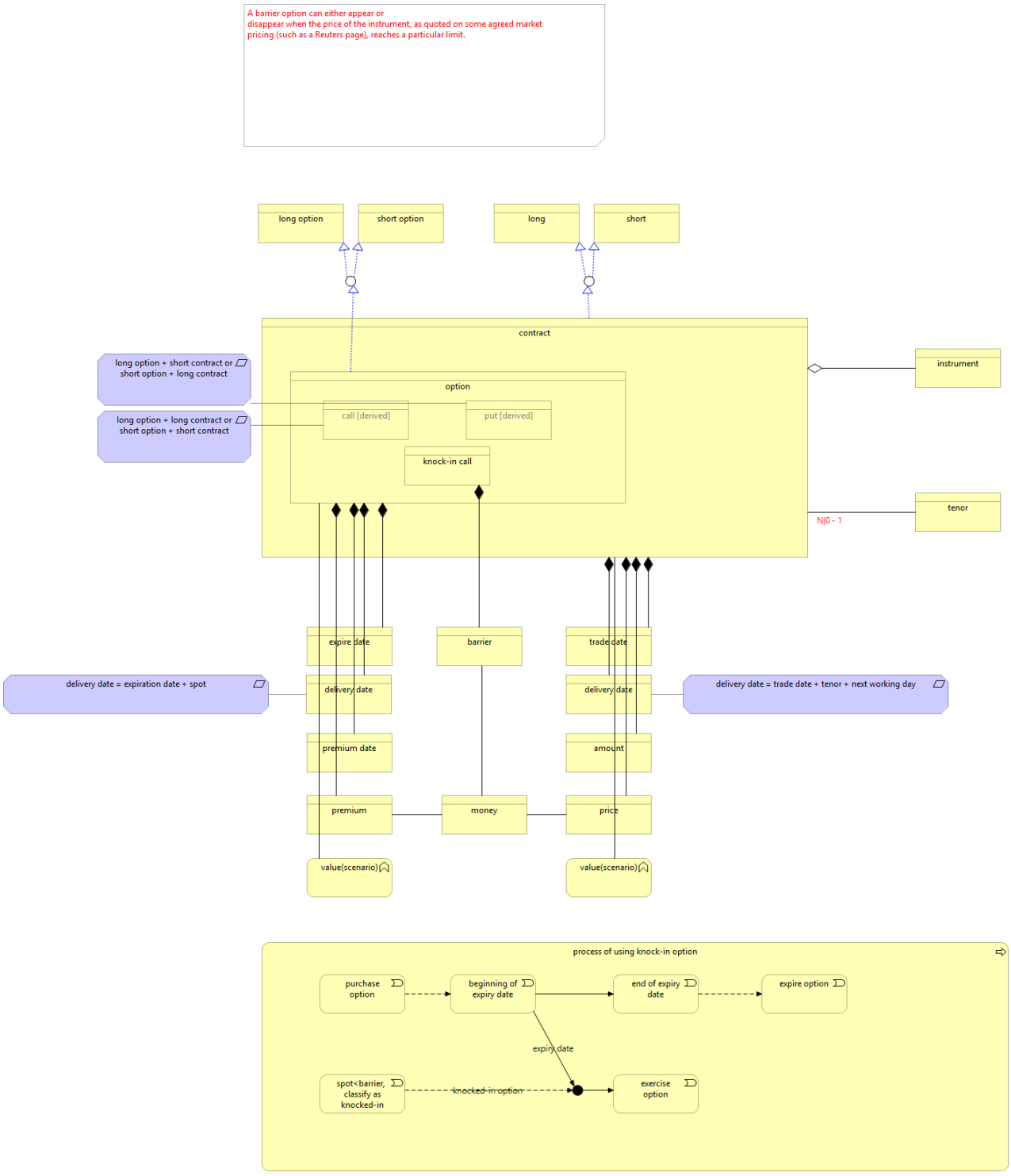

- An option gives the buyer the right to buy dollars at a prearranged exchange rate if the holder wishes
long option
short option
long
short
contract
long option + short contract or short option + long contract
long option + long contract or short option + short contract
option
call [derived]
put [derived]
instrument
tenor
N|0 - 1
expire date
trade date
delivery date = expiration date + spot
delivery date
delivery date
delivery date = trade date + tenor + next working day
premium date
amount
premium
money
price
value(scenario)
value(scenario)


process of using knock-in option
purchase option
beginning of expiry date
end of expiry date
expire option
"in the money"
exercise option

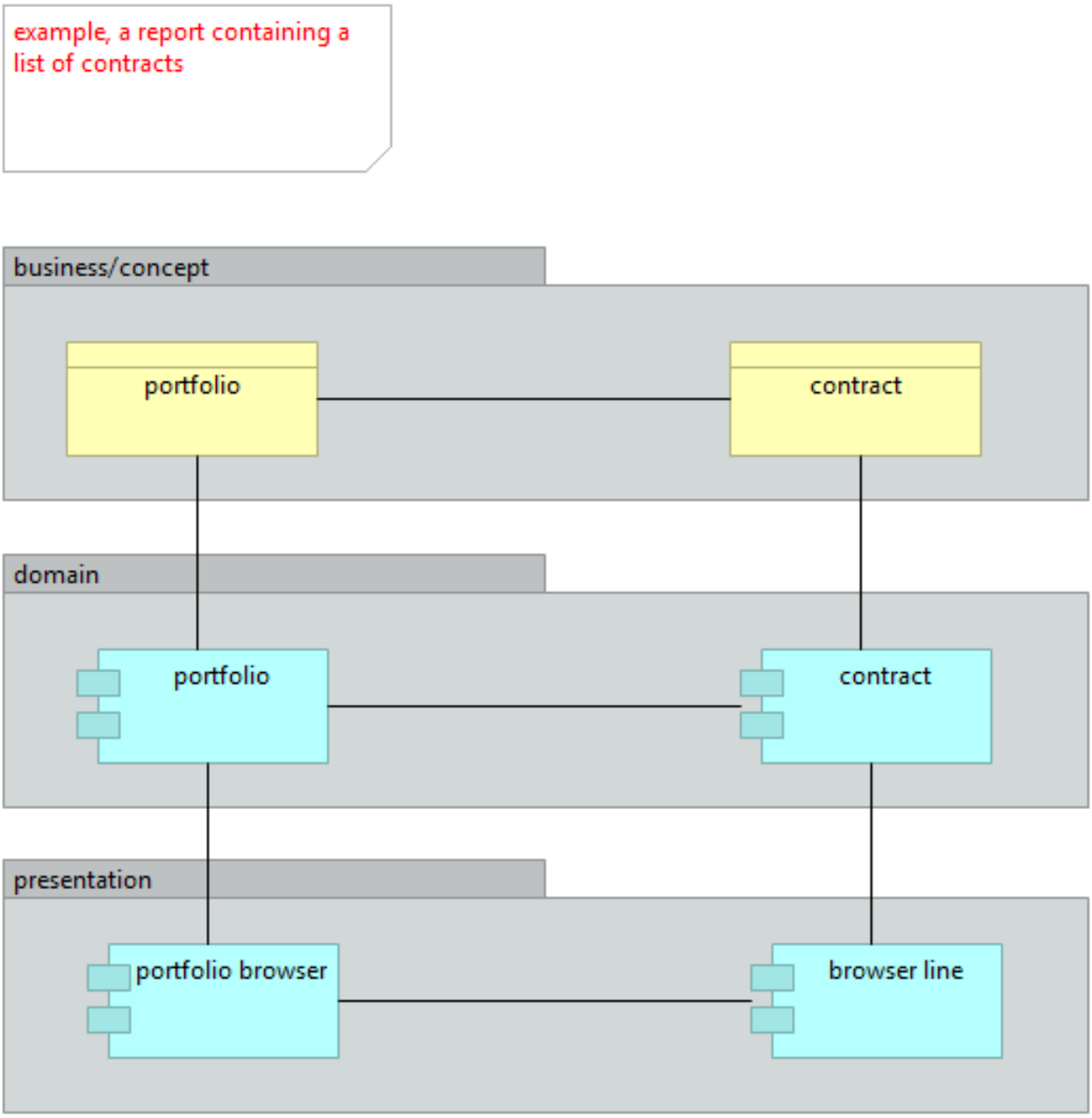
PRODUCT


- combination options can be seen as a composite of other options.
2 legs: 1 long 1 short, of wich 1 call, 1 put
2 legs: 1 call 1 put, both long OR both short
all legs calls OR all legs puts
product
combination product
risk reversal
strangle
straddle
spread
put spread
call spread
N|0 - 1 primary party
N|0 - 1 counterparty
party
strike prices of legs are the same
2 legs, both puts with same expiry
2 legs, both calls with same expiry

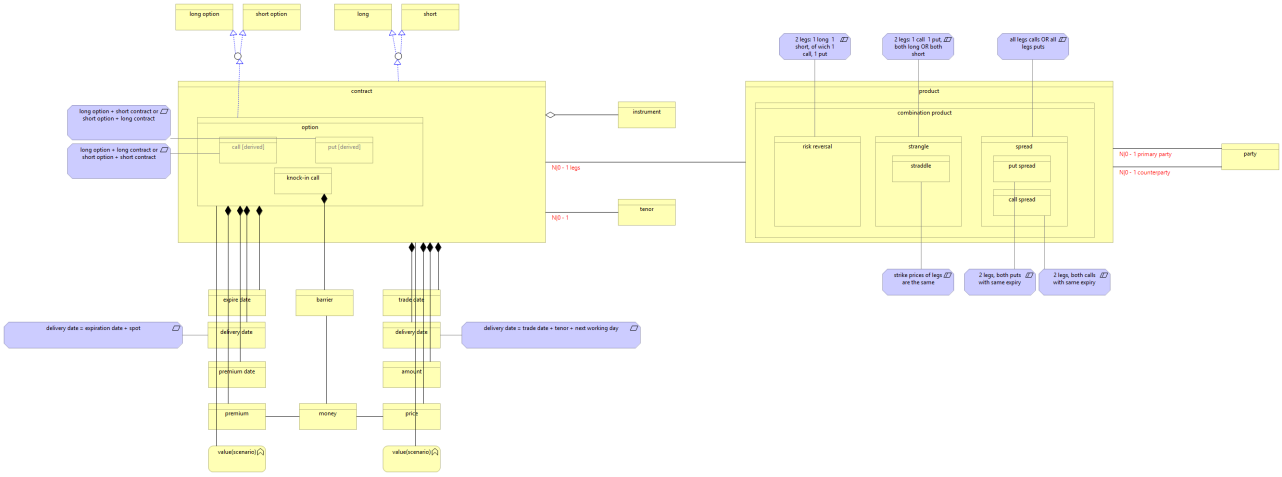
SUBTYPE STATE MACHINES



PARALLEL APPLICATION AND DOMAIN HIERARCHIES



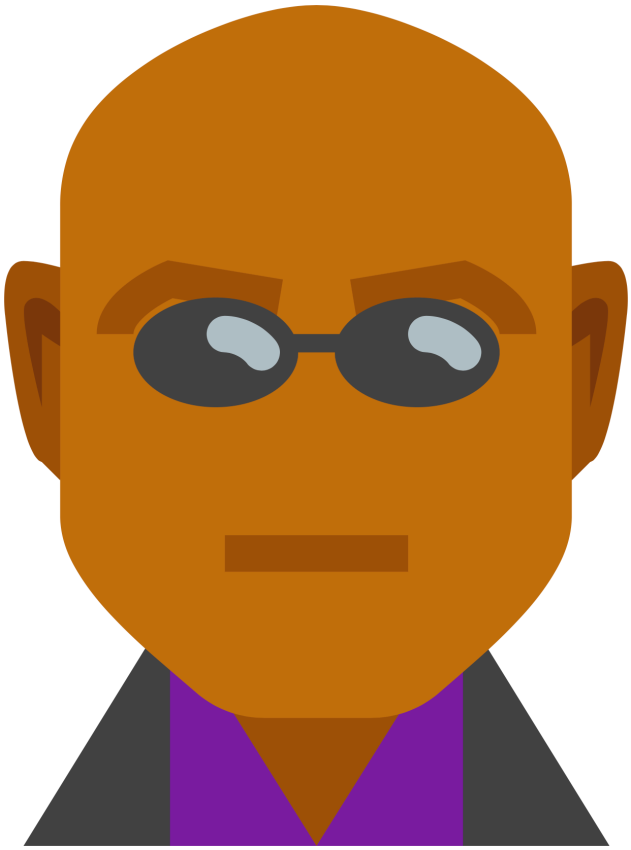
DERIVATIVE CONTRACTS



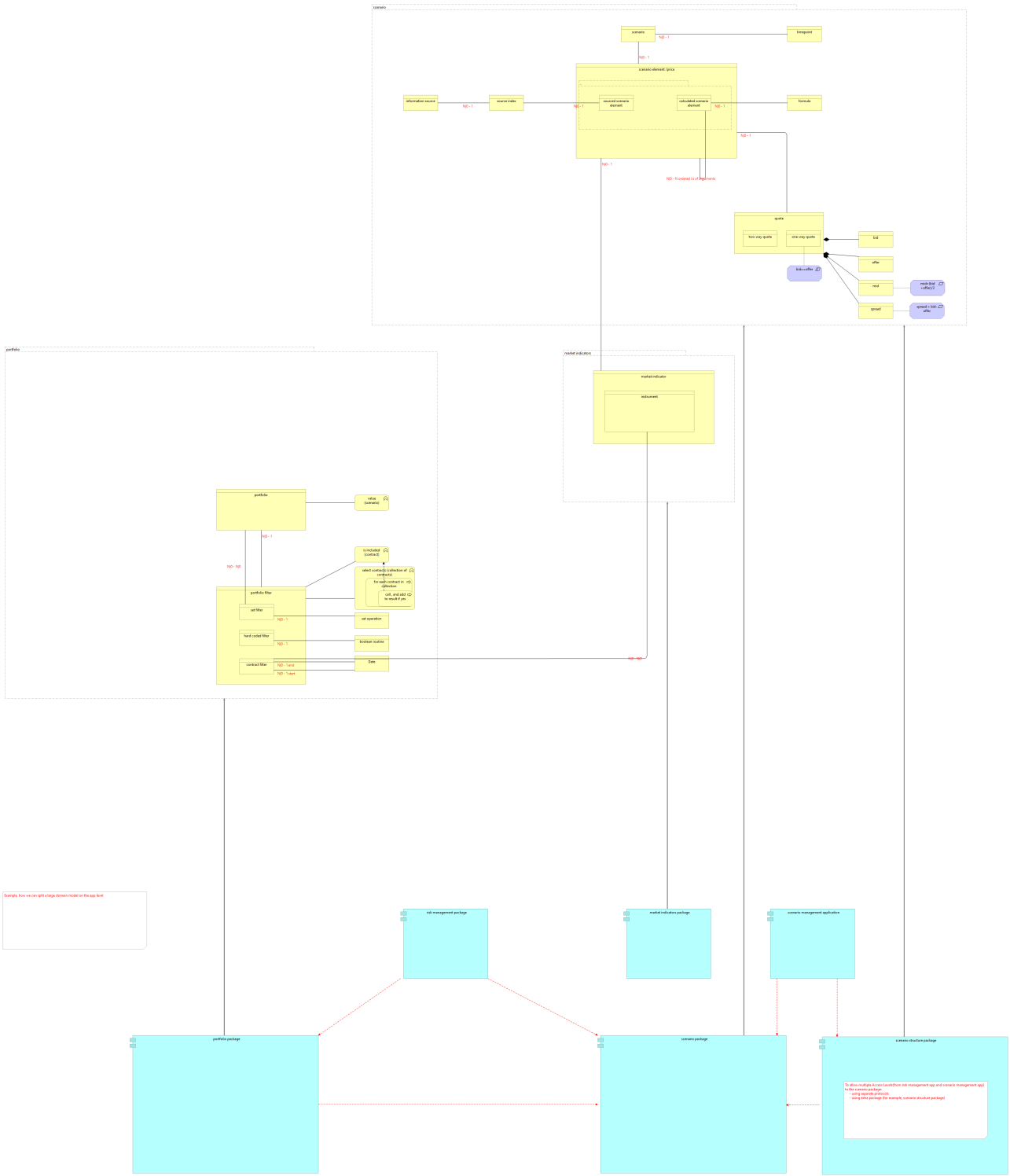
TRADING PACKAGES

ANALYSIS PATTERNS

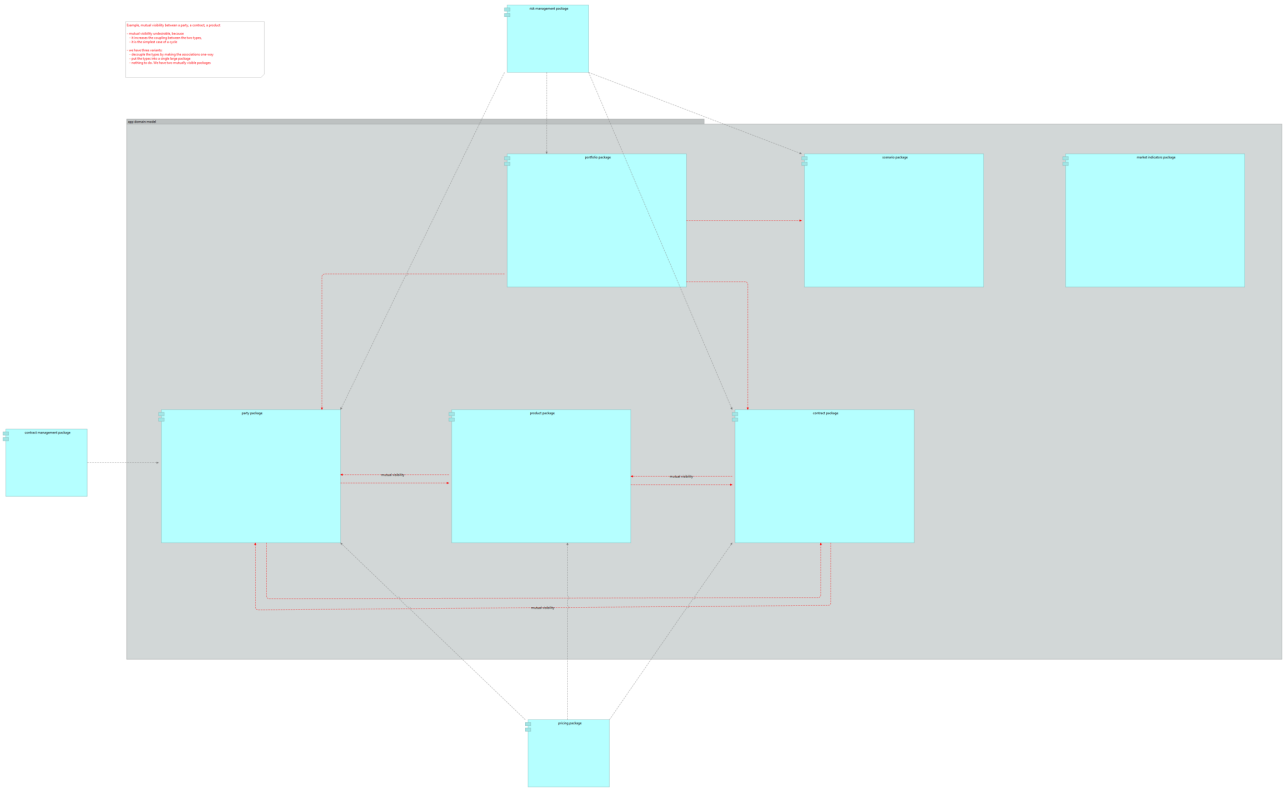
Martin Fowler



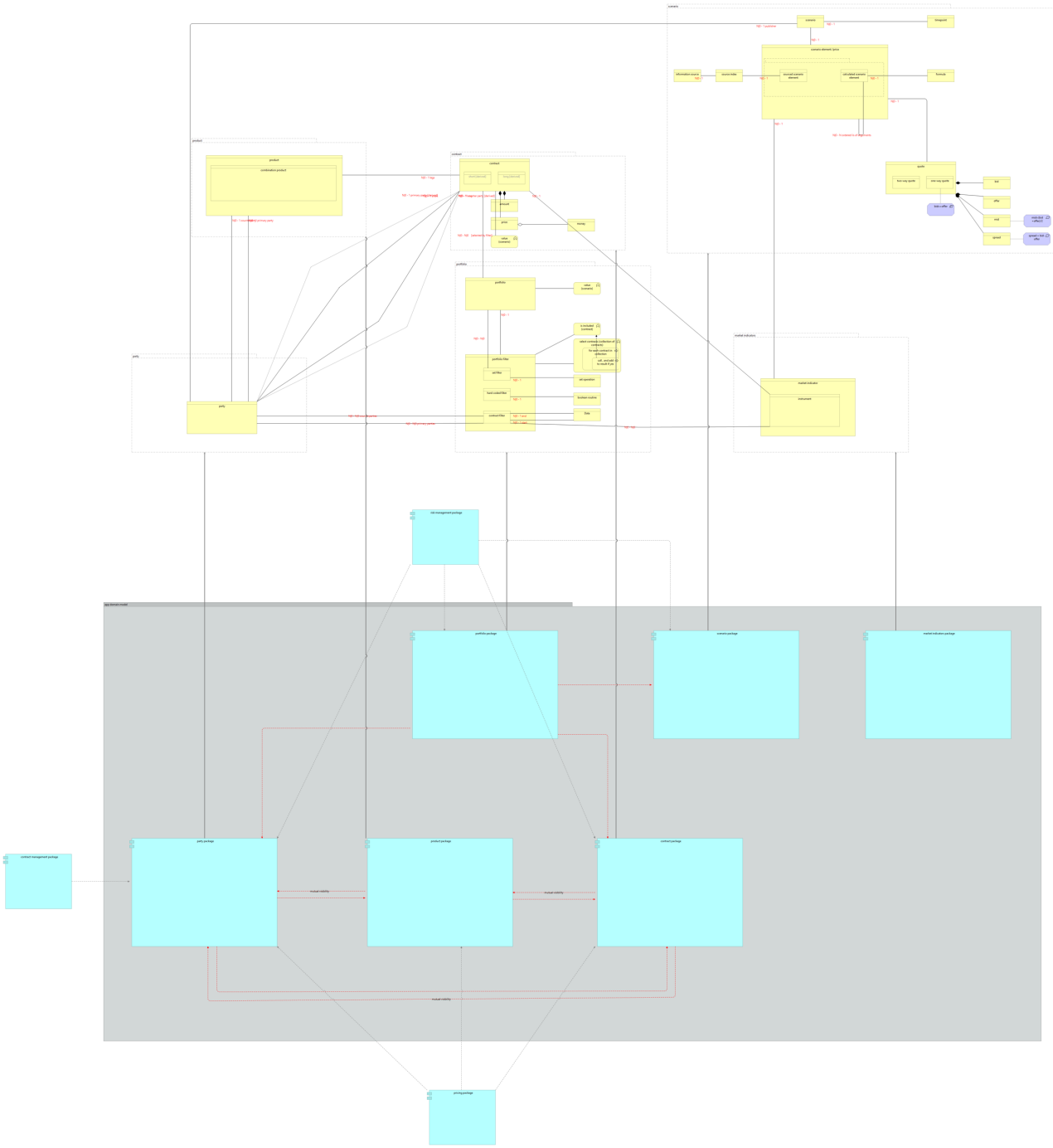
MULTIPLE ACCESS LEVELS TO A PACKAGE



MUTUAL VISIBILITY



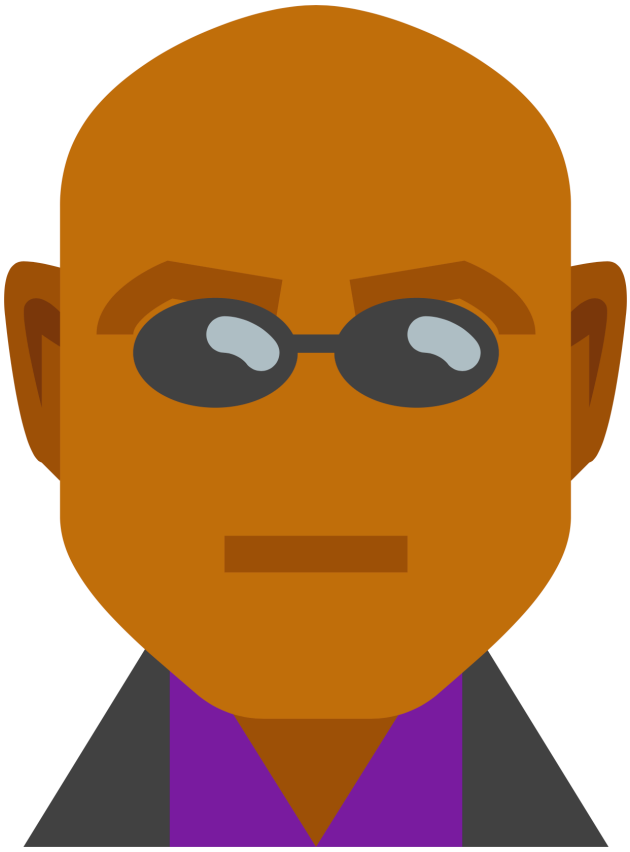
TRADING PACKAGES



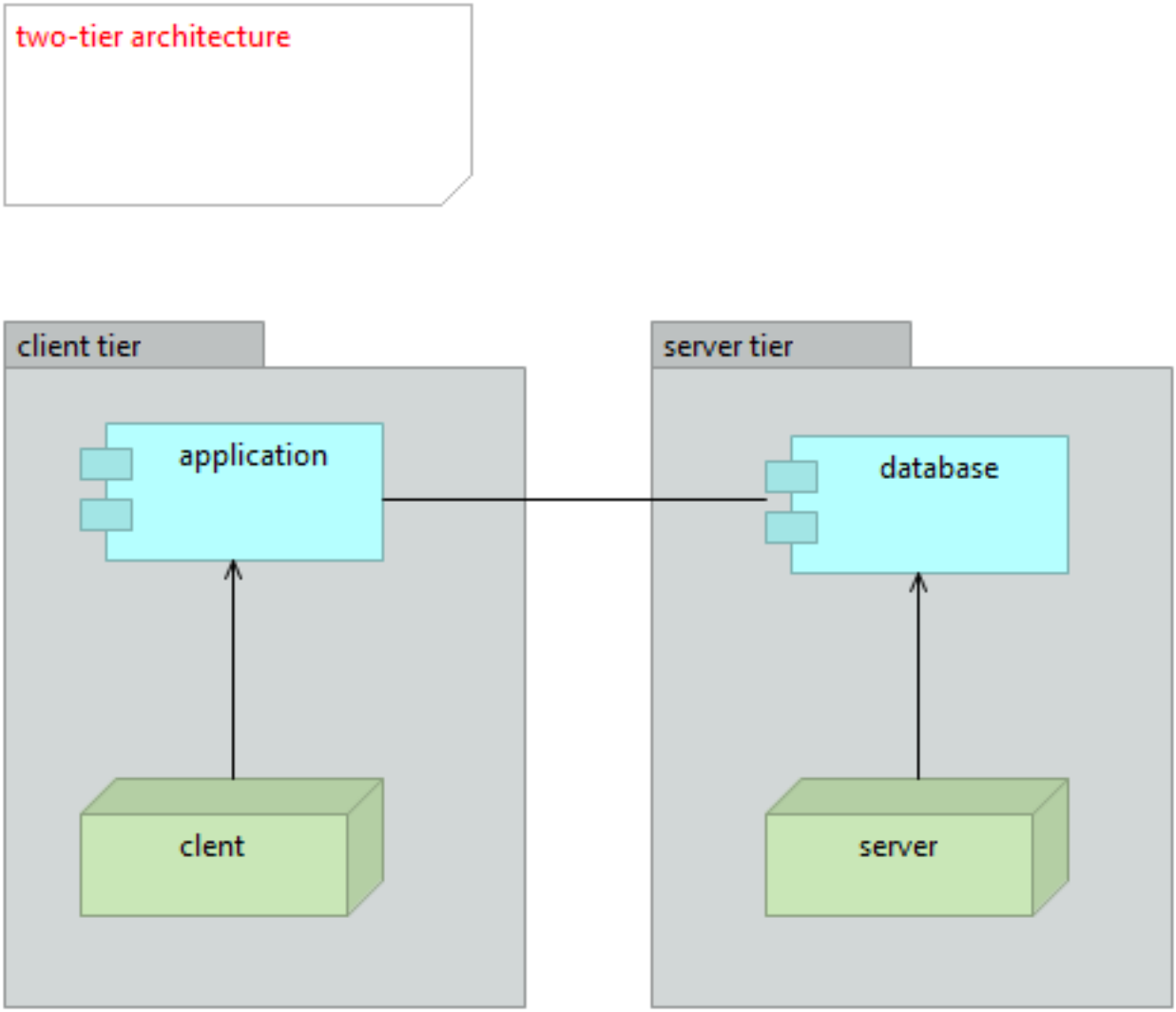
LAYERED ARCHITECTURE

ANALYSIS PATTERNS

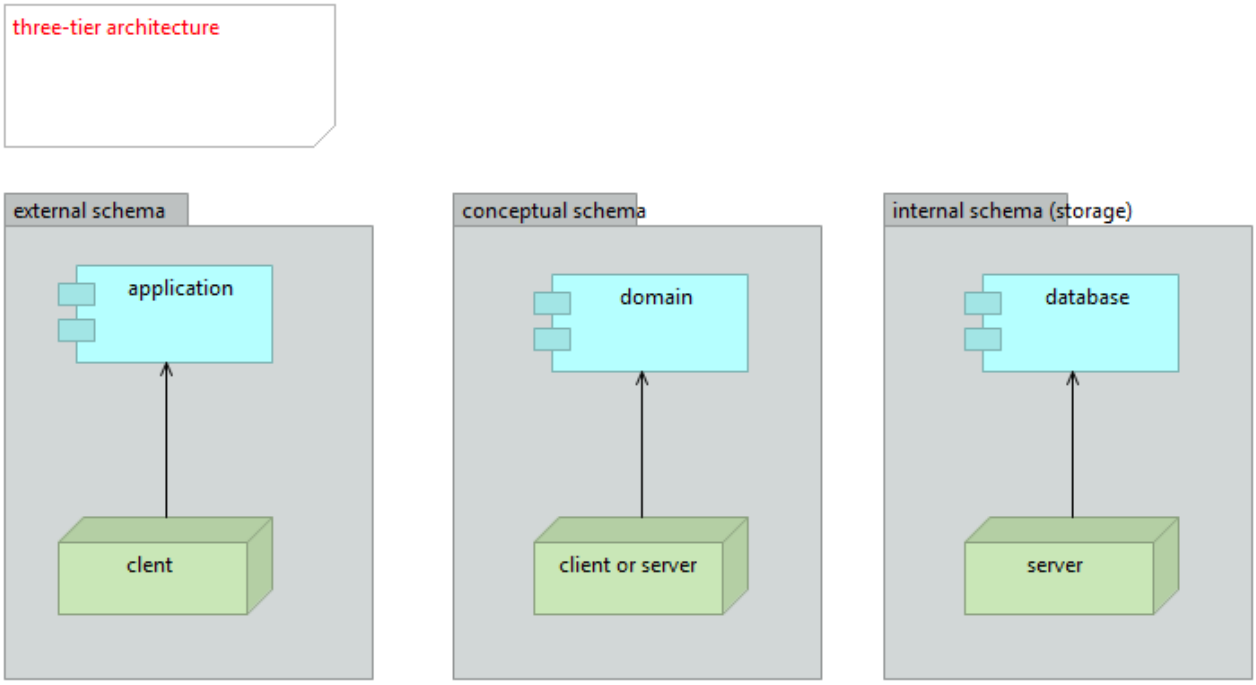
Martin Fowler



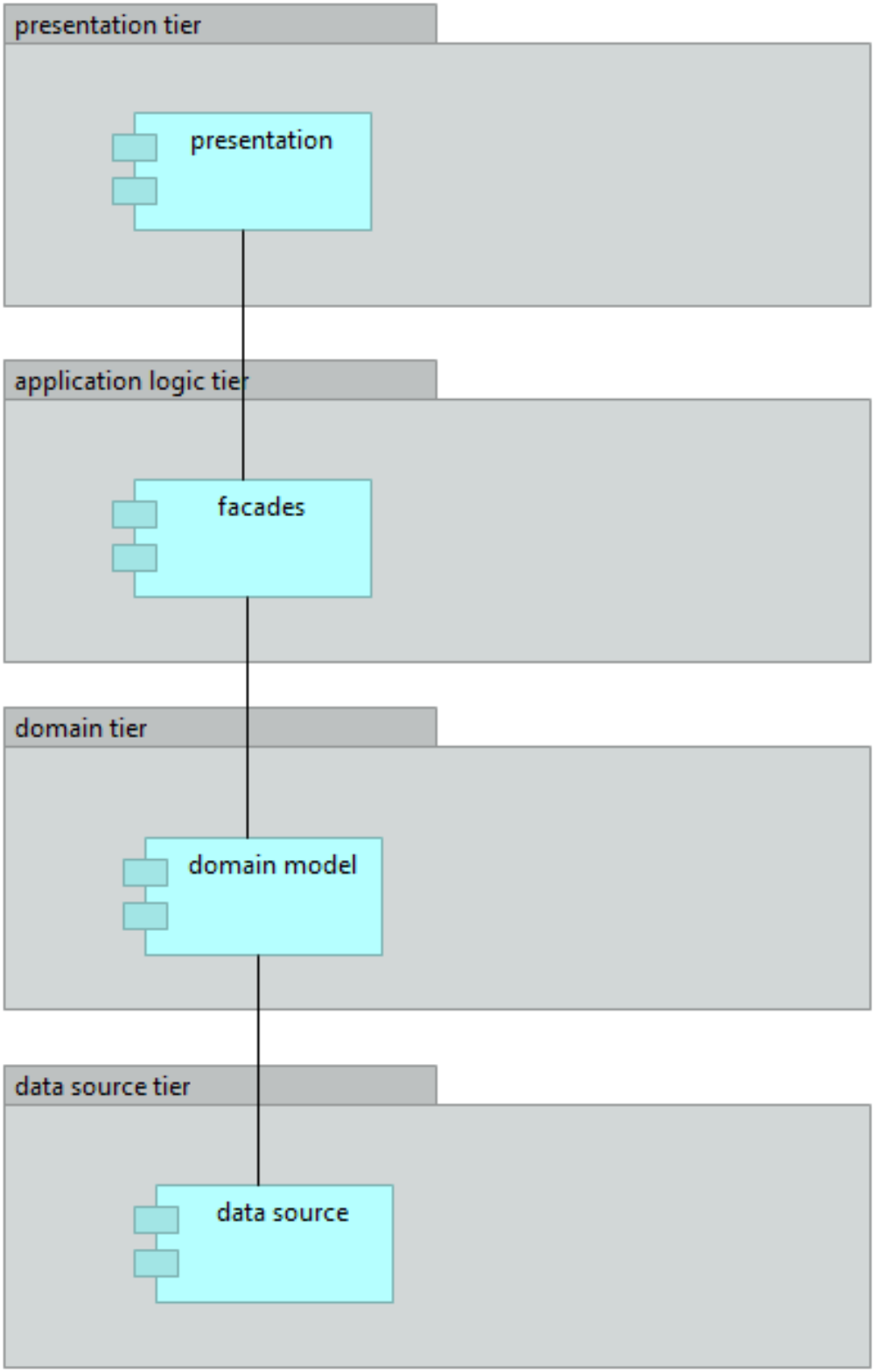
TWO-TIER ARCHITECTURE



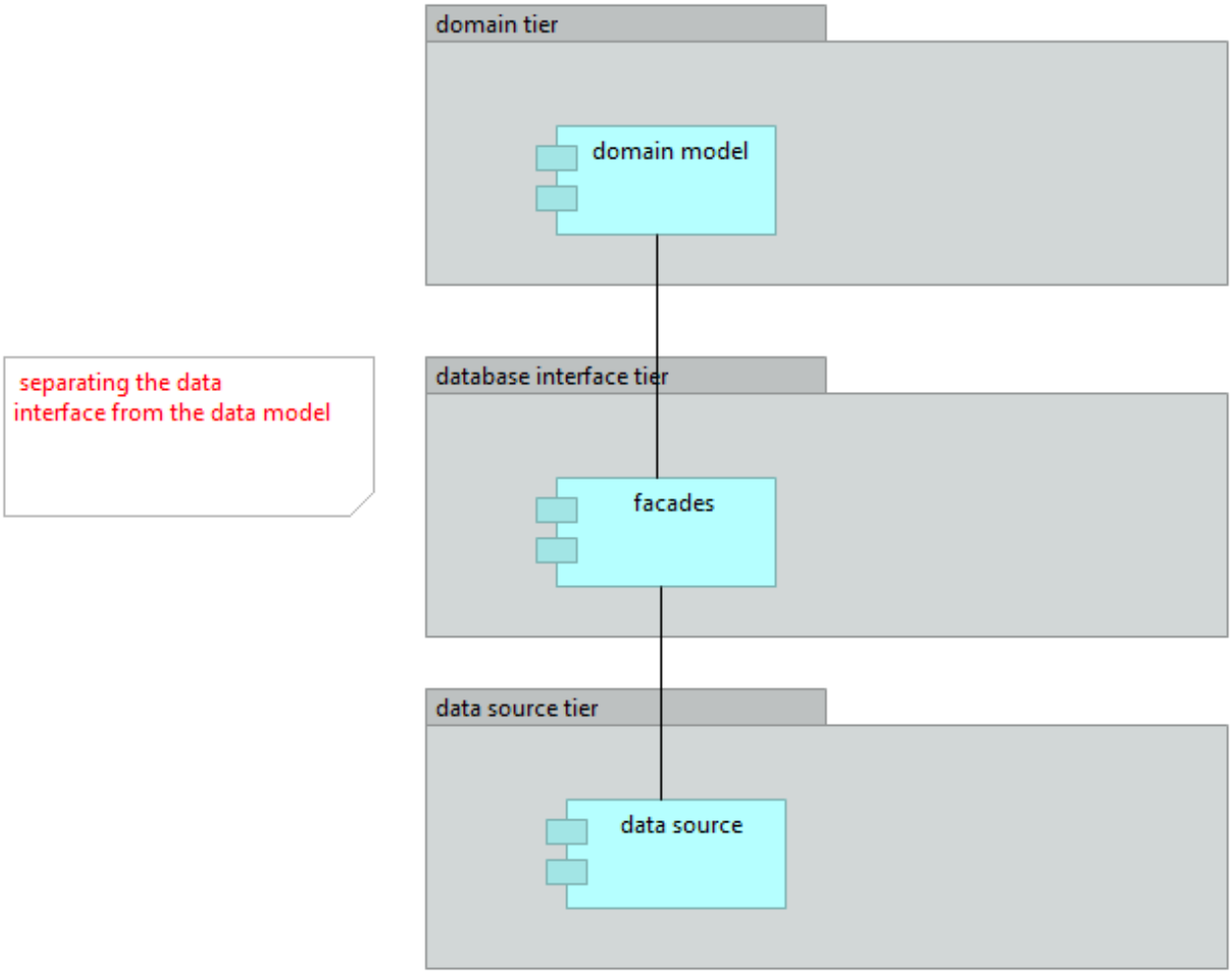
THREE-TIER ARCHITECTURE



PRESENTATION AND APPLICATION LOGIC



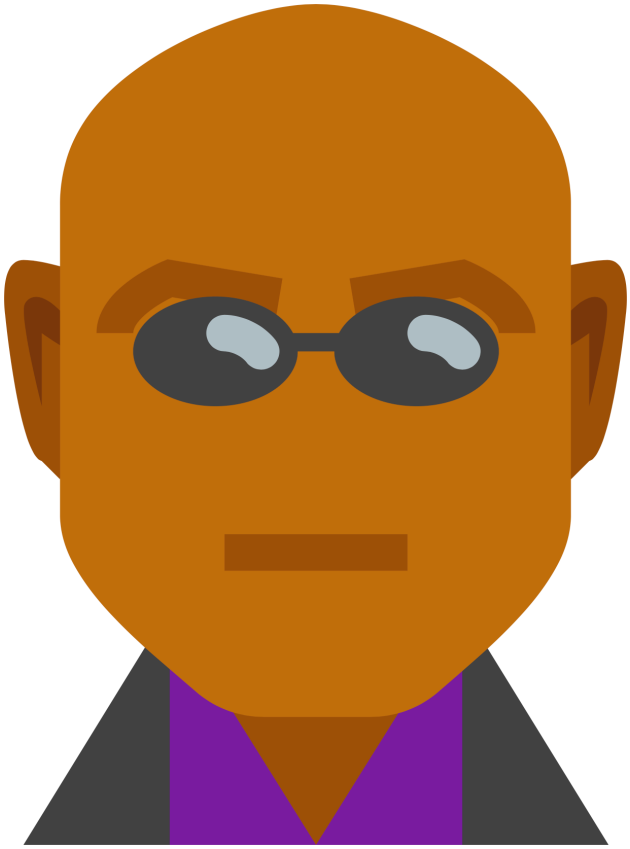
DATABASE INTERACTION



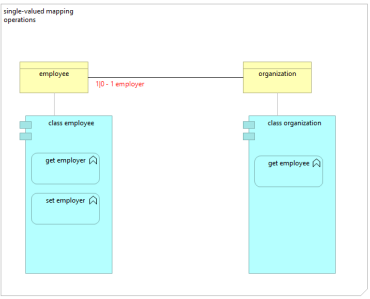
TYPE MODEL DESIGN

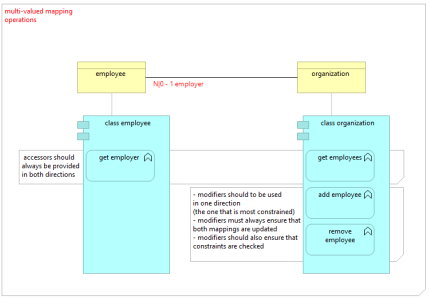
ANALYSIS PATTERNS

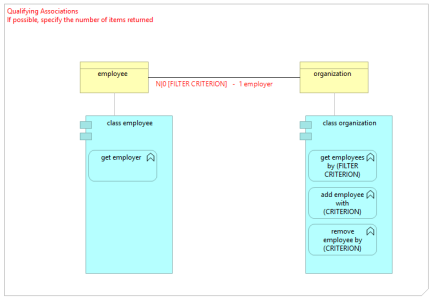
Martin Fowler

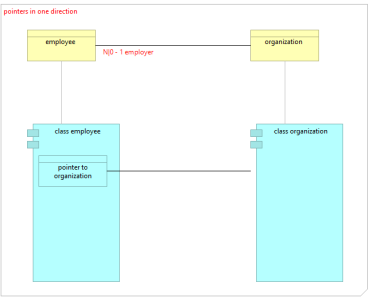


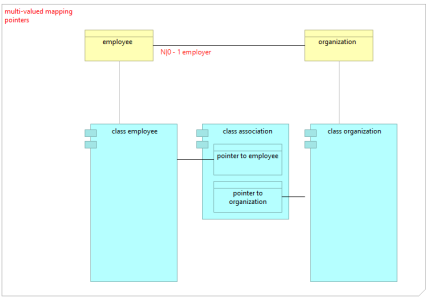
IMPLEMENTING ASSOCIATIONS

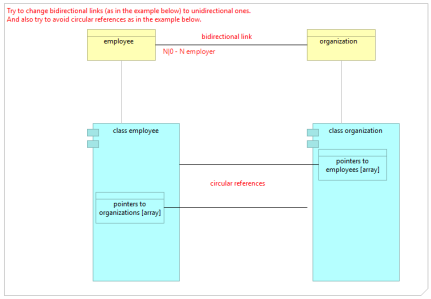




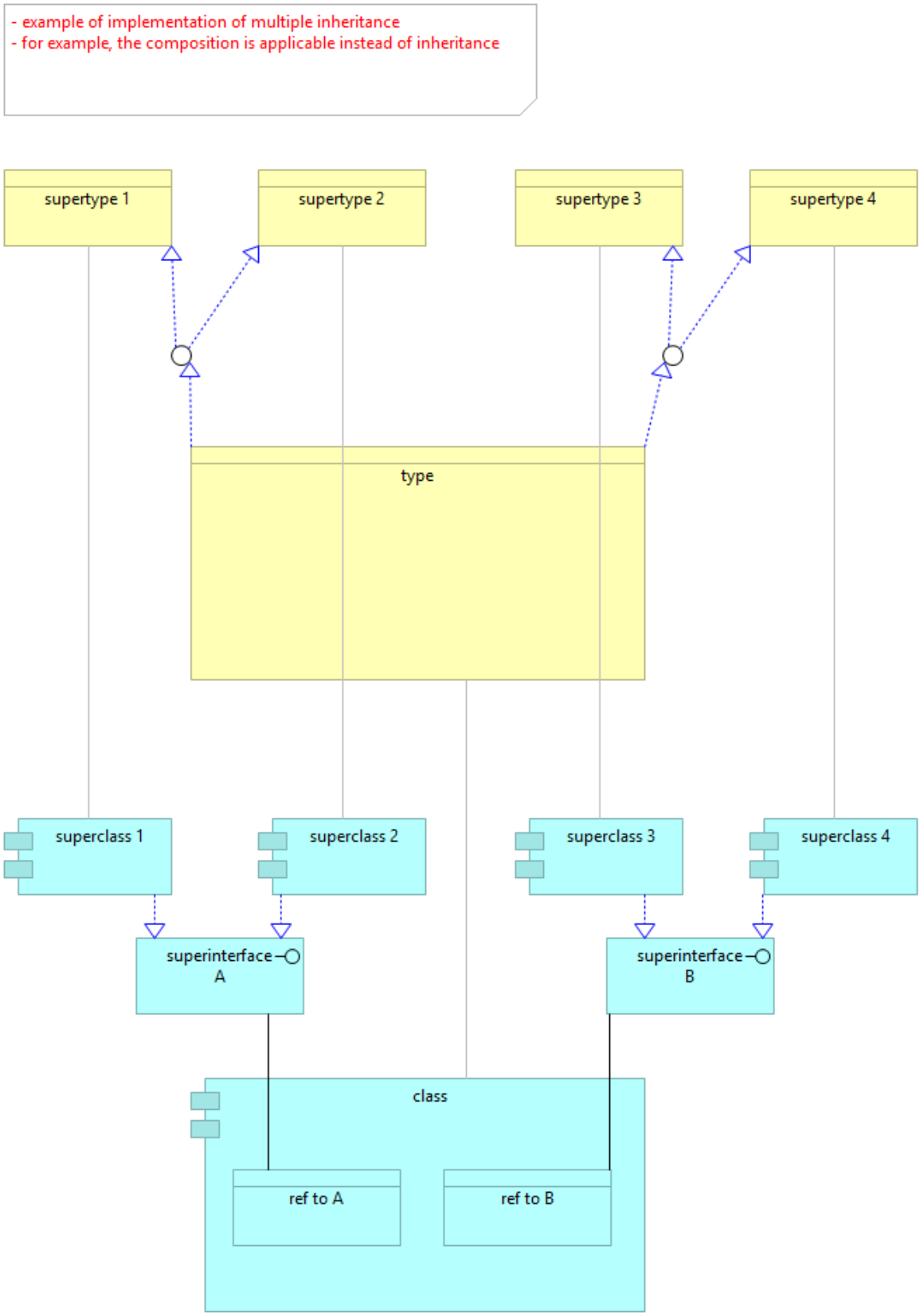




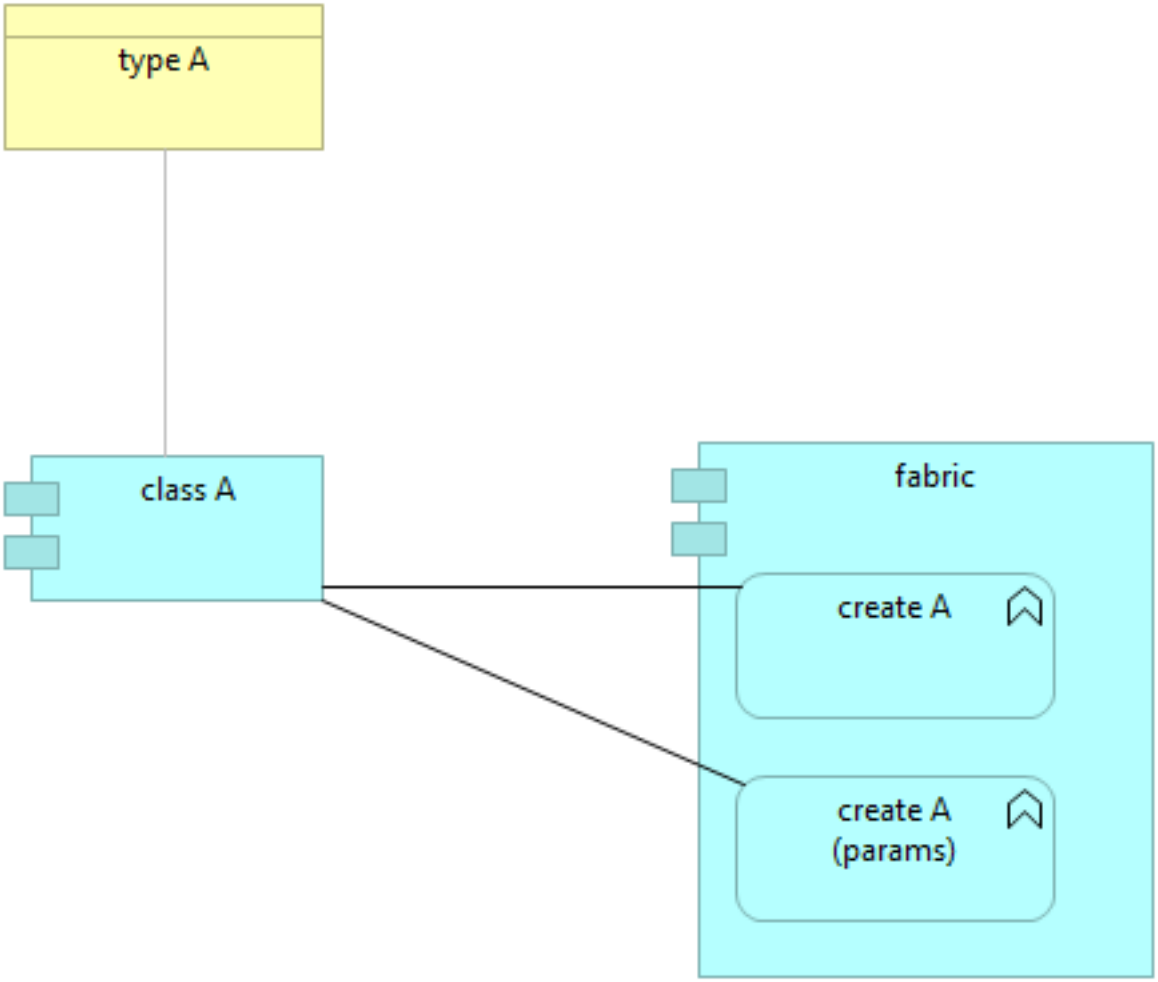




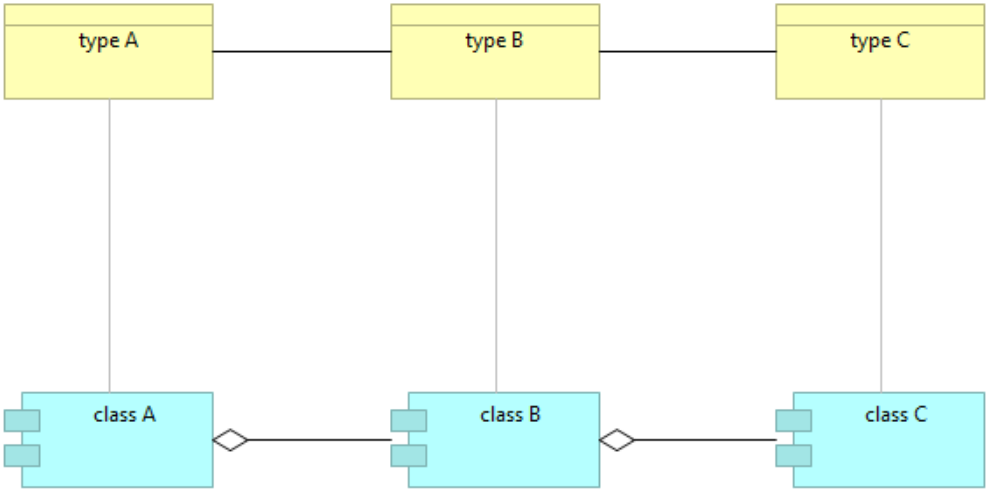
IMPLEMENTING GENERALIZATION

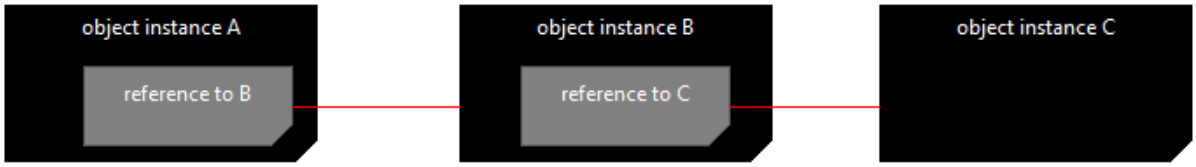


OBJECT CREATION

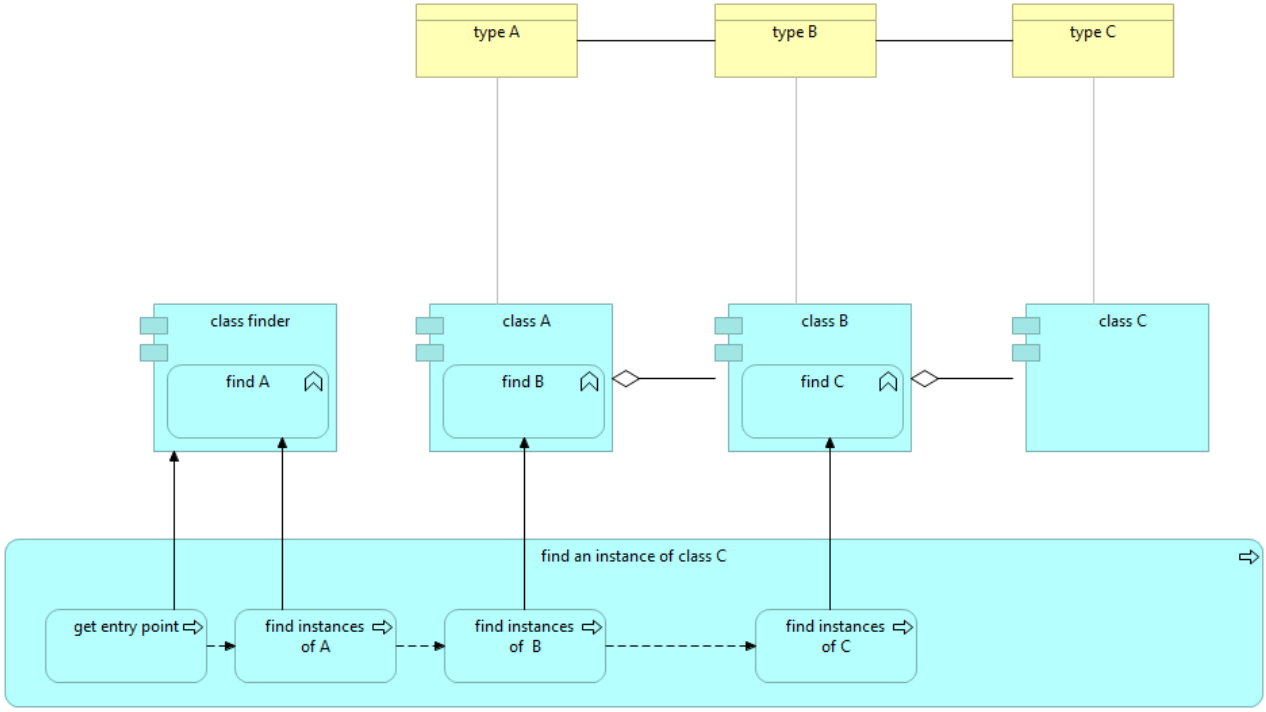


OBJECT DESTRUCTION

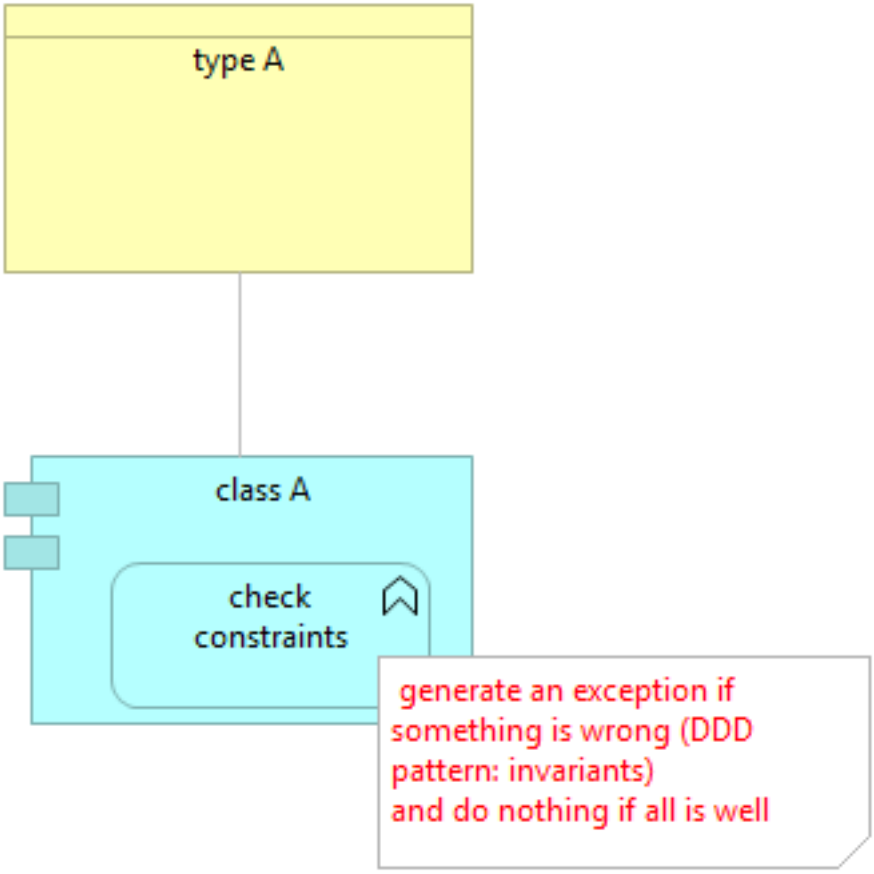




ENTRY POINT.


type A
type B
type C
class finder
find A
class A
find B
class B
find C
class C
find an instance of class C
get entry point
find instances of A
find instances of B
find instances of C

IMPLEMENTING CONSTRAINTS

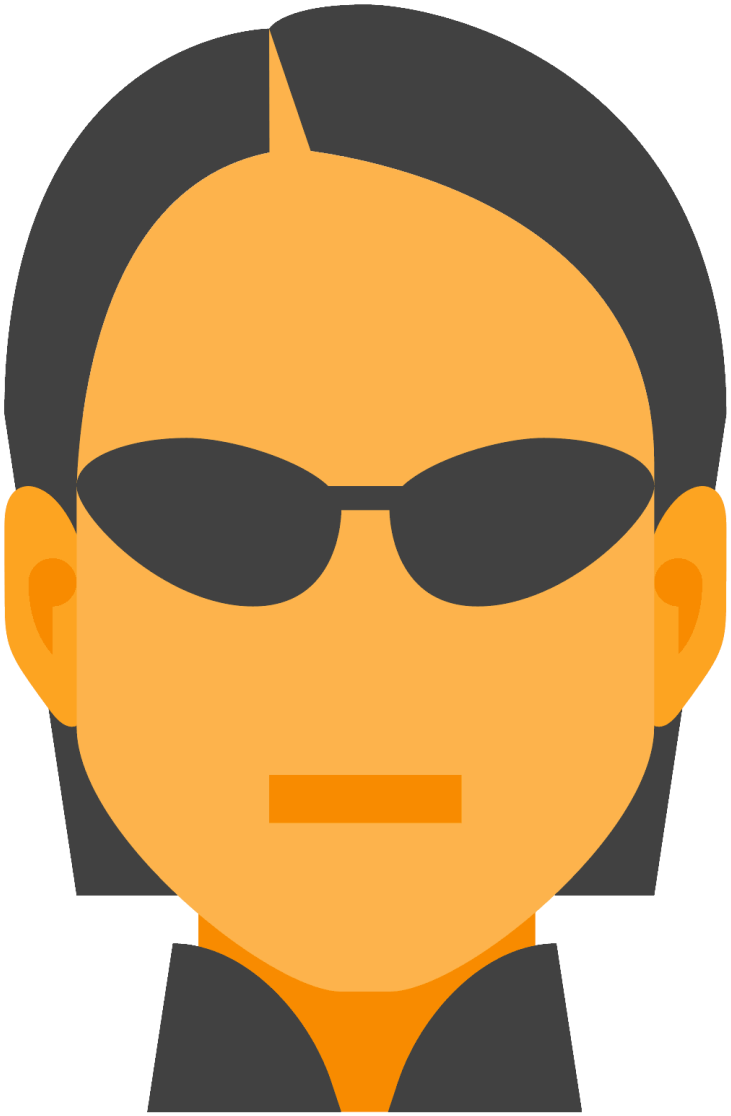


04

Domain Driven Design

Tackling Complexity in the Heart of Software.

ERIC EVANS

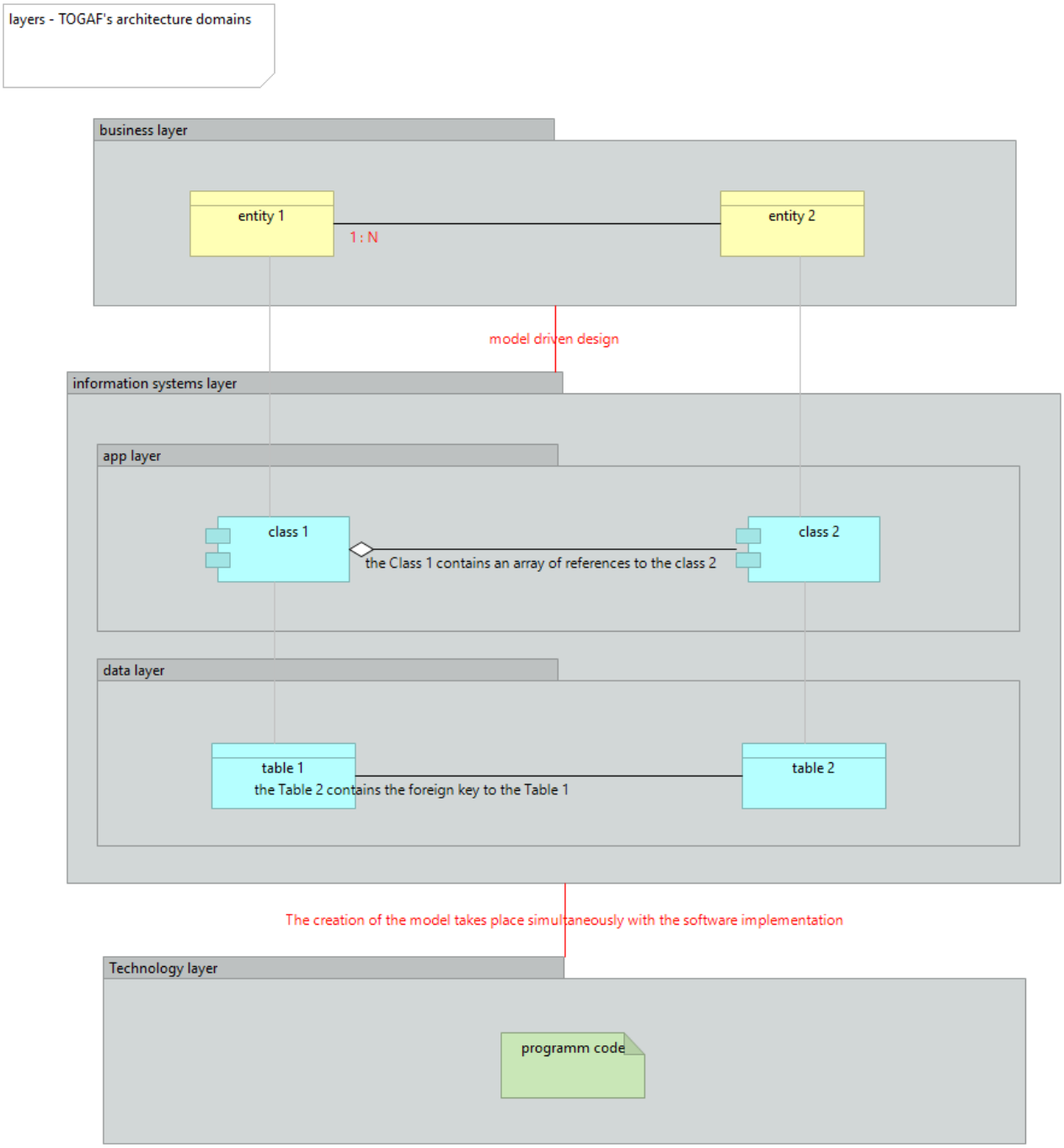


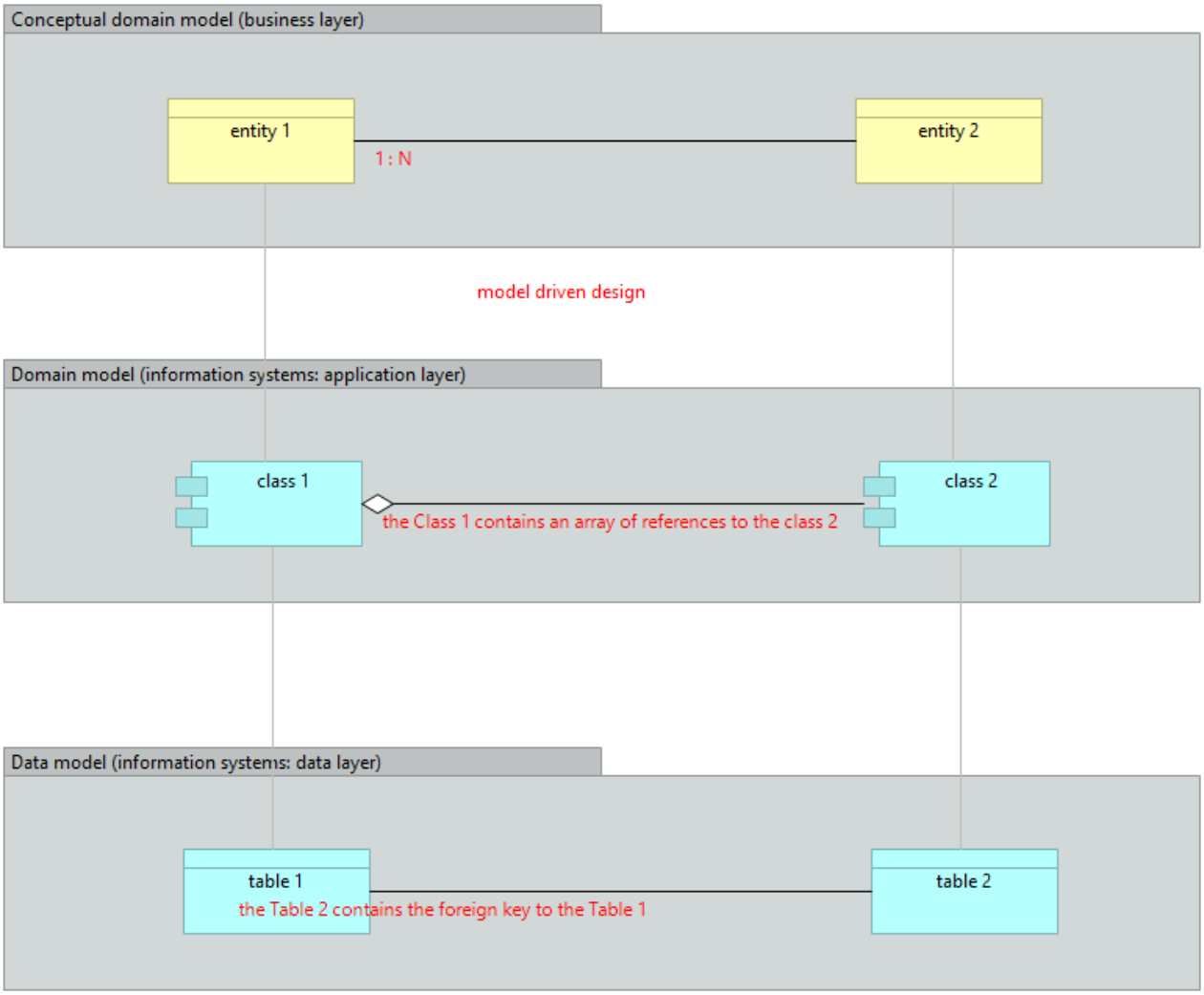
MODEL AND STRUCTURAL ELEMENTS

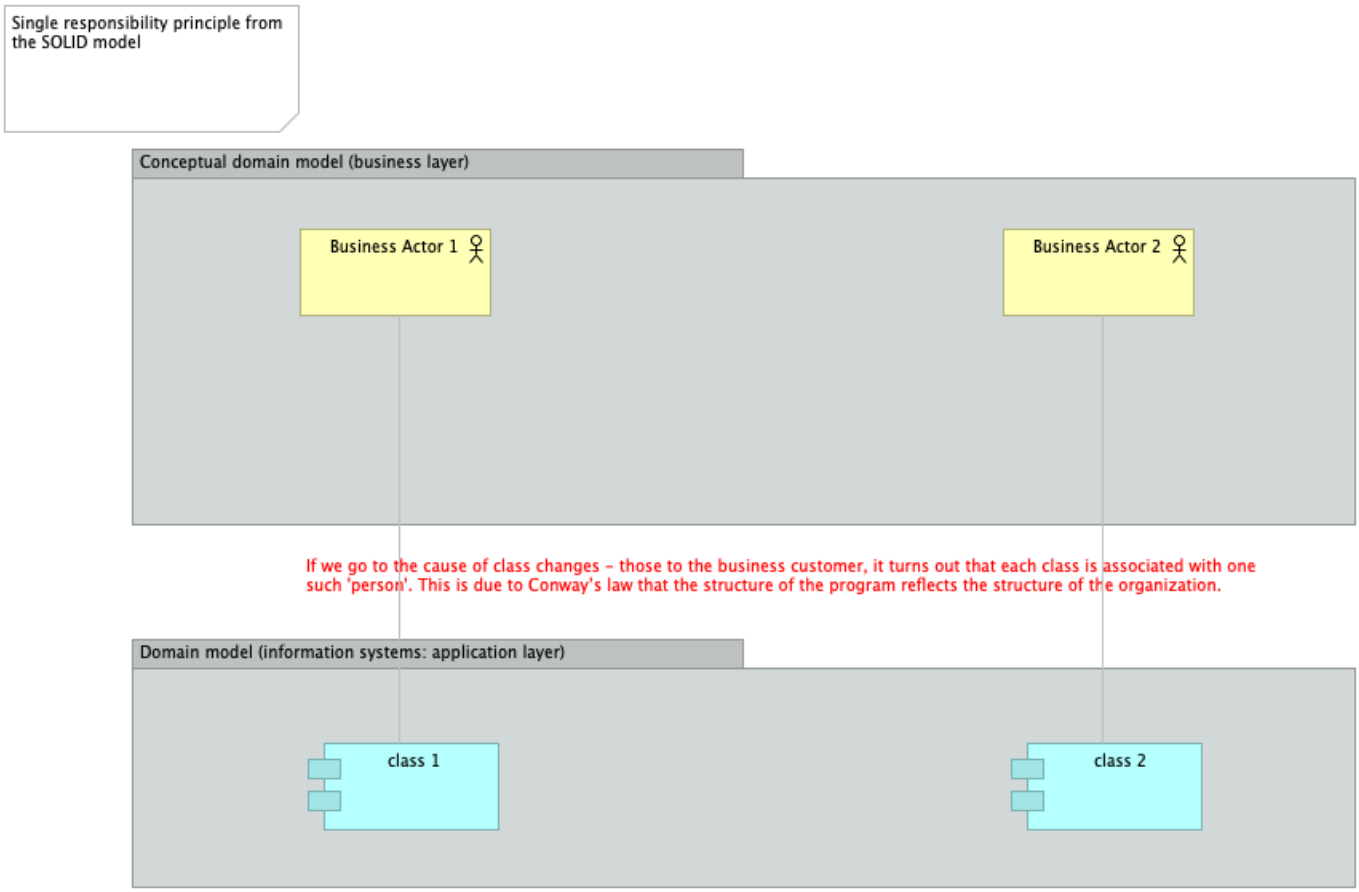
DOMAIN DRIVEN DESIGN

Eric Evans

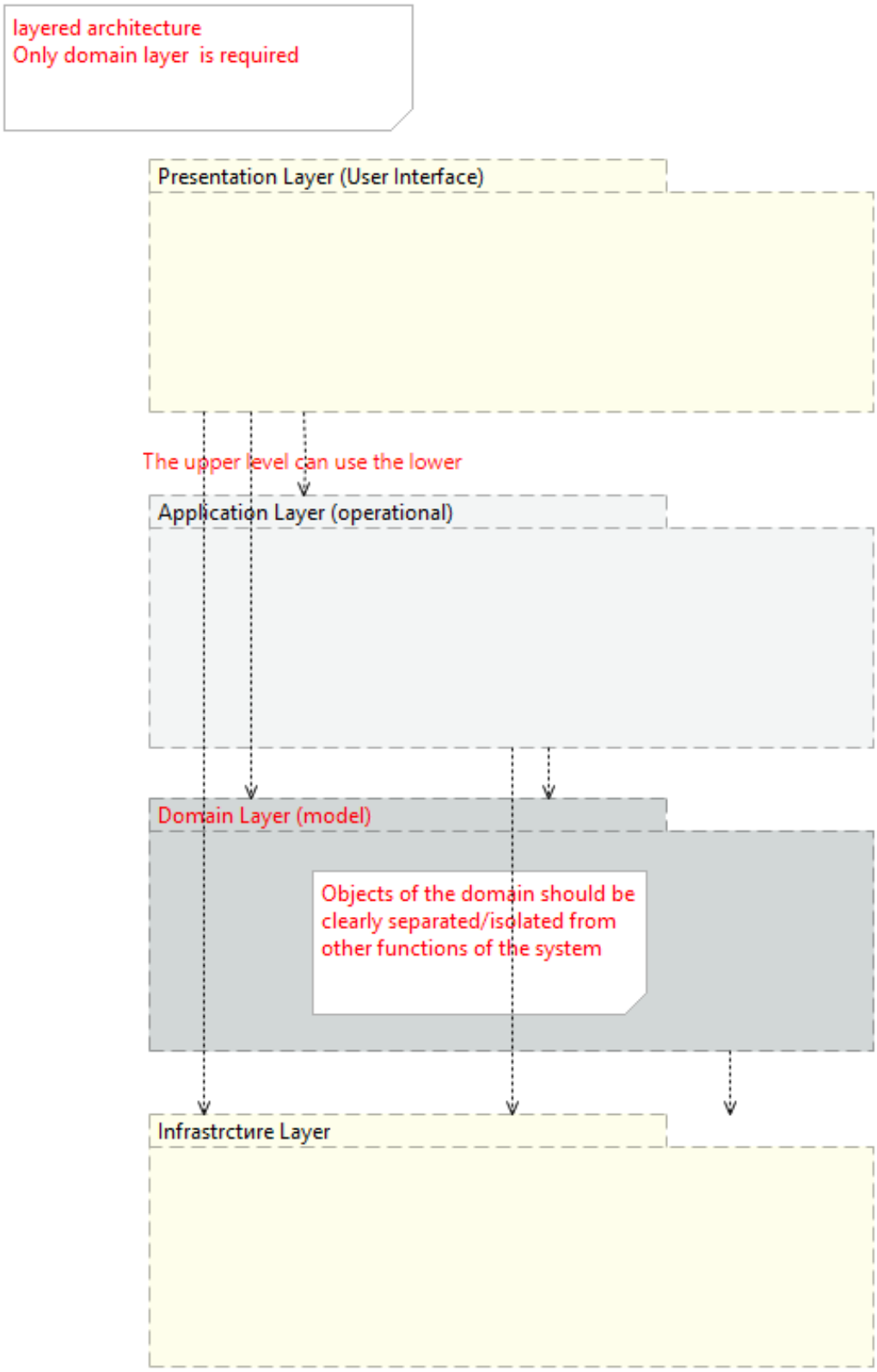
MODEL-DRIVEN DESIGN



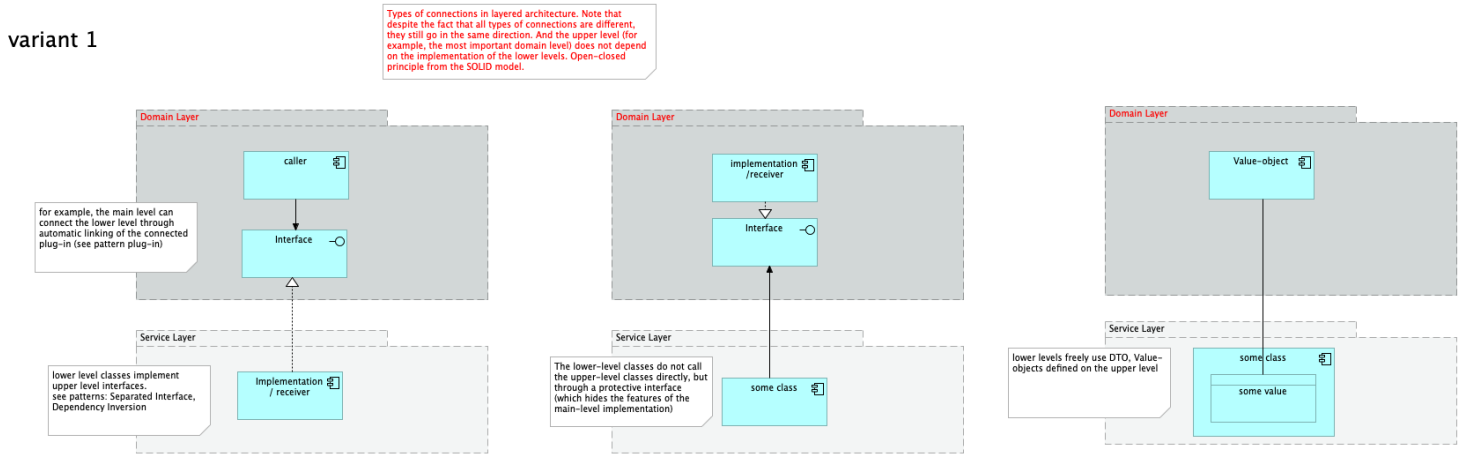

Conceptual domain model (business layer)
entity 1
1 : N
entity 2
model driven design
Domain model (information systems: application layer)
class 1
the Class 1 contains an array of references to the class 2
class 2
Data model (information systems: data layer)
table 1
the Table 2 contains the foreign key to the Table 1
table 2


Single responsibility principle from the SOLID model
Conceptual domain model (business layer)
Business Actor 1
Business Actor 2
If we go to the cause of class changes – those to the business customer, it turns out that each class is associated with one such 'person'. This is due to Conway's law that the structure of the program reflects the structure of the organization.
Domain model (information systems: application layer)
class 1
class 2

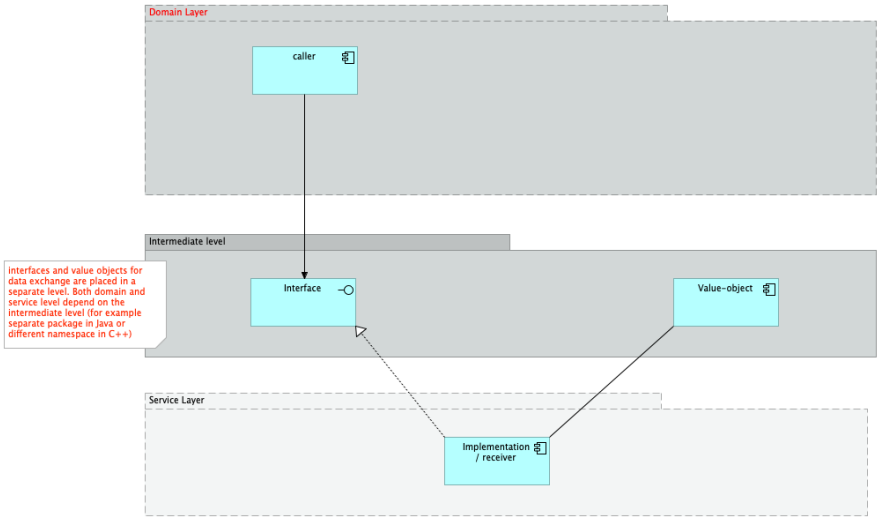
LAYERED ARCHITECTURE (ASYMMETRIC)



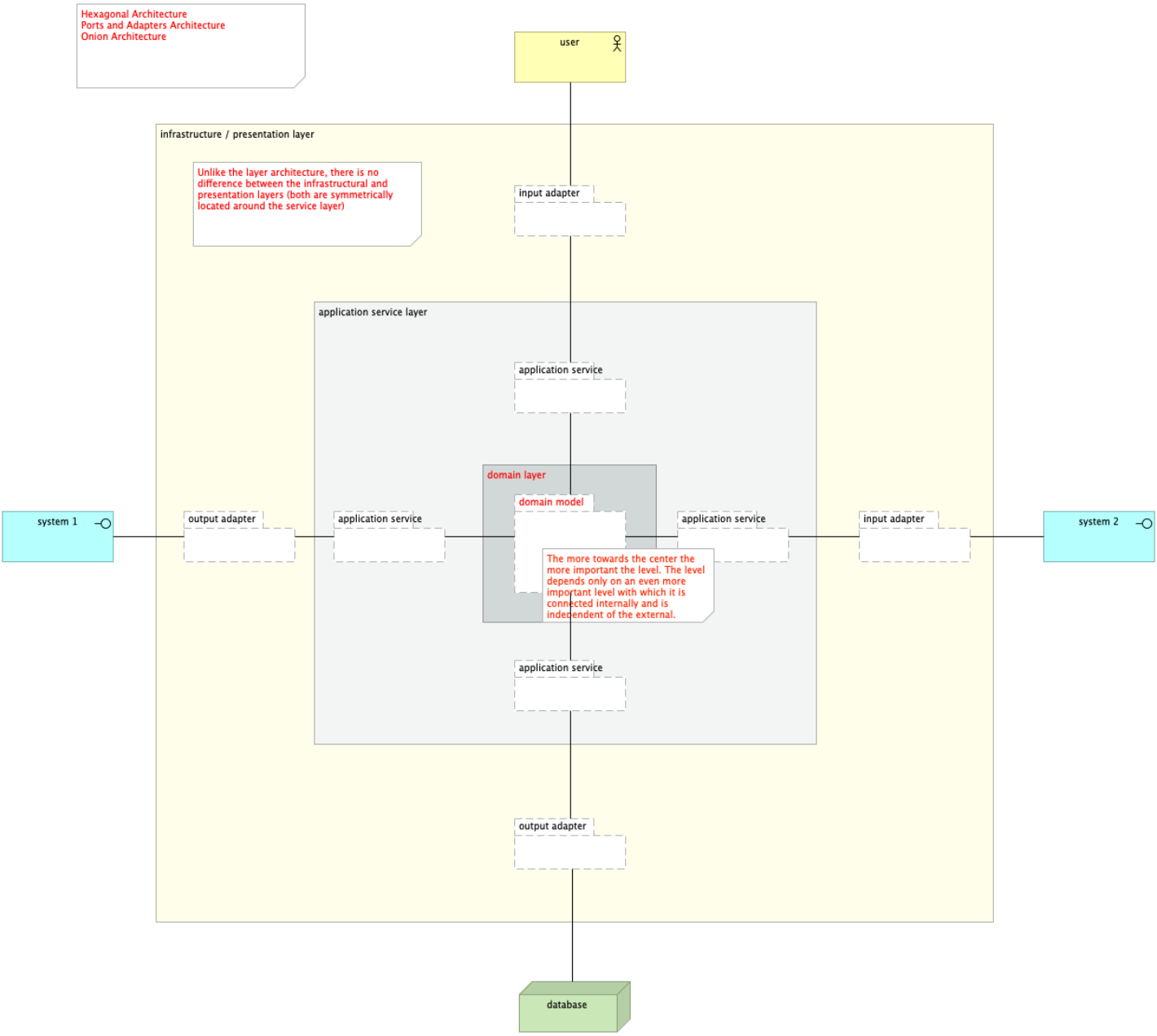
variant 1

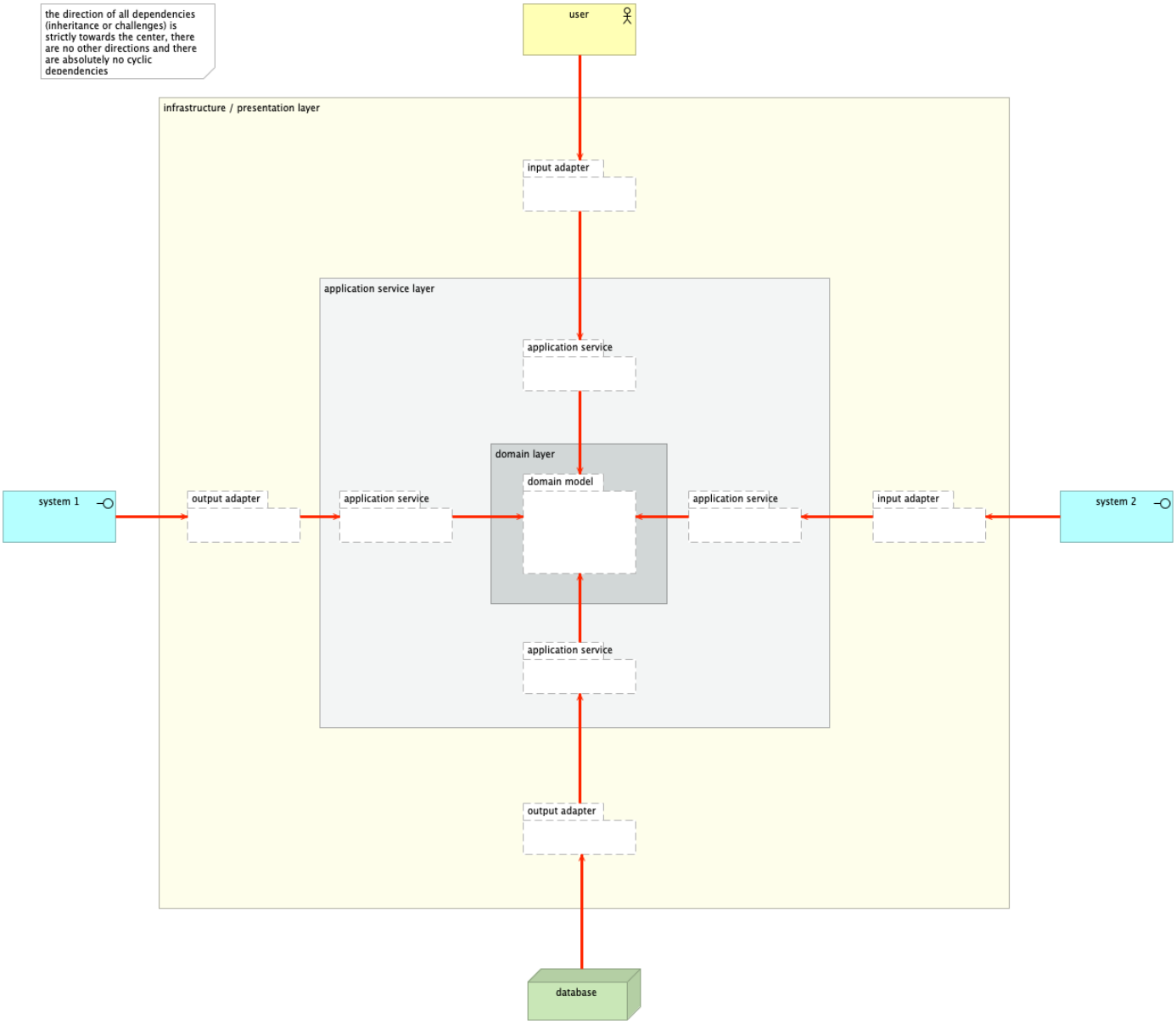


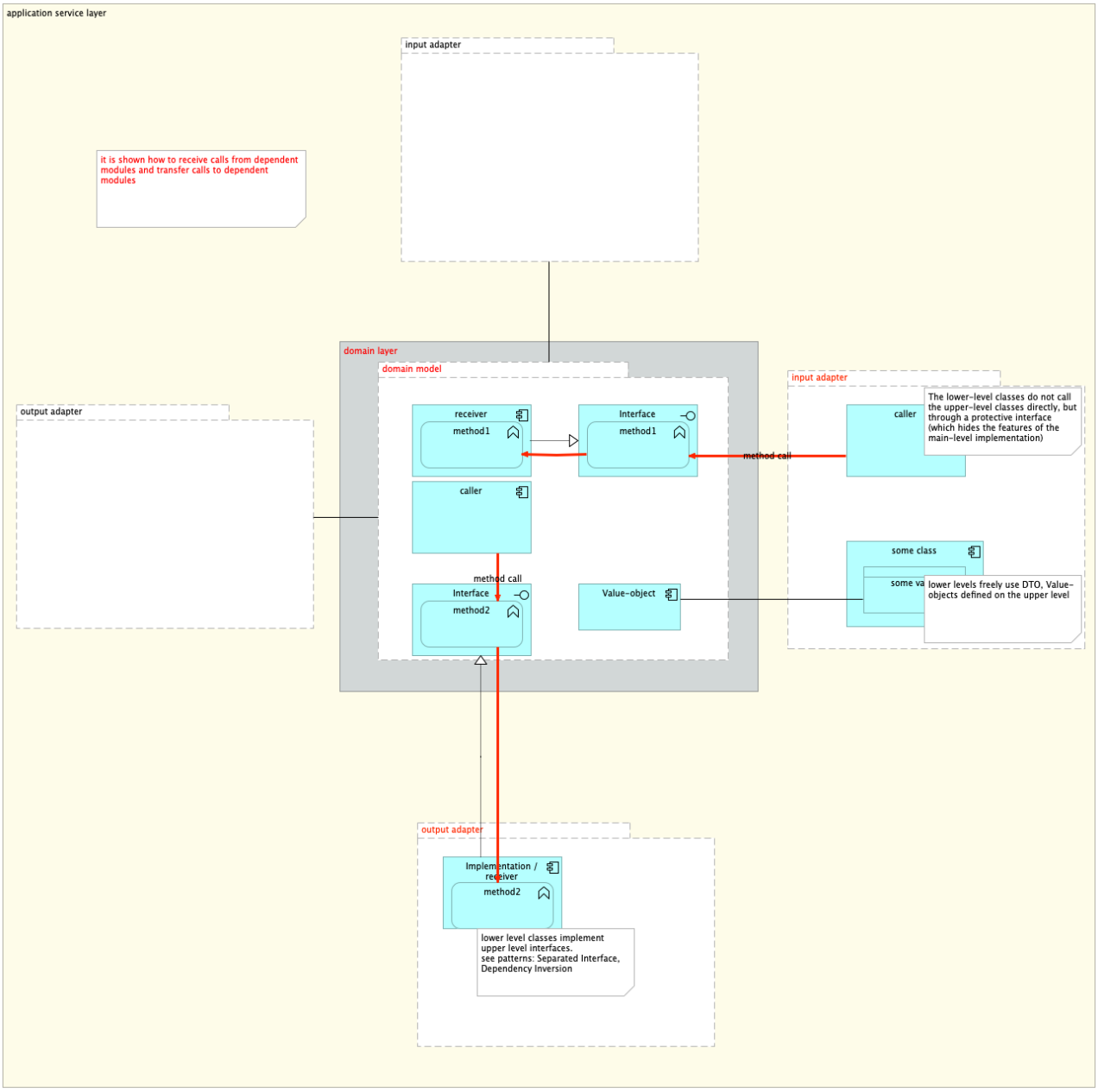
variant 2

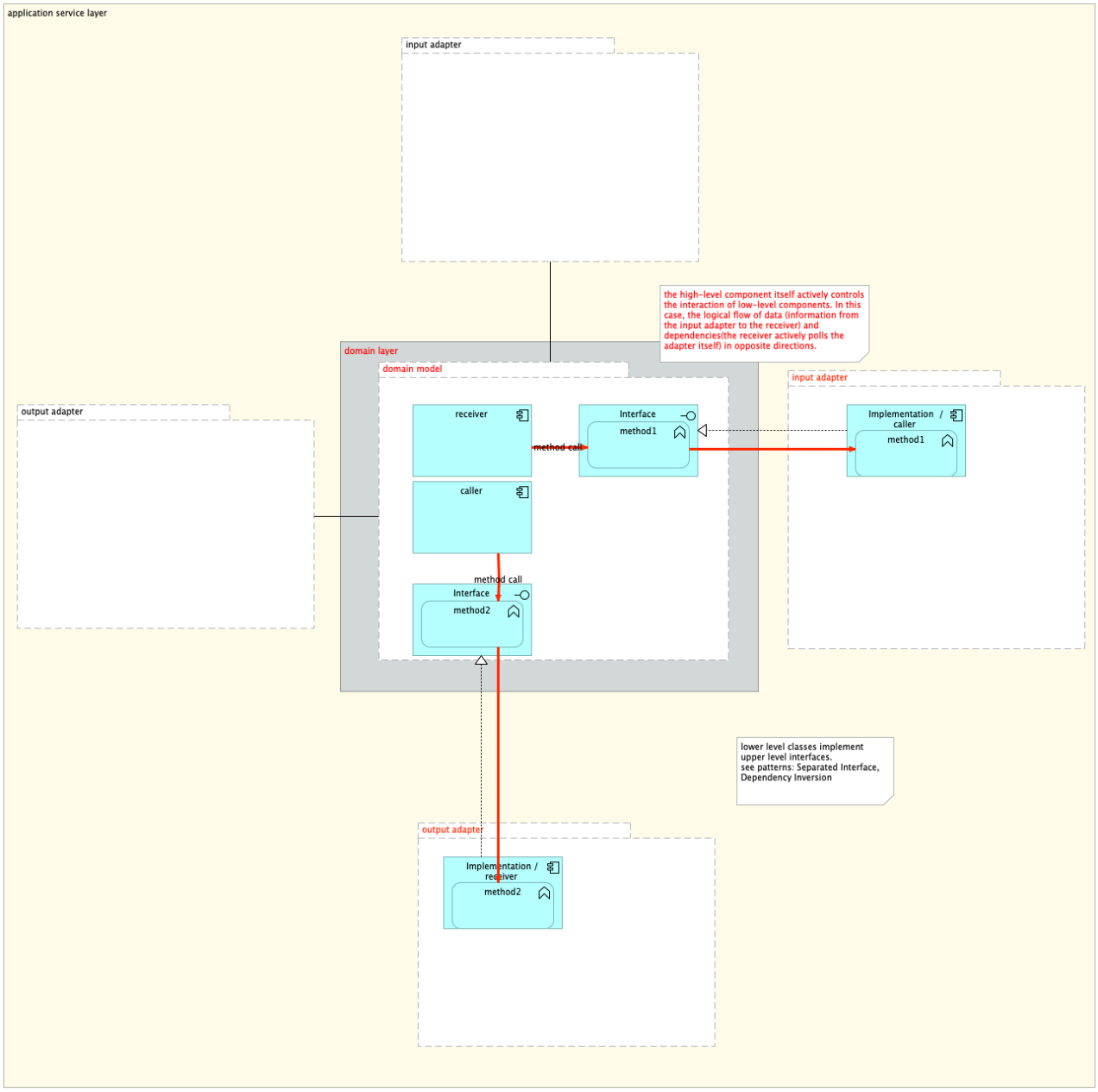


HEXAGONAL ARCHITECTURE (SYMMETRIC)

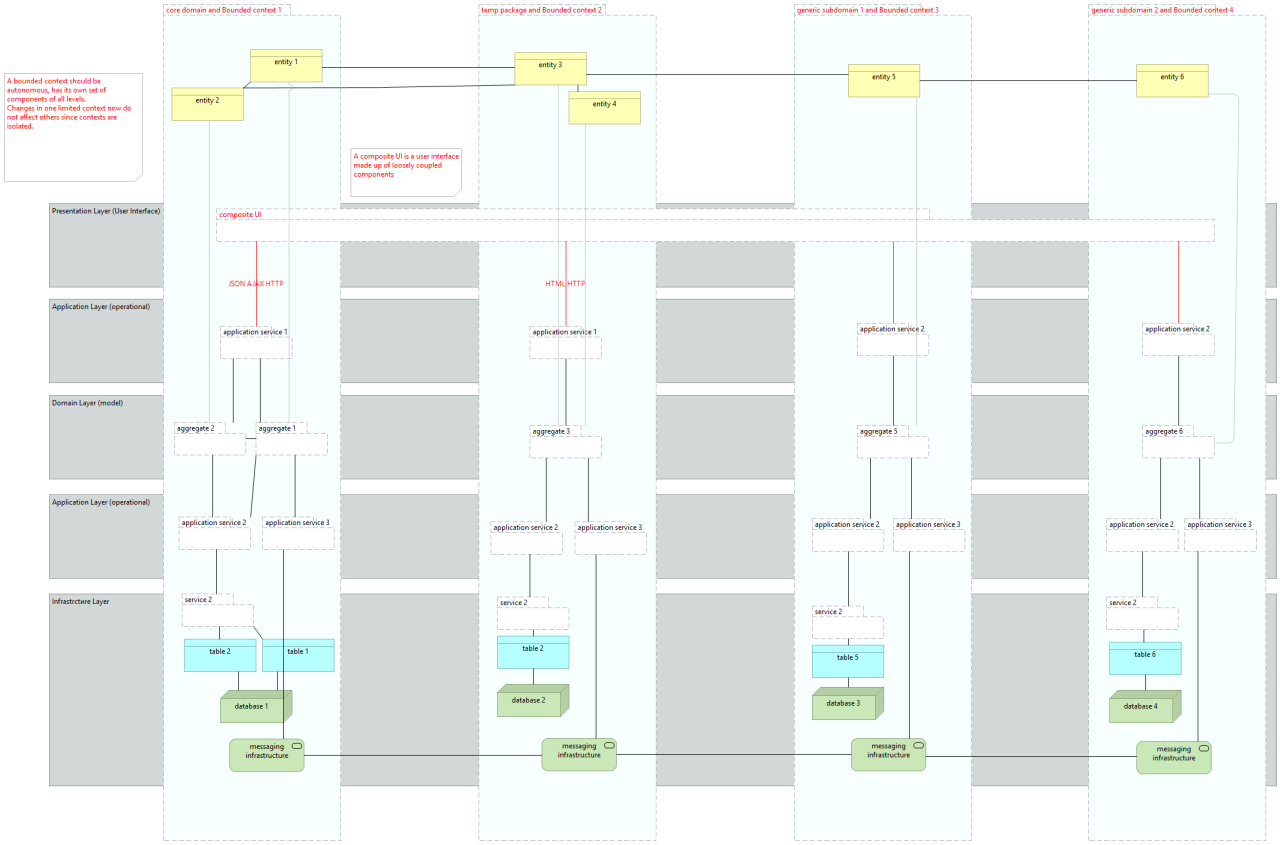



the direction of all dependencies (inheritance or challenges) is strictly towards the center, there are no other directions and there are absolutely no cyclic dependencies
user
infrastructure / presentation layer
input adapter
application service layer
application service
domain layer
domain model
system 1
output adapter
application service
application service
input adapter
system 2
application service
output adapter
database

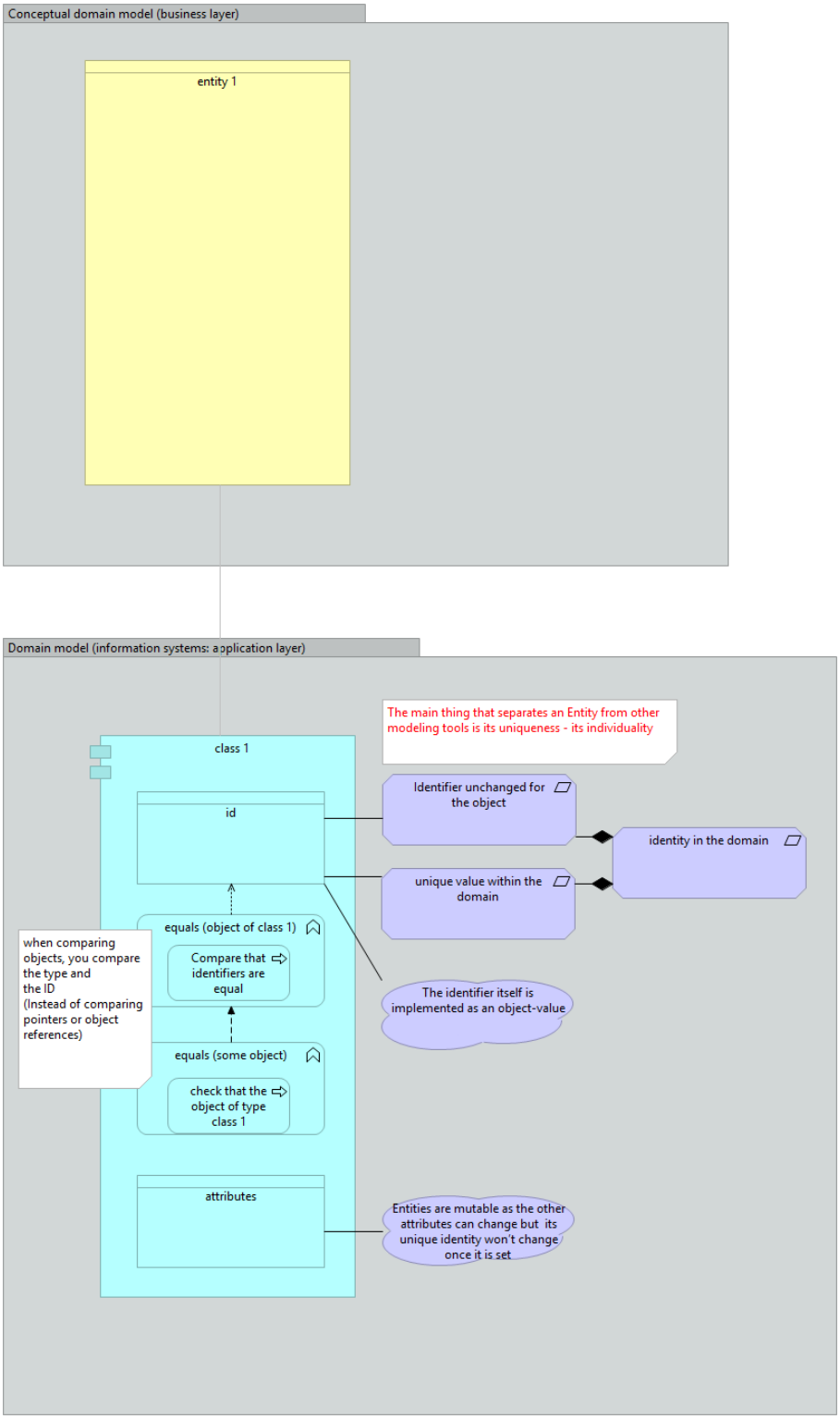

application service layer
input adapter
it is shown how to receive calls from dependent modules and transfer calls to dependent modules
domain layer
domain model
receiver
method1
Interface
method1
caller
method call
Interface
method2
Value-object
output adapter
input adapter
caller
The lower-level classes do not call the upper-level classes directly, but through a protective interface (which hides the features of the main-level implementation)
method call
some class
some va
lower levels freely use DTO, Value-objects defined on the upper level
output adapter
Implementation / receiver
method2
lower level classes implement upper level interfaces.
see patterns: Separated Interface, Dependency Inversion

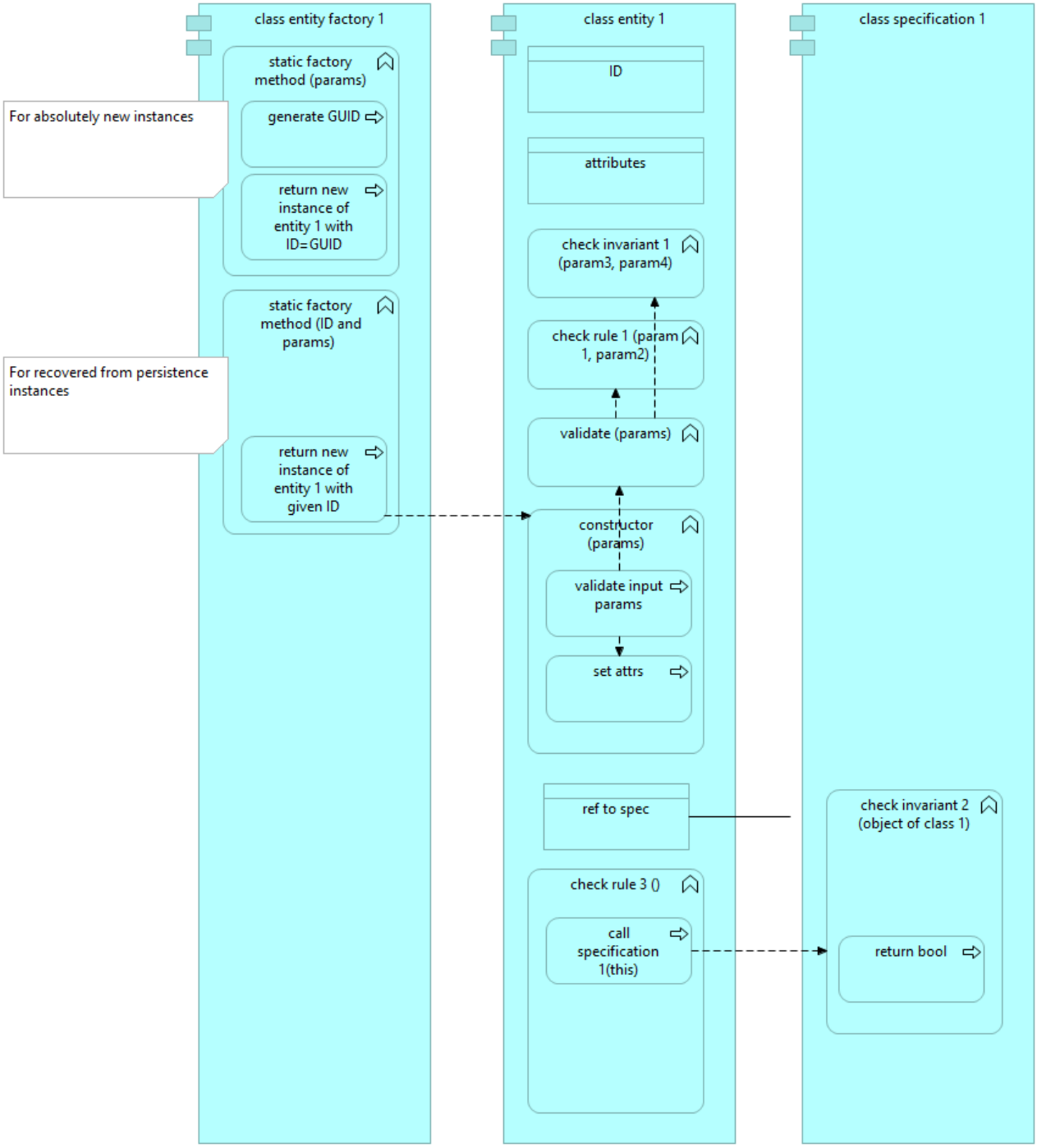

application service layer
input adapter
the high-level component itself actively controls the interaction of low-level components. In this case, the logical flow of data (information from the input adapter to the receiver) and dependencies(the receiver actively polls the adapter itself) in opposite directions.
domain layer
domain model
receiver
Interface
method1
method call
caller
method call
Interface
method2
input adapter
Implementation / caller
method1
output adapter
lower level classes implement upper level interfaces.
see patterns: Separated Interface, Dependency Inversion
output adapter
Implementation / receiver
method2

COMPOSITE UI

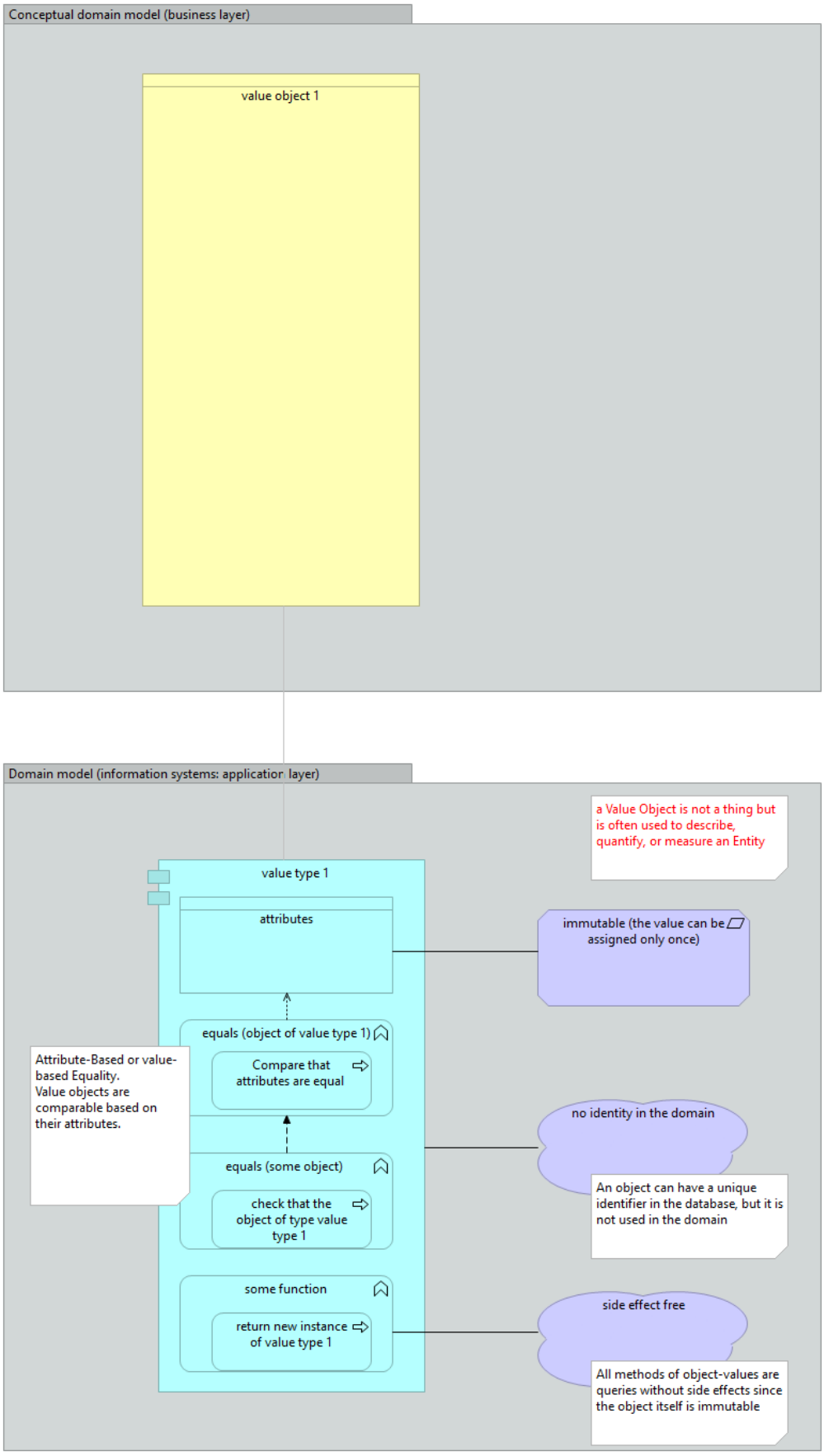


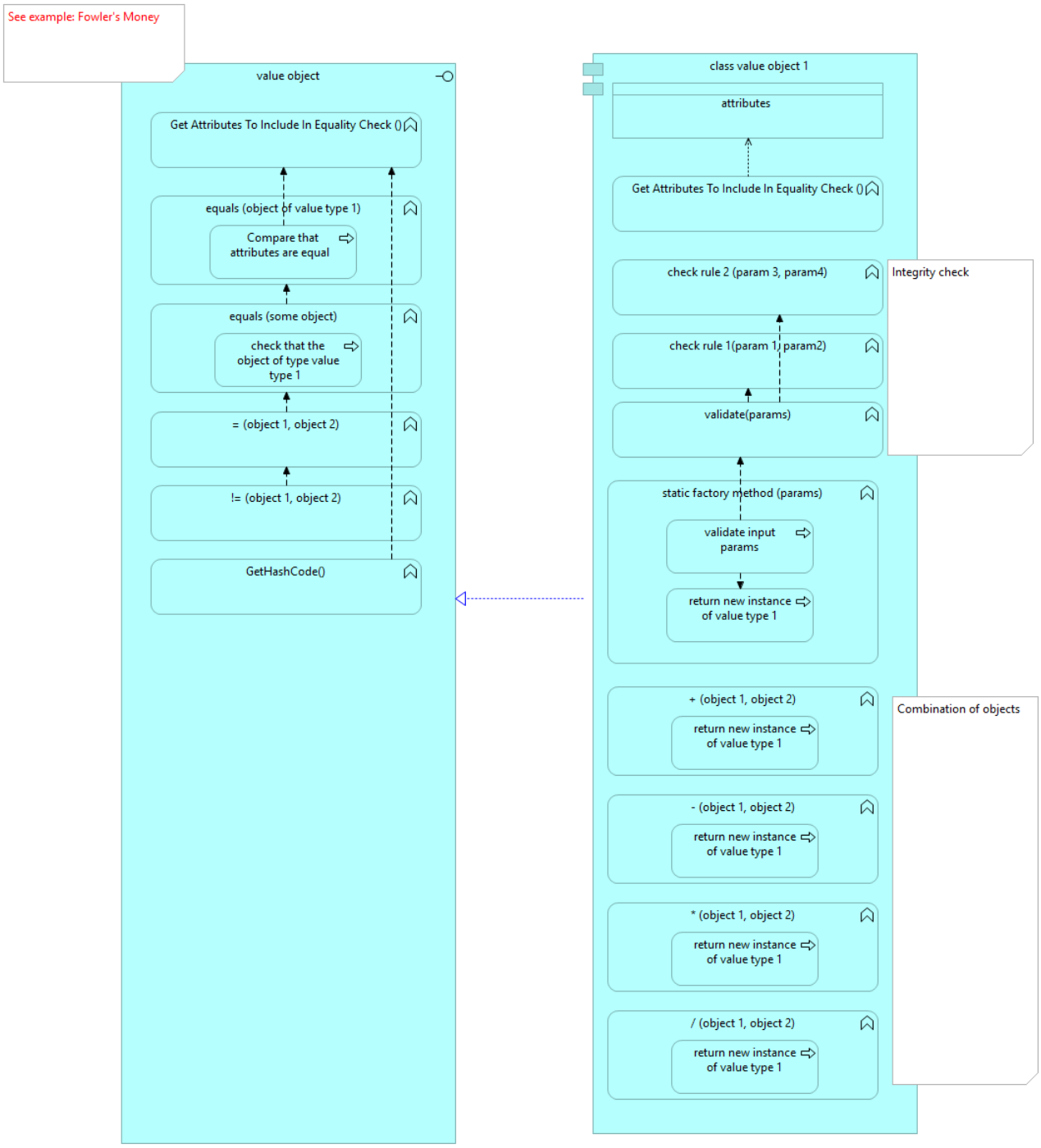
ENTITIES


Conceptual domain model (business layer)
entity 1
Domain model (information systems: application layer)
The main thing that separates an Entity from other modeling tools is its uniqueness - its individuality
class 1
id
Identifier unchanged for the object
identity in the domain
unique value within the domain
equals (object of class 1)
Compare that identifiers are equal
when comparing objects, you compare the type and the ID (Instead of comparing pointers or object references)
The identifier itself is implemented as an object-value
equals (some object)
check that the object of type class 1
attributes
Entities are mutable as the other attributes can change but its unique identity won't change once it is set

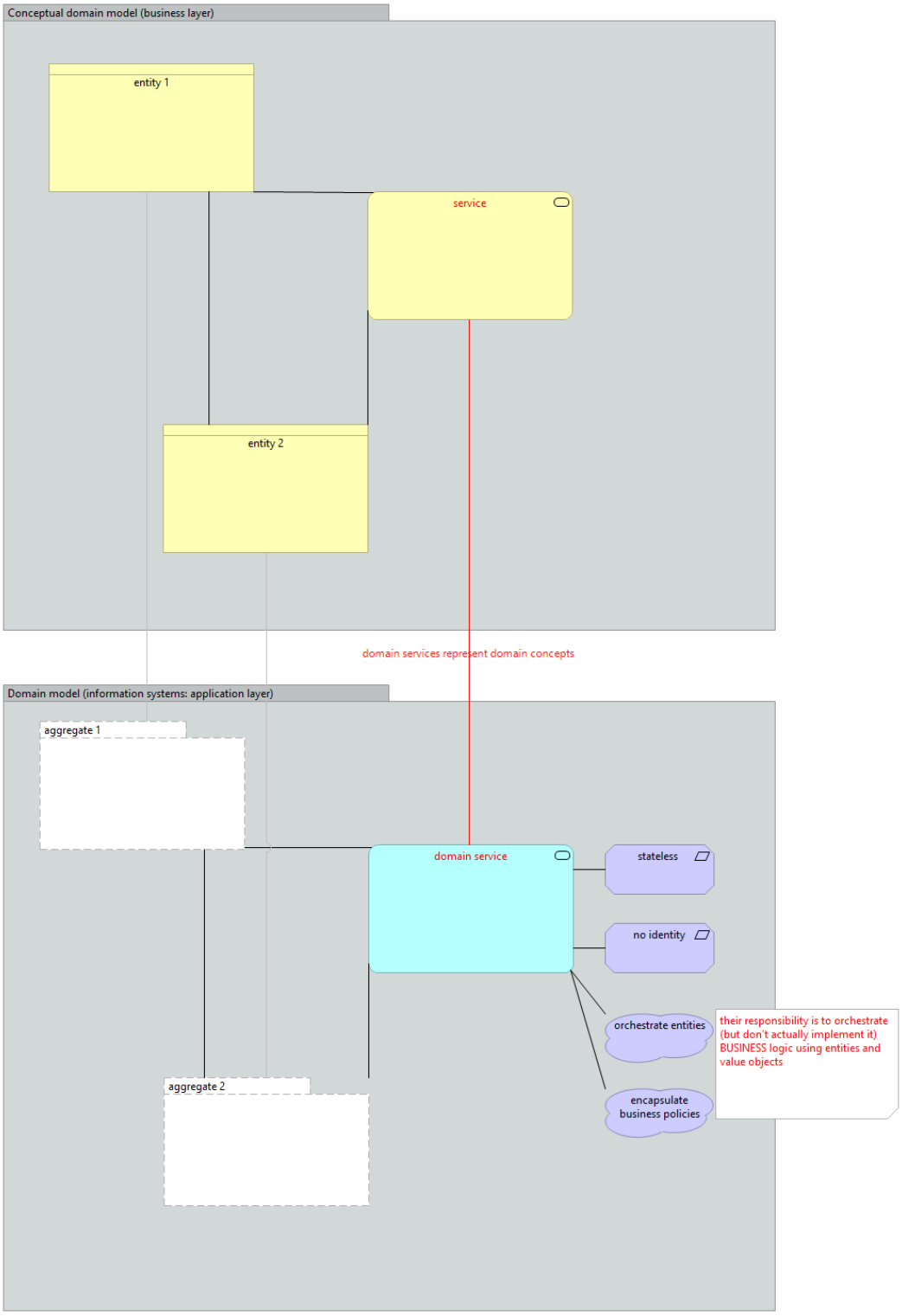

class entity factory 1
static factory method (params)
generate GUID
return new instance of entity 1 with ID=GUID
For absolutely new instances
static factory method (ID and params)
return new instance of entity 1 with given ID
For recovered from persistence instances
class entity 1
ID
attributes
check invariant 1 (param3, param4)
check rule 1 (param 1, param2)
validate (params)
constructor (params)
validate input params
set attrs
ref to spec
check rule 3 ()
call specification 1(this)
class specification 1
check invariant 2 (object of class 1)
return bool

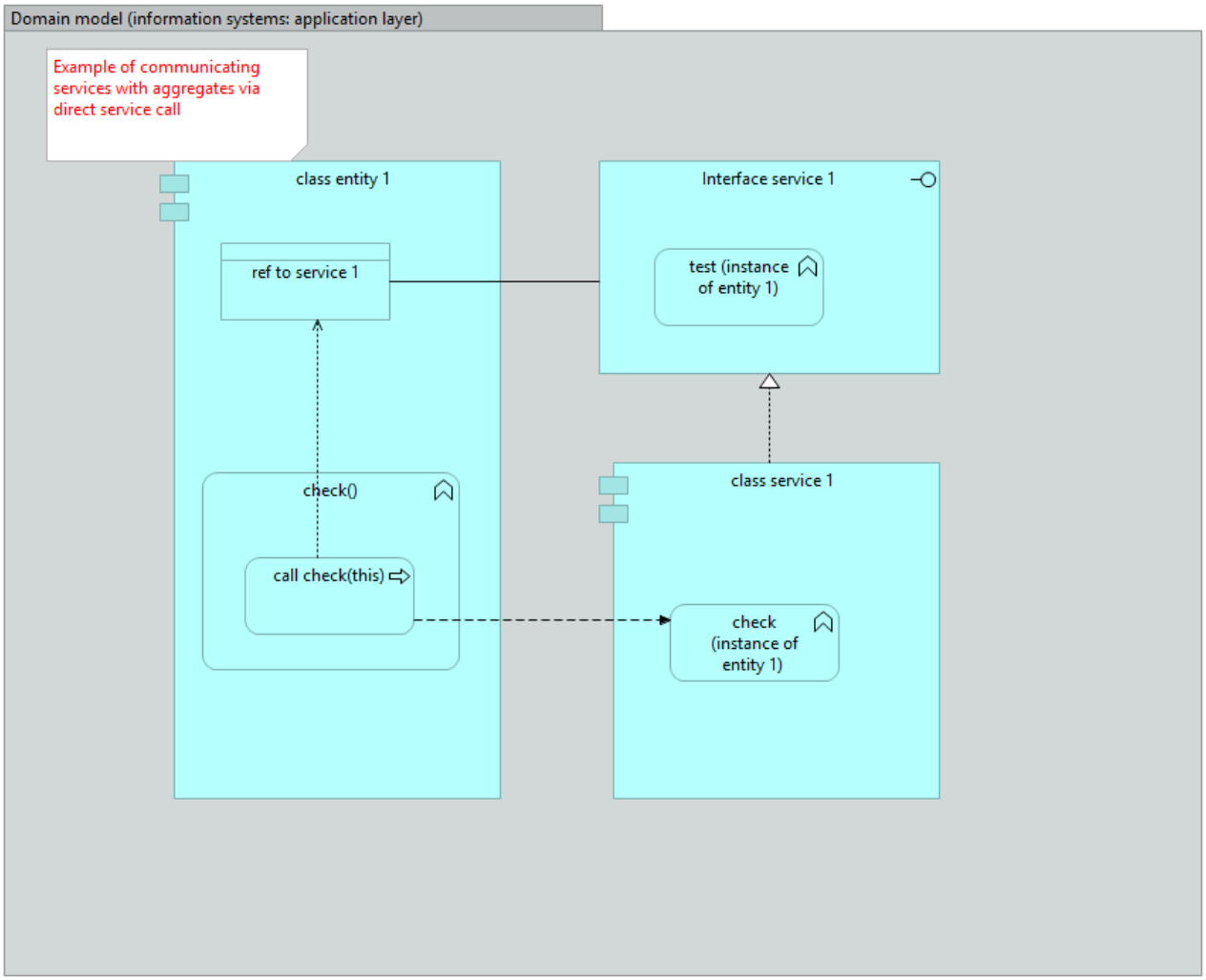
VALUE-OBJECTS



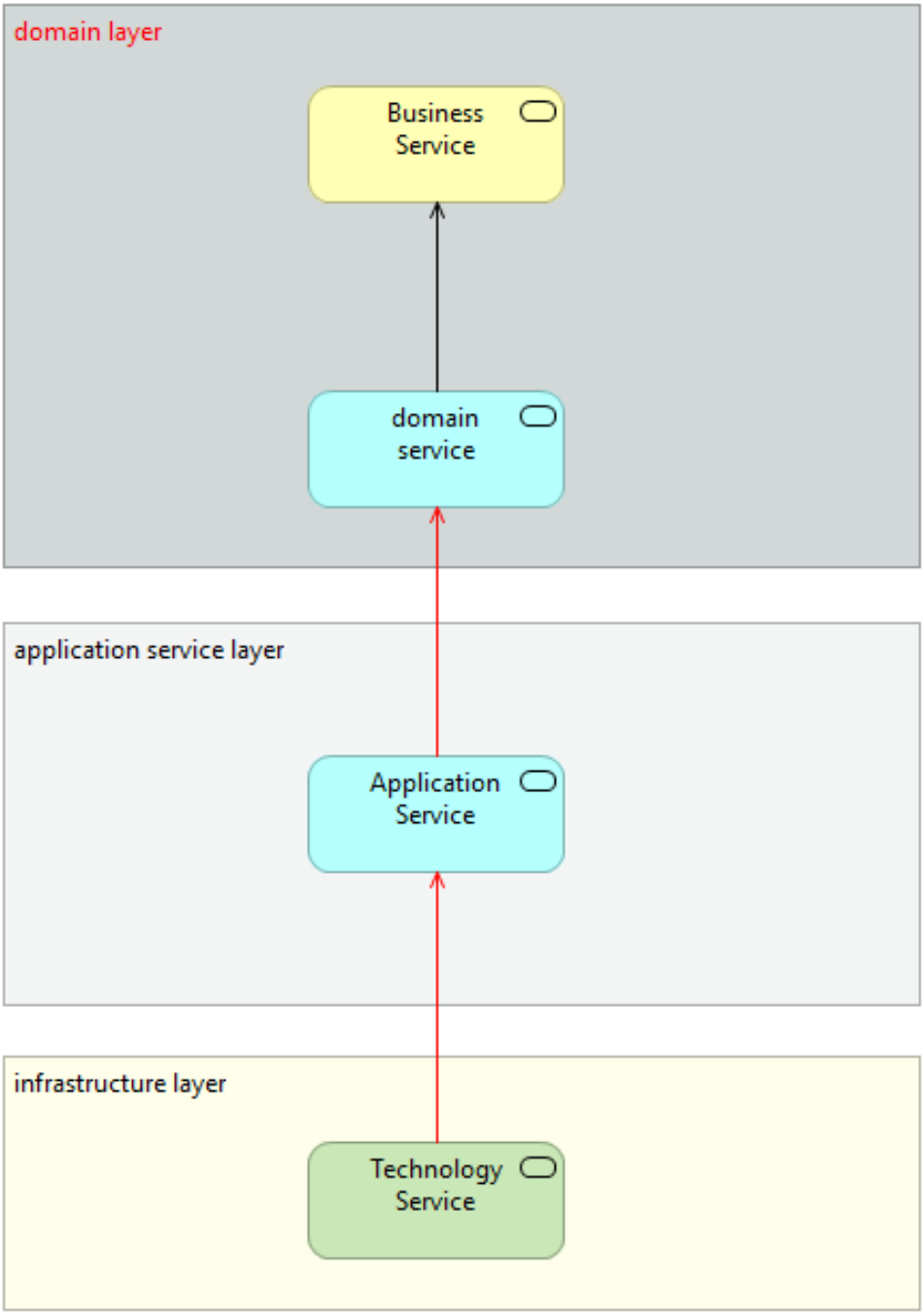

See example: Fowler's Money
value object
Get Attributes To Include In Equality Check ()
equals (object of value type 1)
Compare that attributes are equal
equals (some object)
check that the object of type value type 1
= (object 1, object 2)
!= (object 1, object 2)
GetHashCode()
class value object 1
attributes
Get Attributes To Include In Equality Check ()
check rule 2 (param 3, param4)
check rule 1(param 1, param2)
validate(params)
Integrity check
static factory method (params)
validate input params
return new instance of value type 1
+ (object 1, object 2)
return new instance of value type 1
- (object 1, object 2)
return new instance of value type 1
* (object 1, object 2)
return new instance of value type 1
/ (object 1, object 2)
return new instance of value type 1
Combination of objects

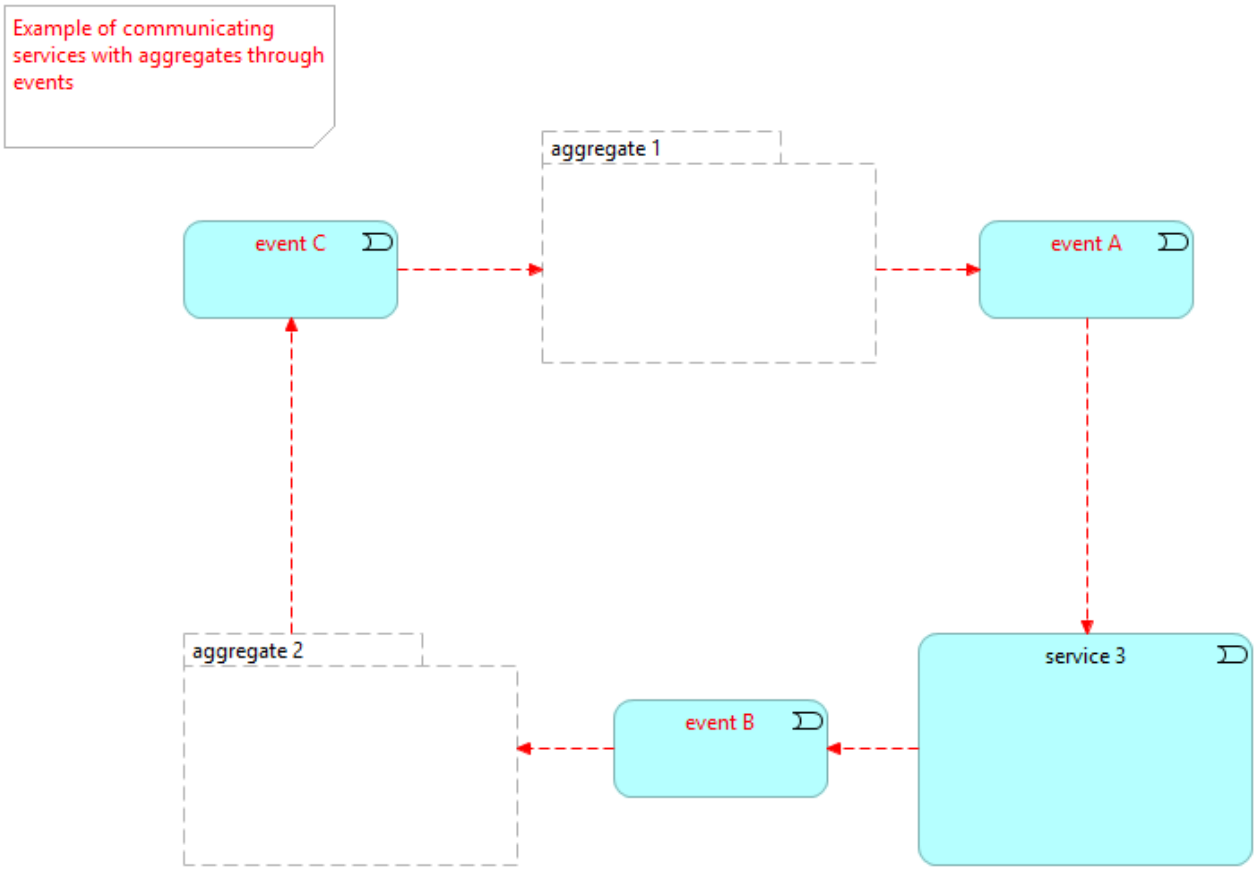
DOMAIN SERVICES

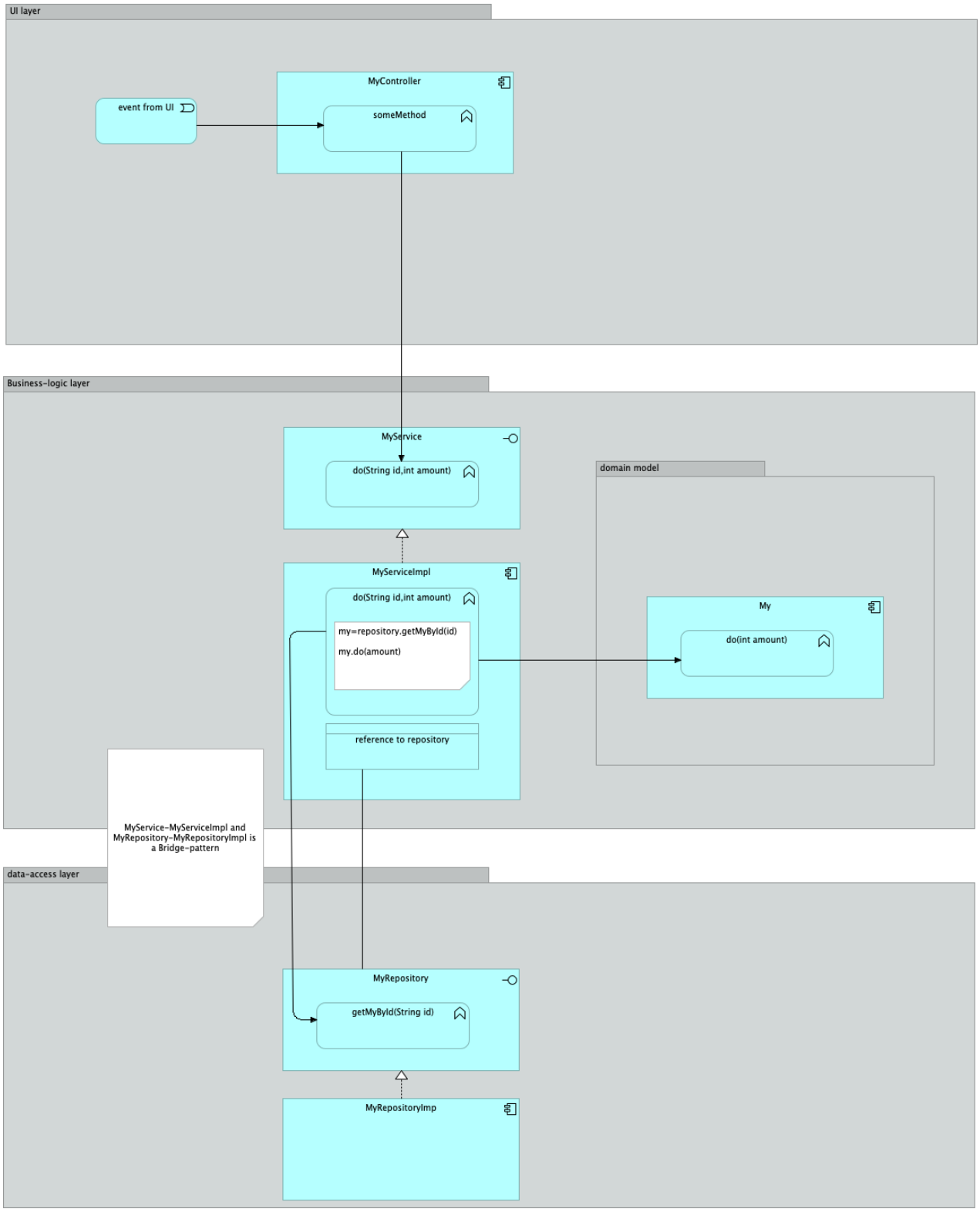



Domain model (information systems: application layer)
Example of communicating services with aggregates via direct service call
class entity 1
ref to service 1
check()
call check(this)
Interface service 1
test (instance of entity 1)
class service 1
check (instance of entity 1)

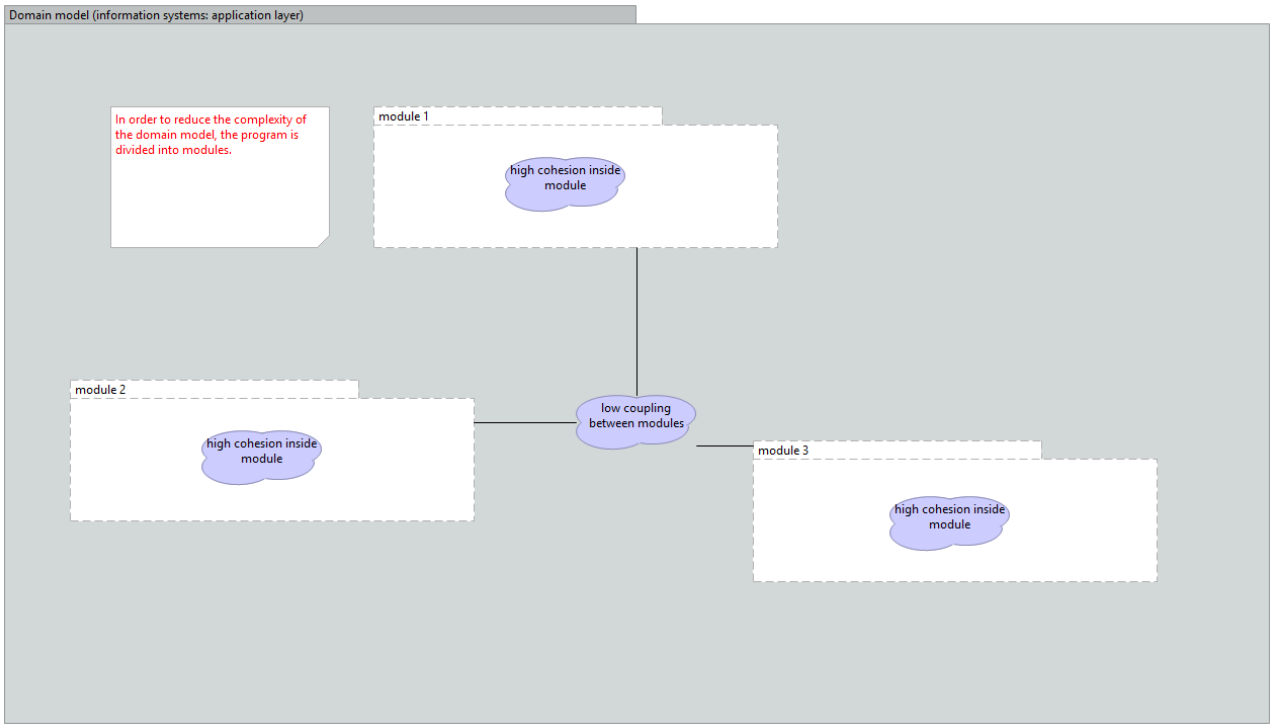
Services support each other



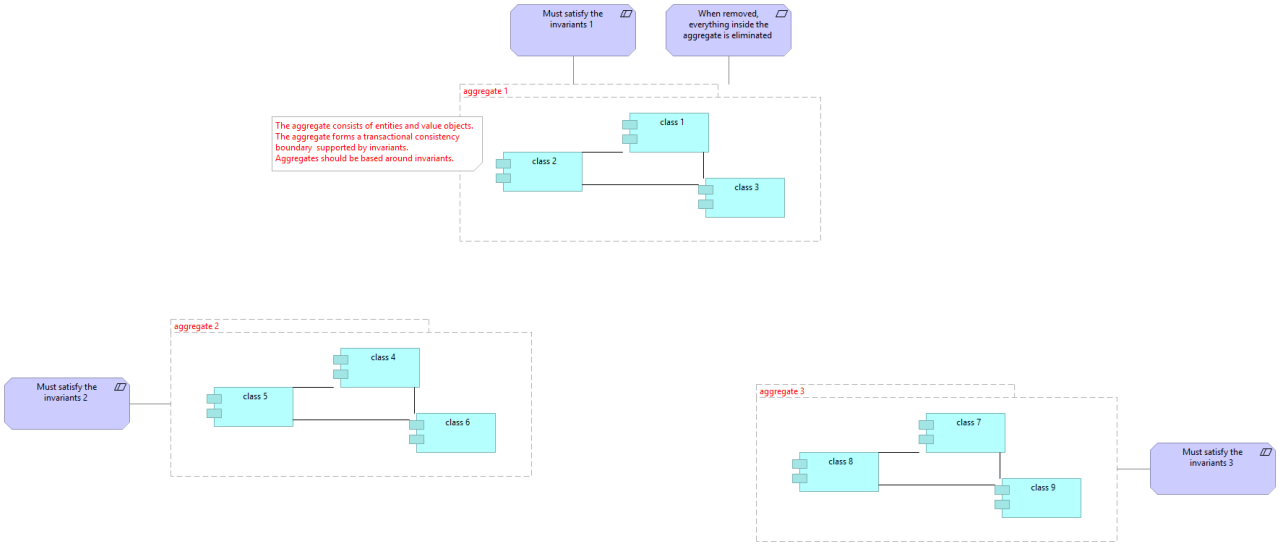

Example of communicating services with aggregates through events
aggregate 1
event C
event A
aggregate 2
service 3
event B


UI layer
event from UI
MyController
someMethod
Business-logic layer
MyService
do(String id,int amount)
domain model
MyServiceImpl
do(String id,int amount)
my=repository.getMyById(id)
my.do(amount)
reference to repository
My
do(int amount)
MyService-MyServiceImpl and MyRepository-MyRepositoryImpl is a Bridge-pattern
data-access layer
MyRepository
getMyById(String id)
MyRepositoryImp

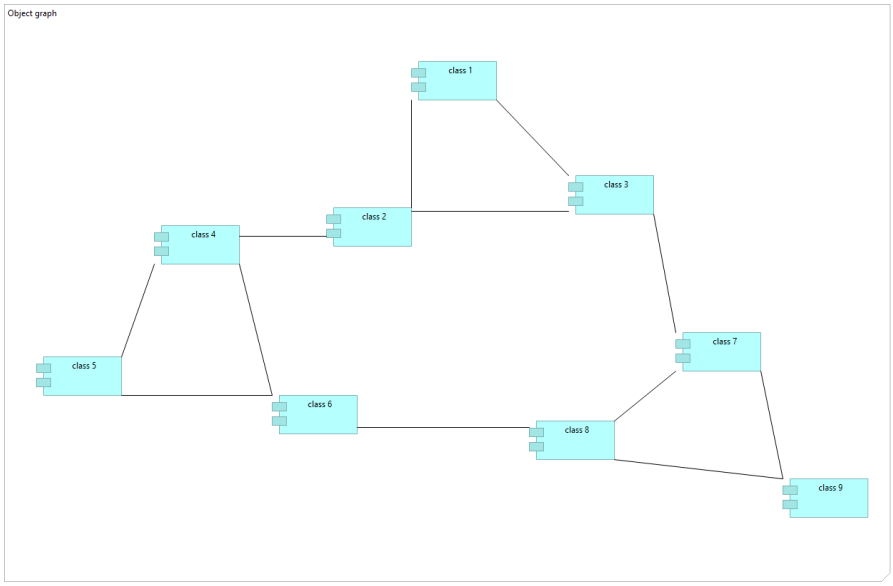
MODULES

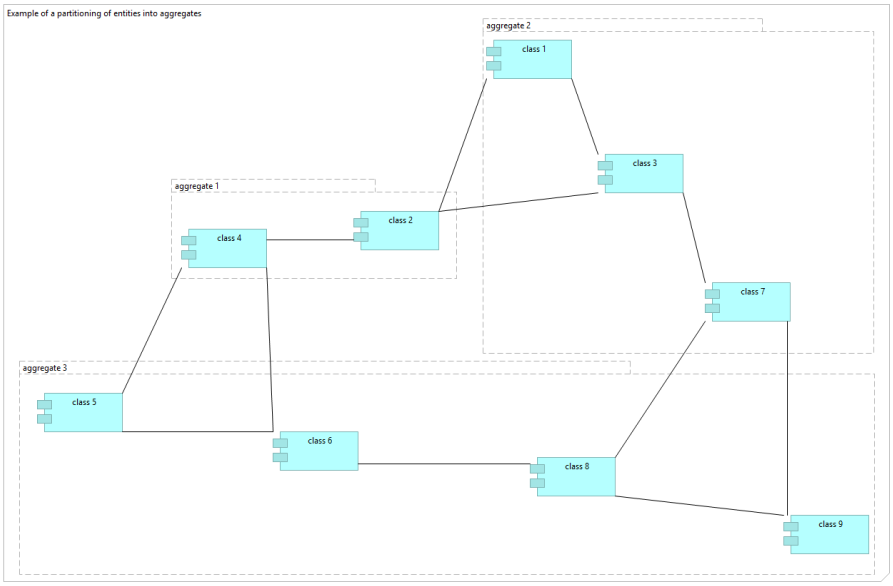

Domain model (information systems: application layer)
In order to reduce the complexity of the domain model, the program is divided into modules.
module 1
high cohesion inside module
module 2
high cohesion inside module
low coupling between modules
module 3
high cohesion inside module

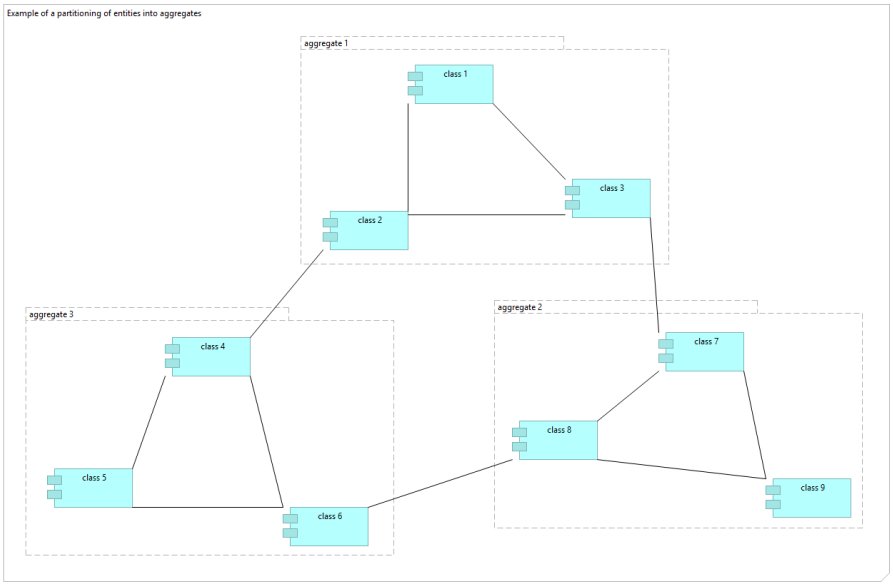
AGGREGATES


Must satisfy the invariants 1
When removed, everything inside the aggregate is eliminated
aggregate 1
The aggregate consists of entities and value objects.
The aggregate forms a transactional consistency boundary supported by invariants.
Aggregates should be based around invariants.
class 1
class 2
class 3
aggregate 2
Must satisfy the invariants 2
class 4
class 5
class 6
aggregate 3
class 7
class 8
class 9
Must satisfy the invariants 3

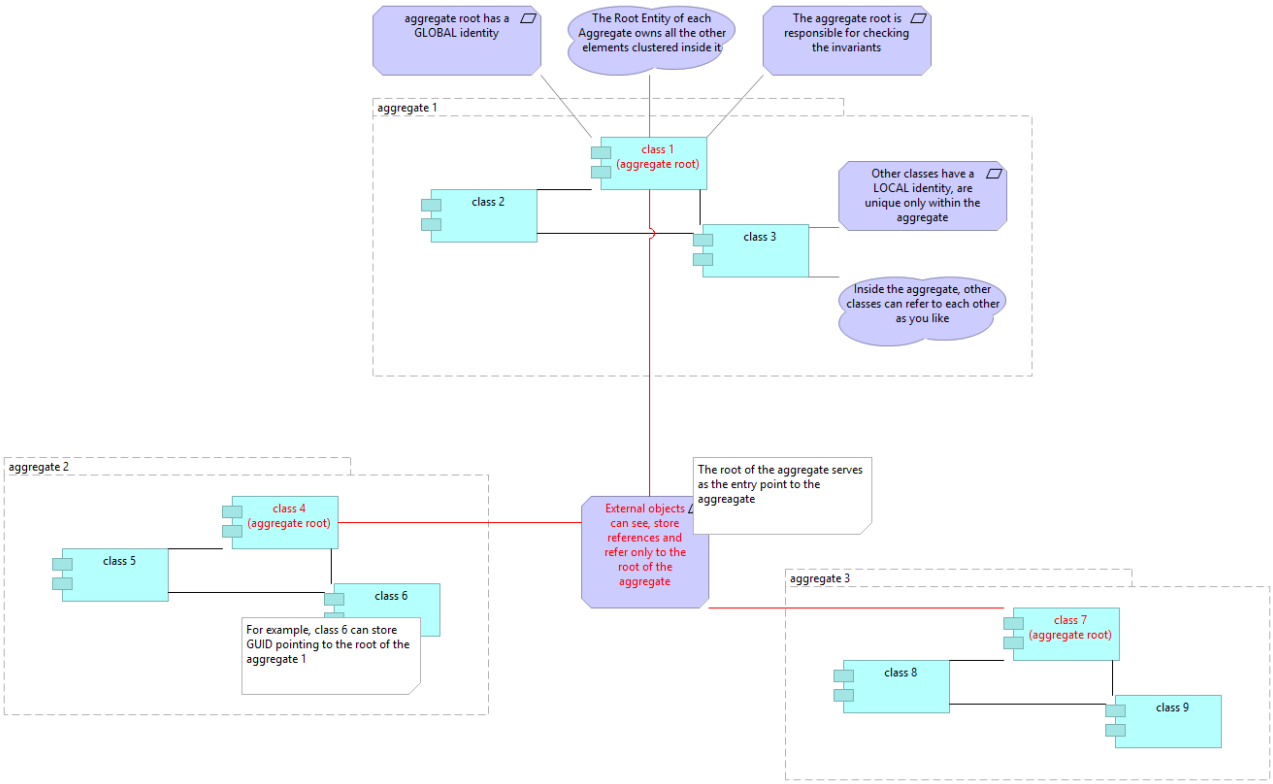
In the course of modeling, think about different ways of splitting the model into aggregates



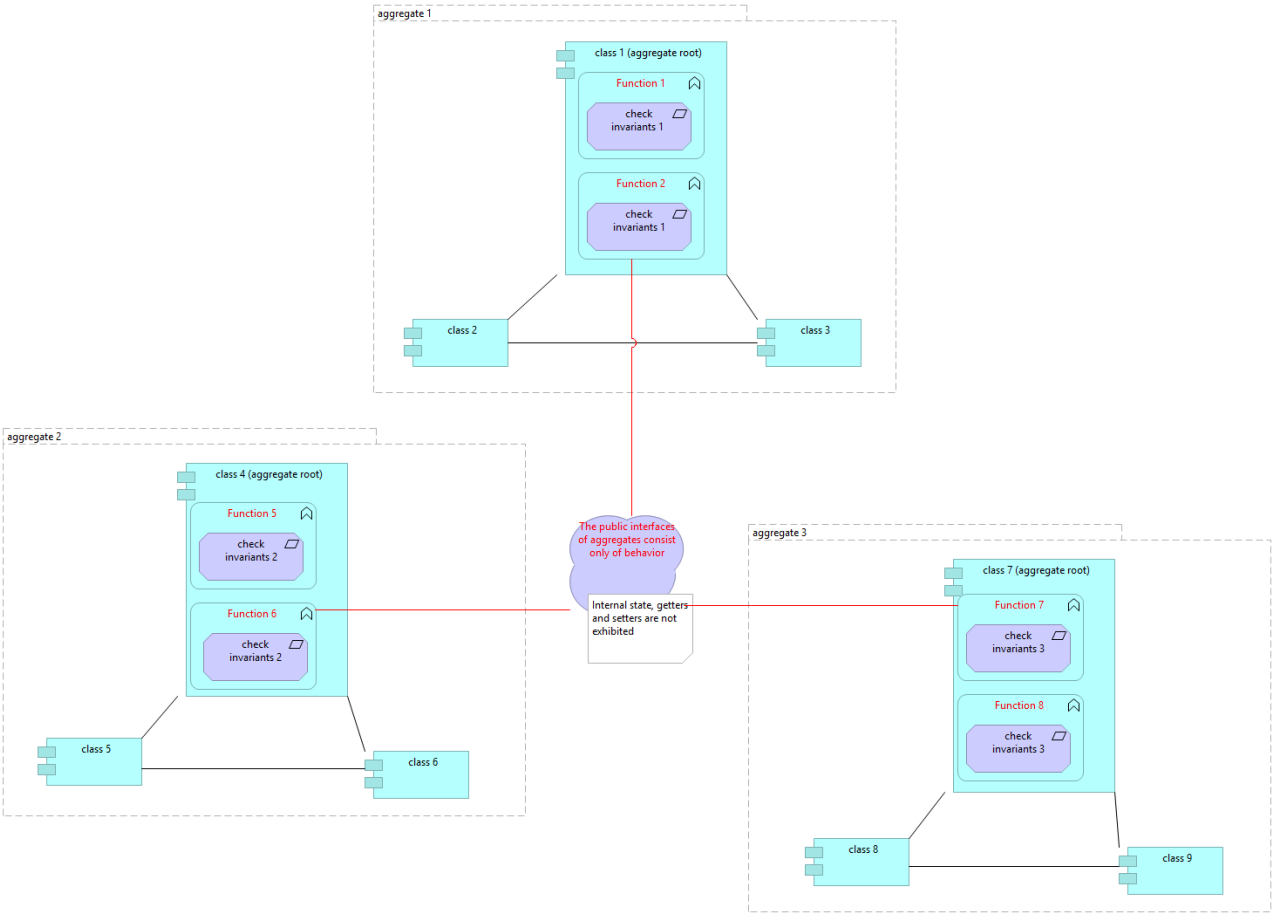




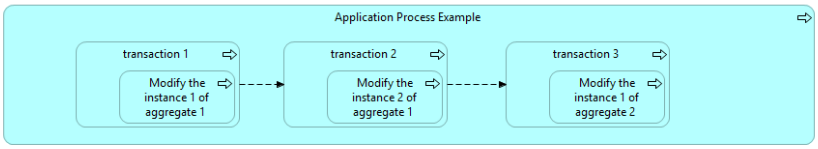
AGGREGATE ROOT

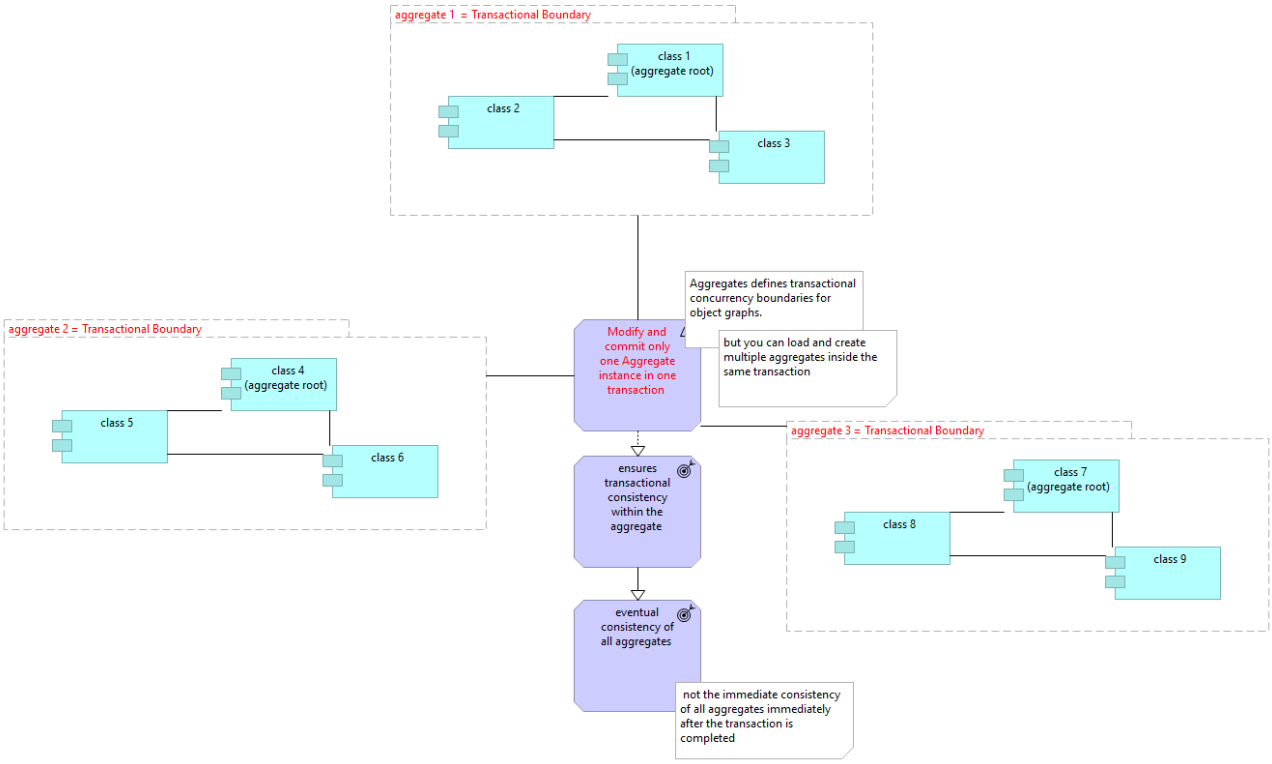


BEHAVIOR-FOCUSED AGGREGATE ROOT

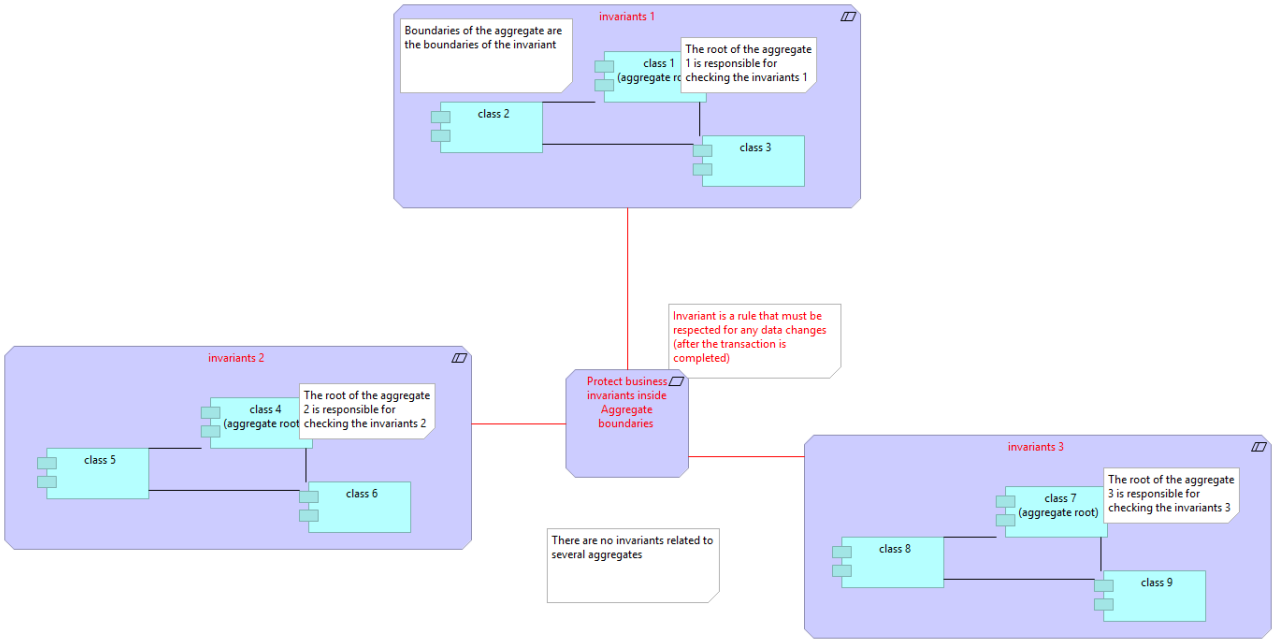


MODIFY AND COMMIT ONLY ONE AGGREGATE INSTANCE IN ONE TRANSACTION

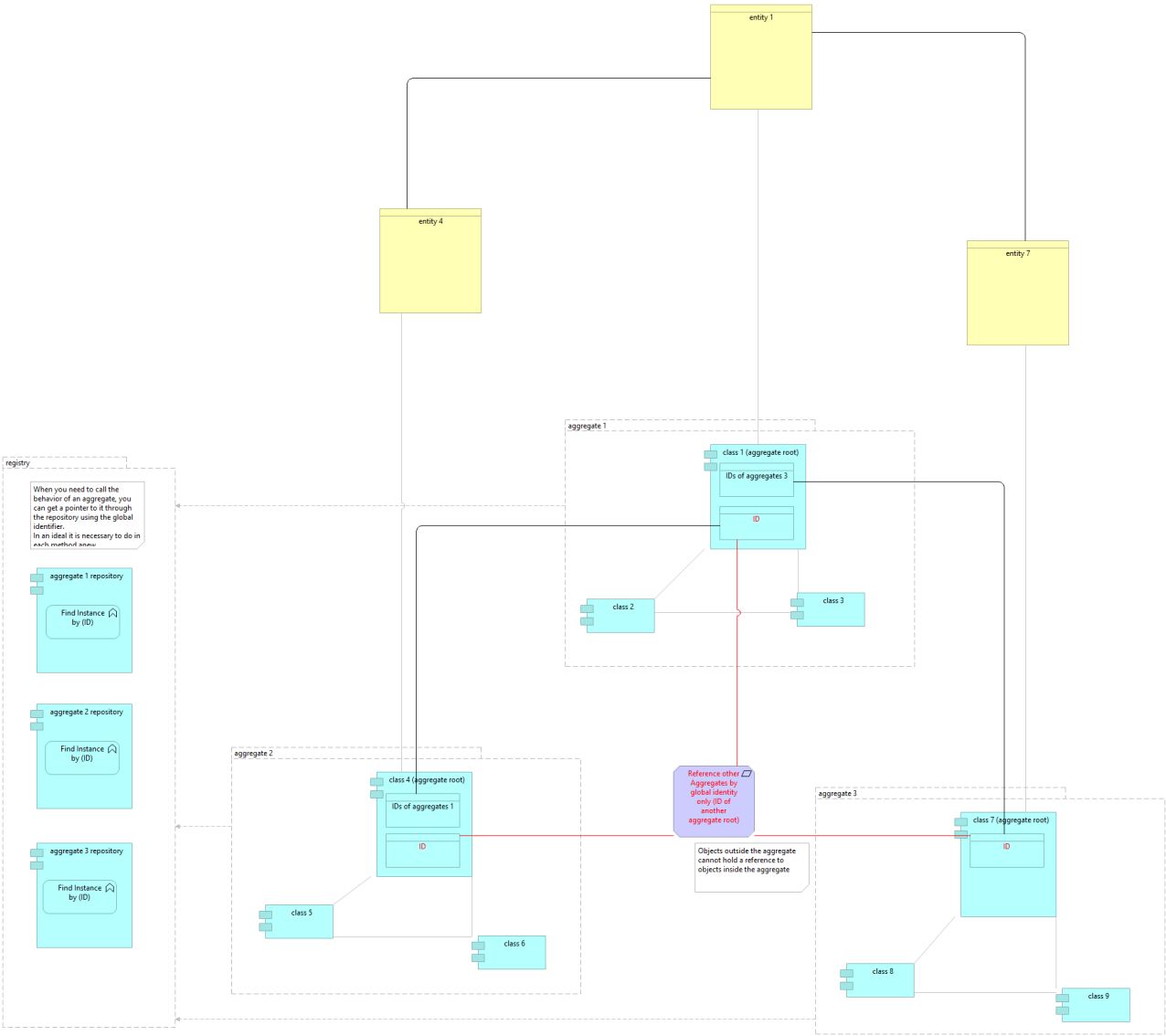




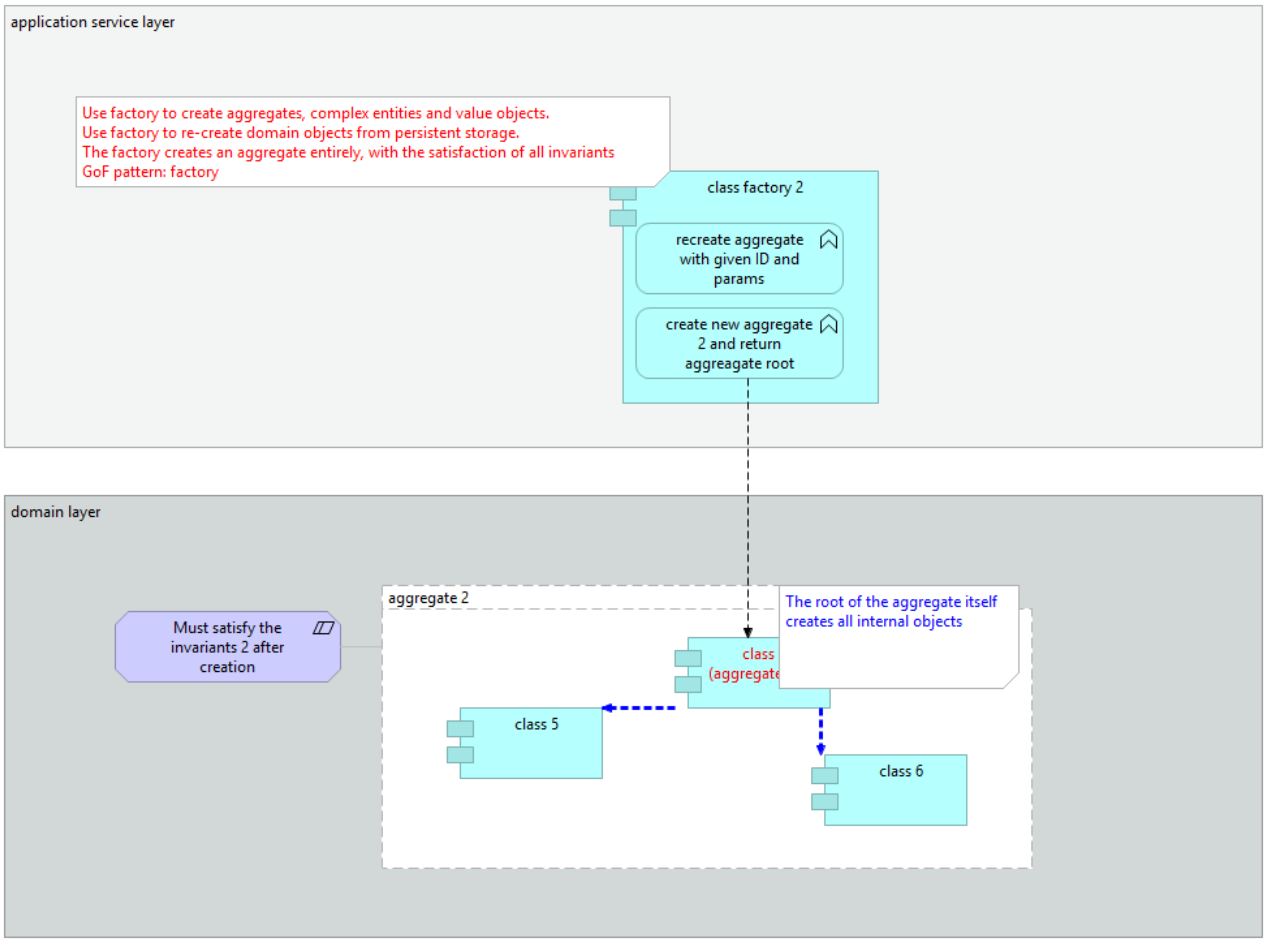
PROTECT BUSINESS INVARIANTS INSIDE AGGREGATE BOUNDARIES

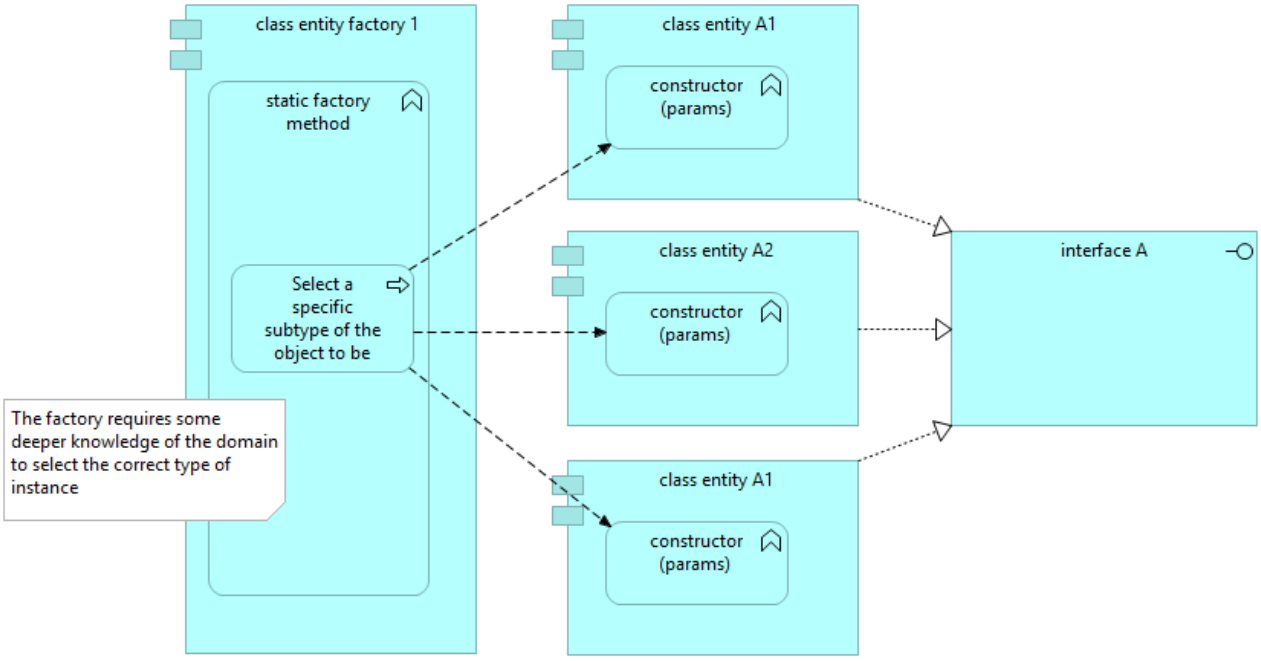


REFERENCE OTHER AGGREGATES BY IDENTITY ONLY

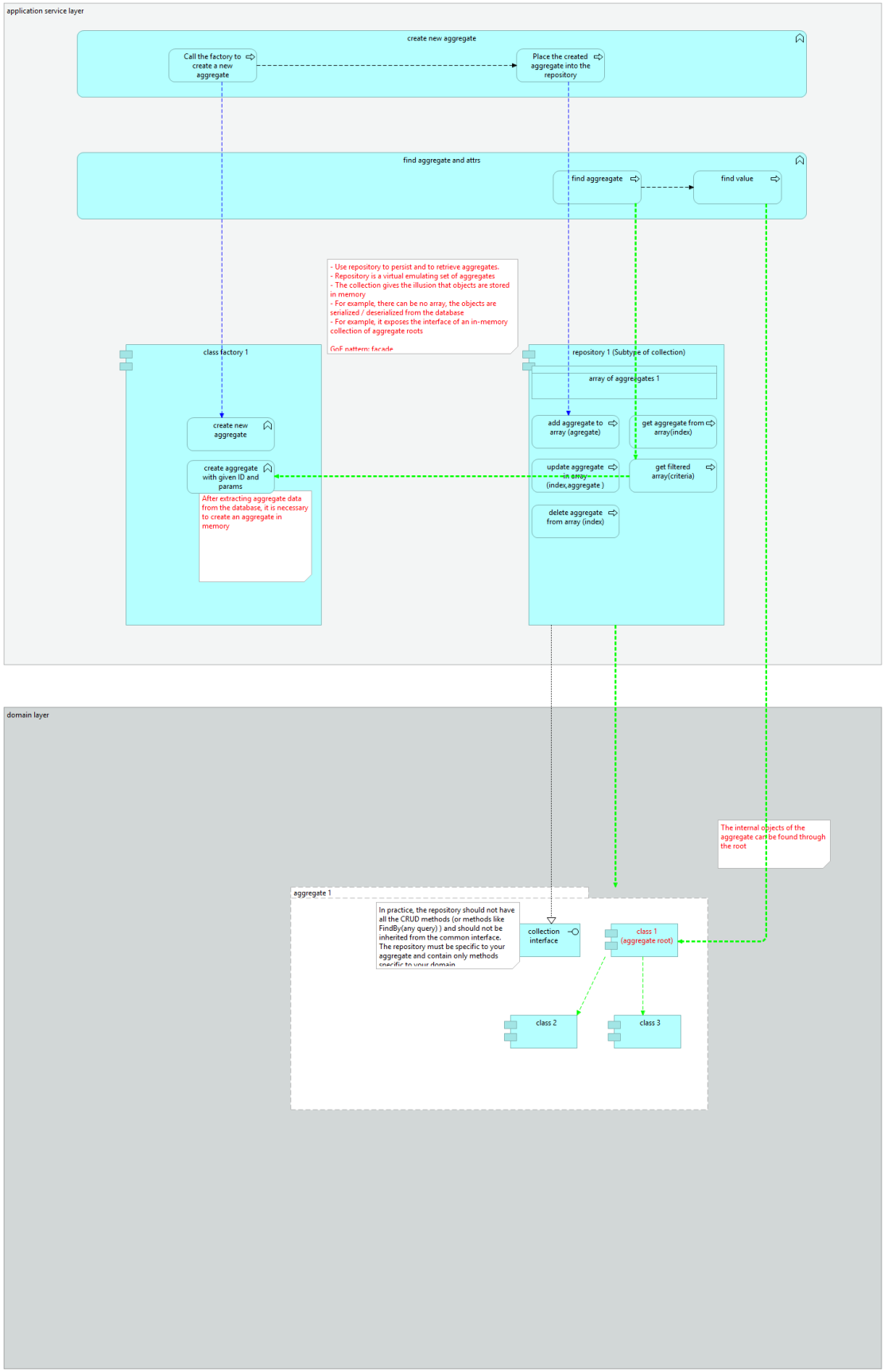


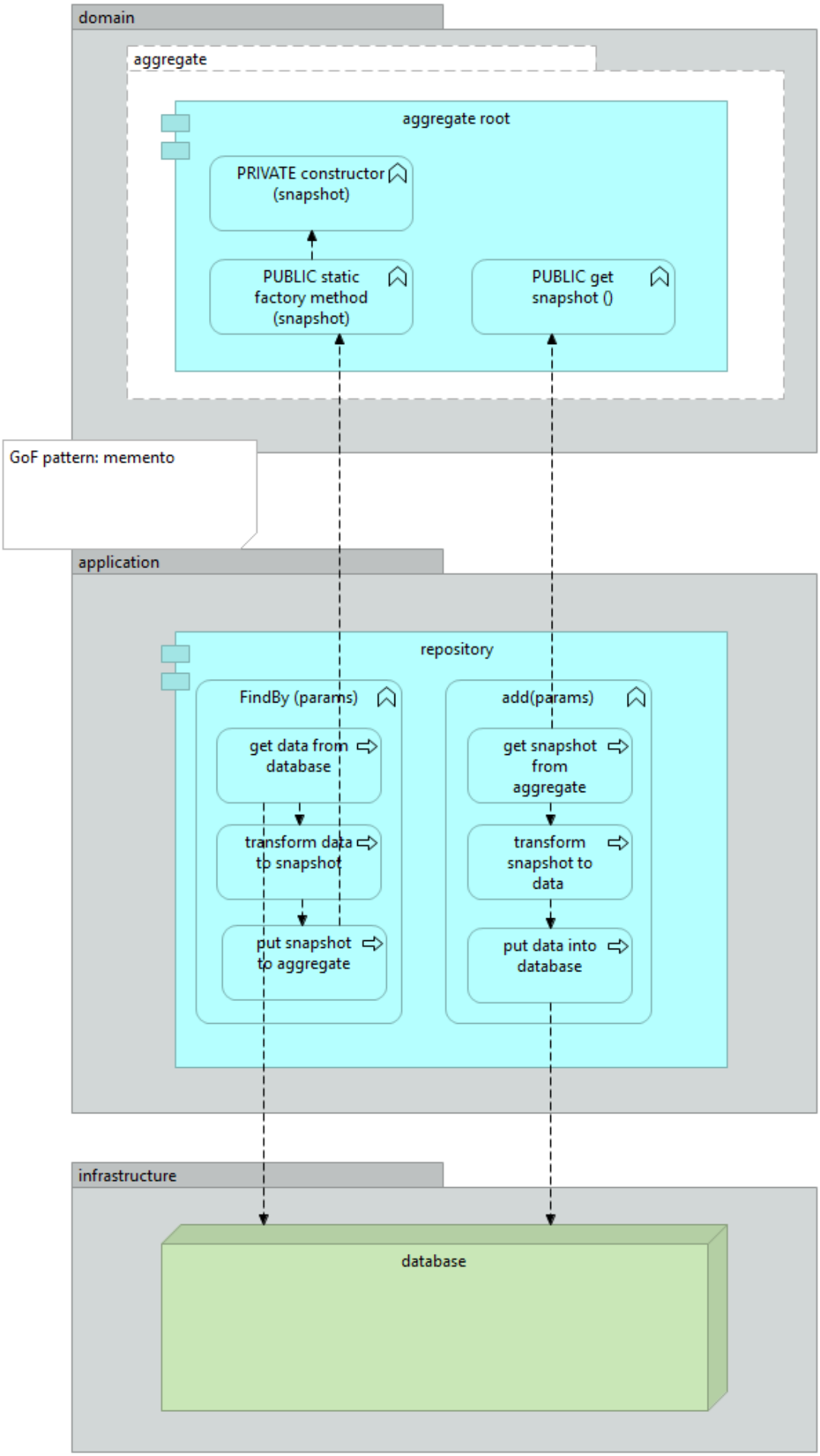
FACTORIES

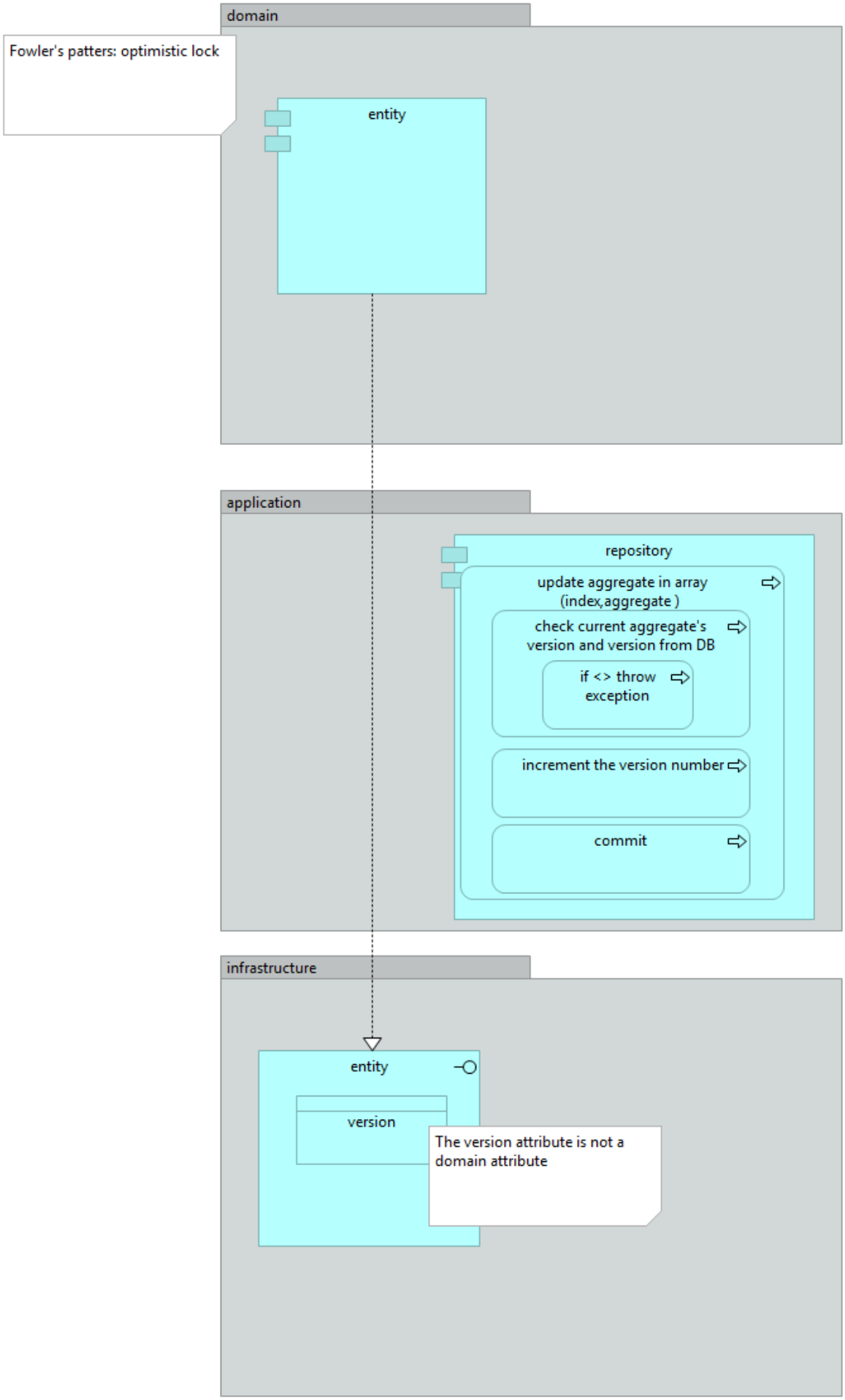

application service layer
Use factory to create aggregates, complex entities and value objects.
Use factory to re-create domain objects from persistent storage.
The factory creates an aggregate entirely, with the satisfaction of all invariants
GoF pattern: factory
class factory 2
recreate aggregate with given ID and params
create new aggregate 2 and return aggreagate root
domain layer
Must satisfy the invariants 2 after creation
aggregate 2
The root of the aggregate itself creates all internal objects
class
class 5
class 6

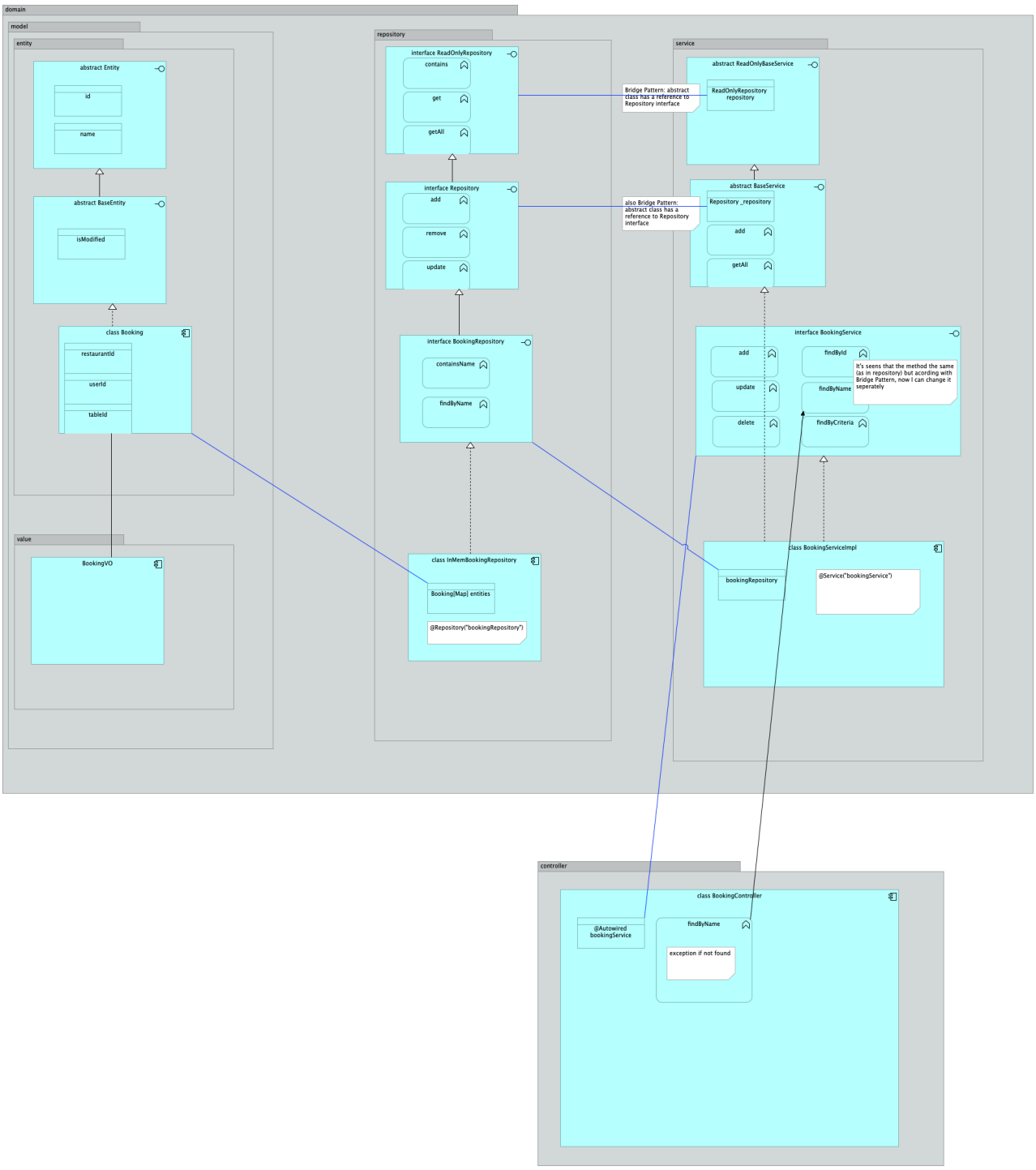

class entity factory 1
static factory method
Select a specific subtype of the object to be
The factory requires some deeper knowledge of the domain to select the correct type of instance
class entity A1
constructor (params)
class entity A2
constructor (params)
class entity A1
constructor (params)
interface A

REPOSITORIES




domain
aggregate
aggregate root
PRIVATE constructor (snapshot)
PUBLIC static factory method (snapshot)
PUBLIC get snapshot ()
GoF pattern: memento
application
repository
FindBy (params)
get data from database
transform data to snapshot
put snapshot to aggregate
add(params)
get snapshot from aggregate
transform snapshot to data
put data into database
infrastructure
database


domain
Fowler's patters: optimistic lock
entity
application
repository
update aggregate in array
(index,aggregate)
check current aggregate's
version and version from DB
if <> throw
exception
increment the version number
commit
infrastructure
entity
version
The version attribute is not a
domain attribute

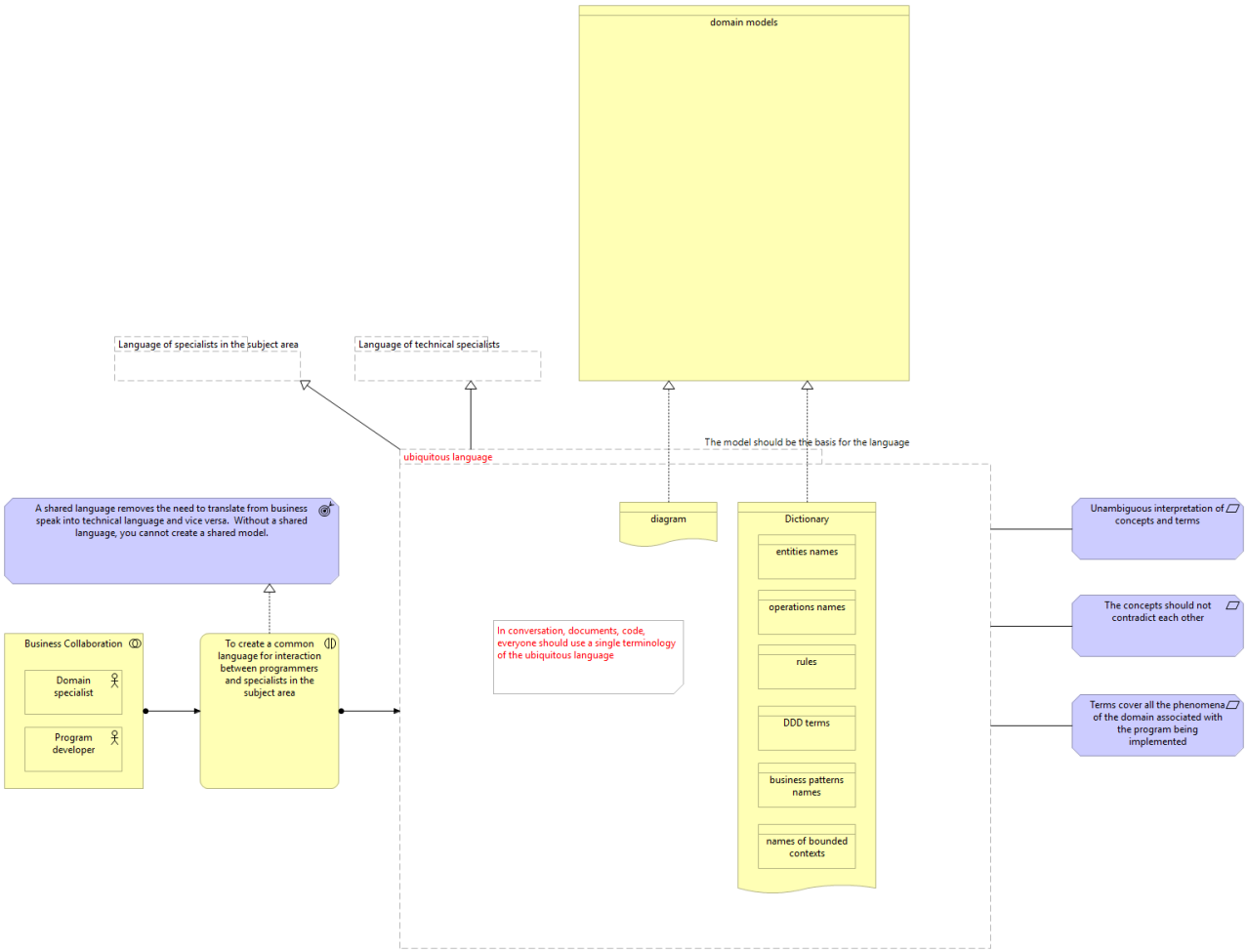

domain
model
entity
abstract Entity
id
name
abstract BaseEntity
isModified
class Booking
restaurantId
userId
tableId
value
BookingVO
repository
interface ReadOnlyRepository
contains
get
getAll
interface Repository
add
remove
update
interface BookingRepository
containsName
findByName
class InMemBookingRepository
Booking[Map] entities
@Repository("bookingRepository")
Bridge Pattern: abstract class has a reference to Repository interface
also Bridge Pattern: abstract class has a reference to Repository interface
service
abstract ReadOnlyBaseService
ReadOnlyRepository repository
abstract BaseService
Repository _repository
add
getAll
interface BookingService
add
update
delete
findById
findByName
findByCriteria
It's seens that the method the same (as in repository) but acording with Bridge Pattern, now I can change it seperately
class BookingServiceImpl
bookingRepository
@Service("bookingService")
controller
class BookingController
@Autowired bookingService
findByName
exception if not found

SUPPLE DESIGN

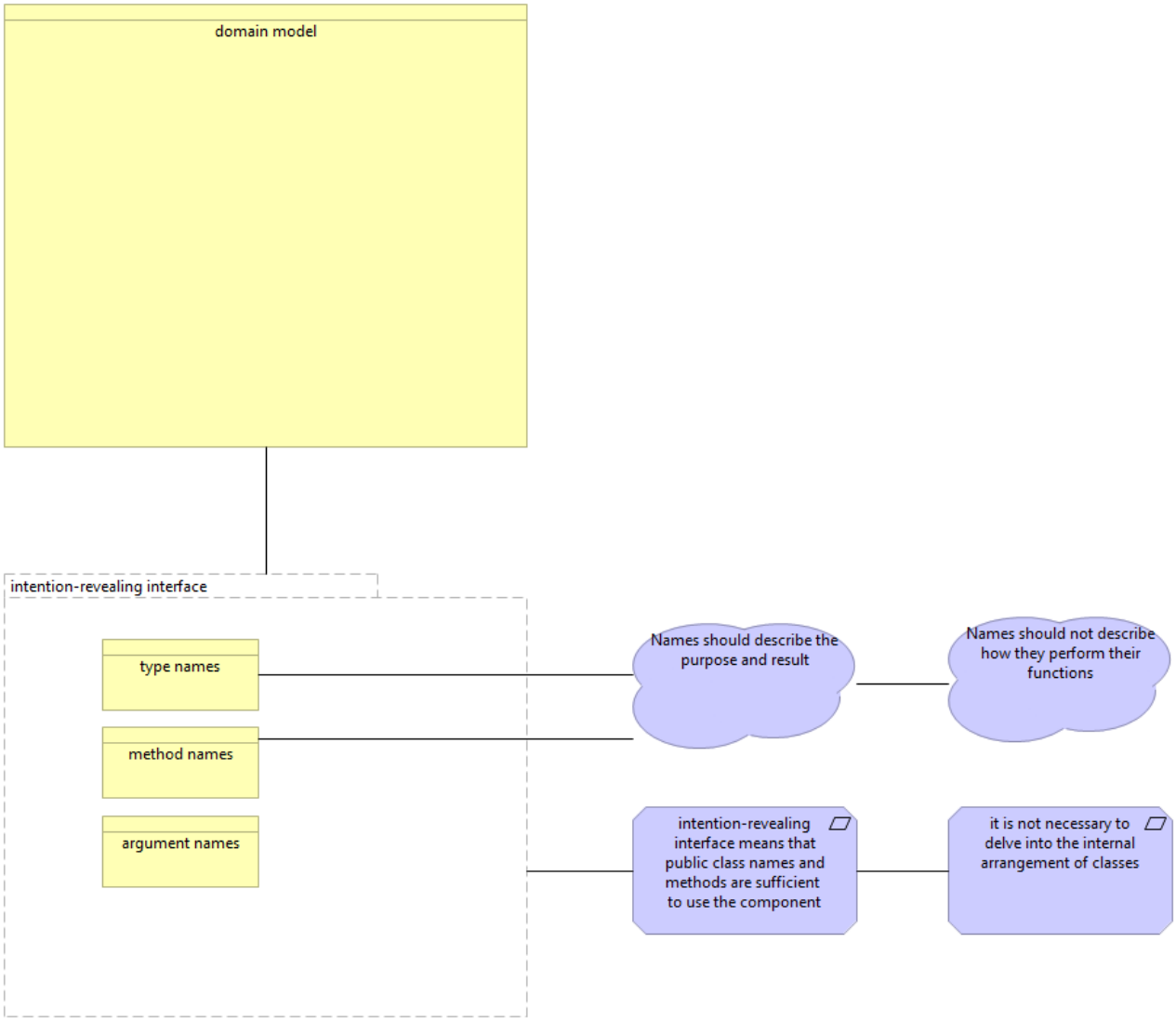
DOMAIN DRIVEN DESIGN

Eric Evans

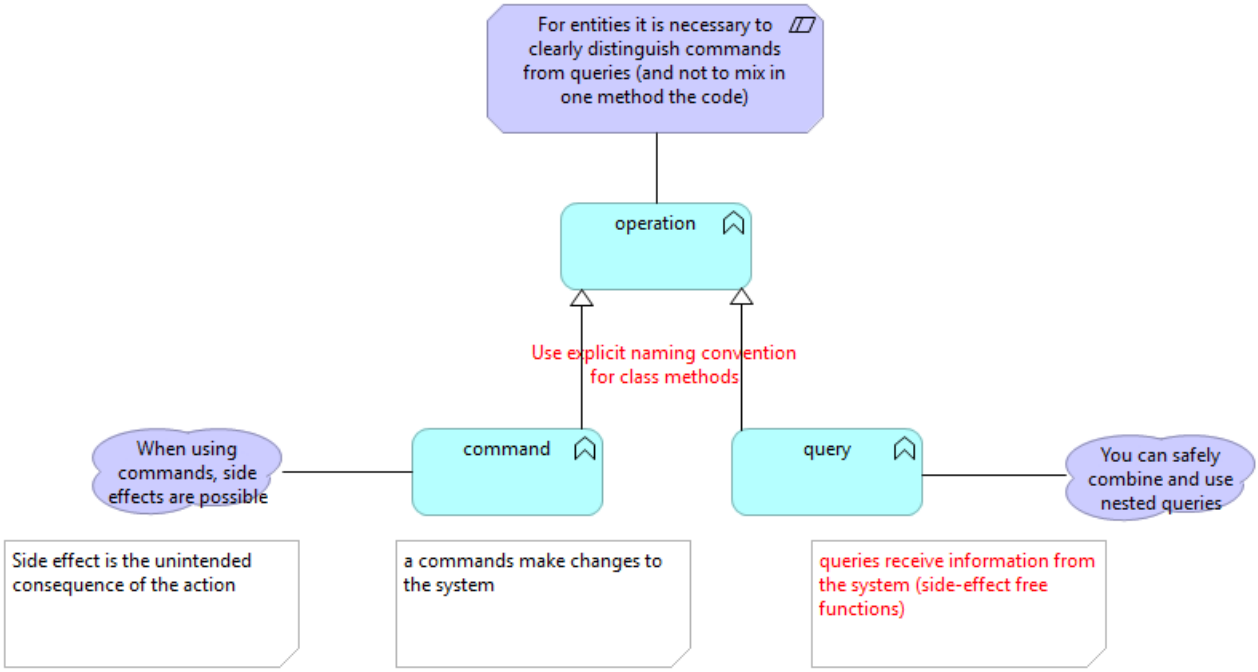
UBIQUITOUS LANGUAGE



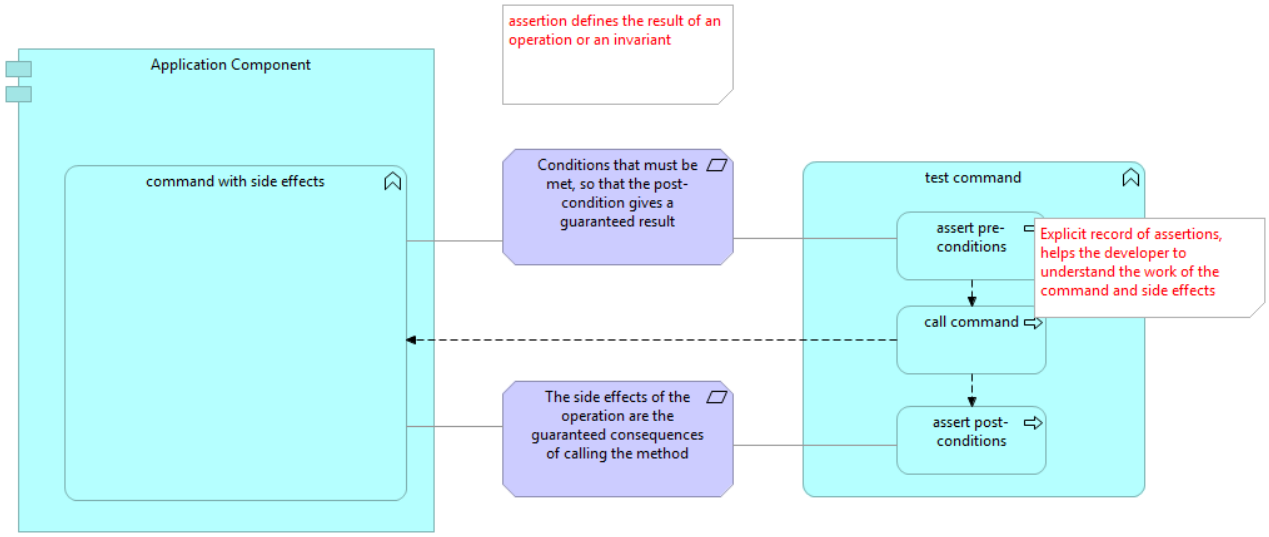
INTENTION-REVEALING INTERFACES



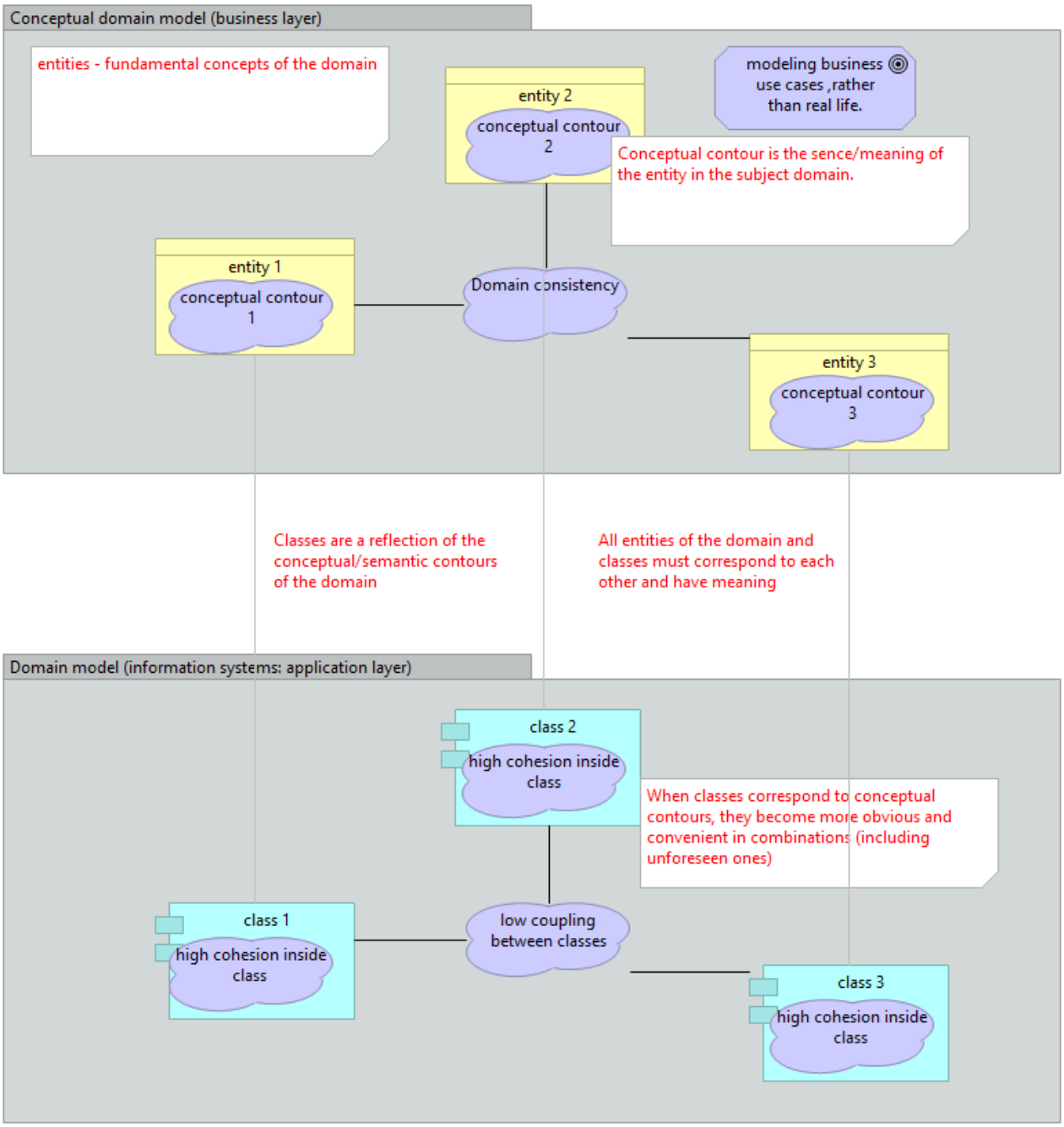
SIDE-EFFECT FREE FUNCTIONS



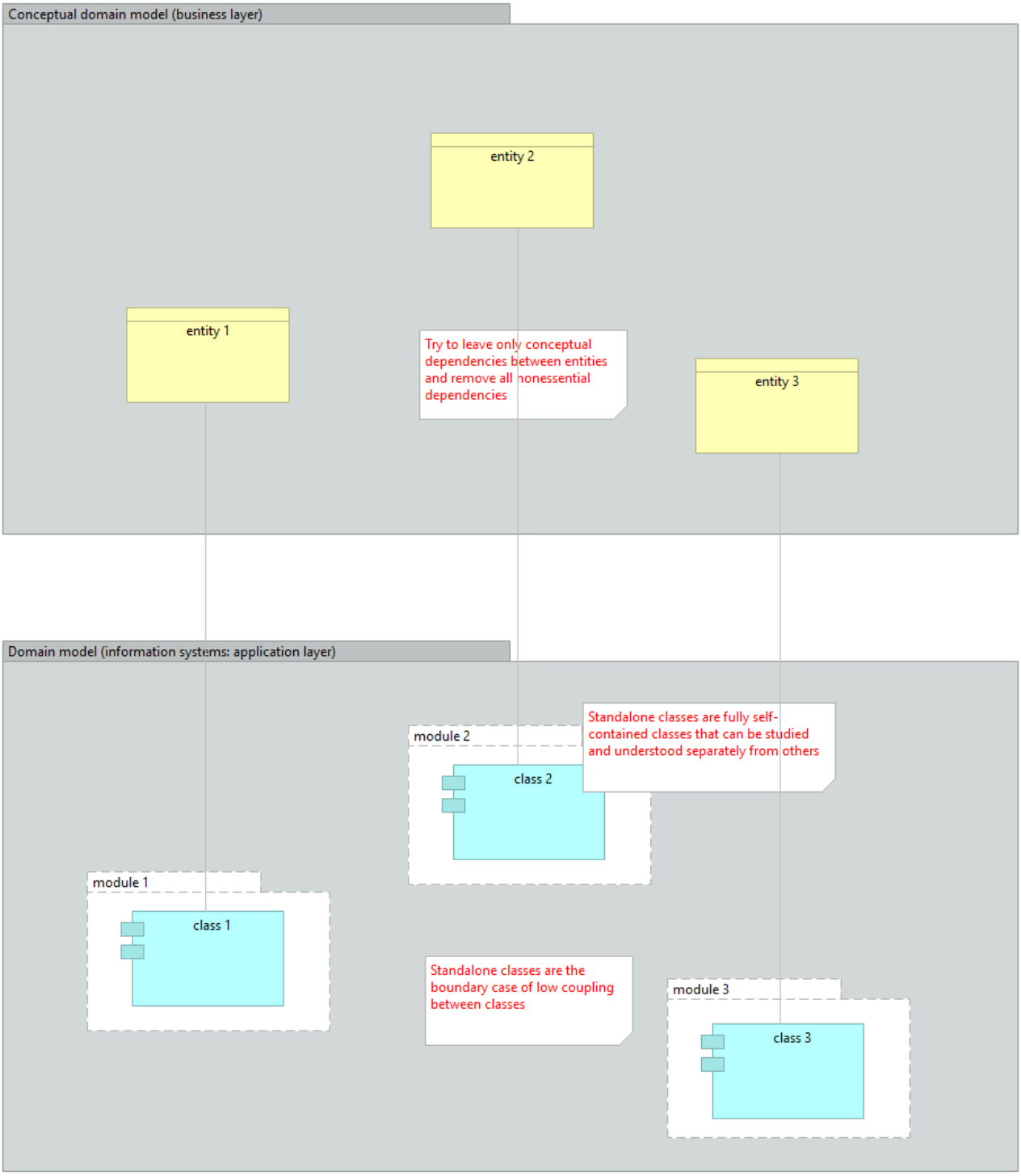
ASSERTIONS


assertion defines the result of an operation or an invariant
Application Component
command with side effects
Conditions that must be met, so that the post-condition gives a guaranteed result
test command
assert pre-conditions
Explicit record of assertions, helps the developer to understand the work of the command and side effects
call command
The side effects of the operation are the guaranteed consequences of calling the method
assert post-conditions

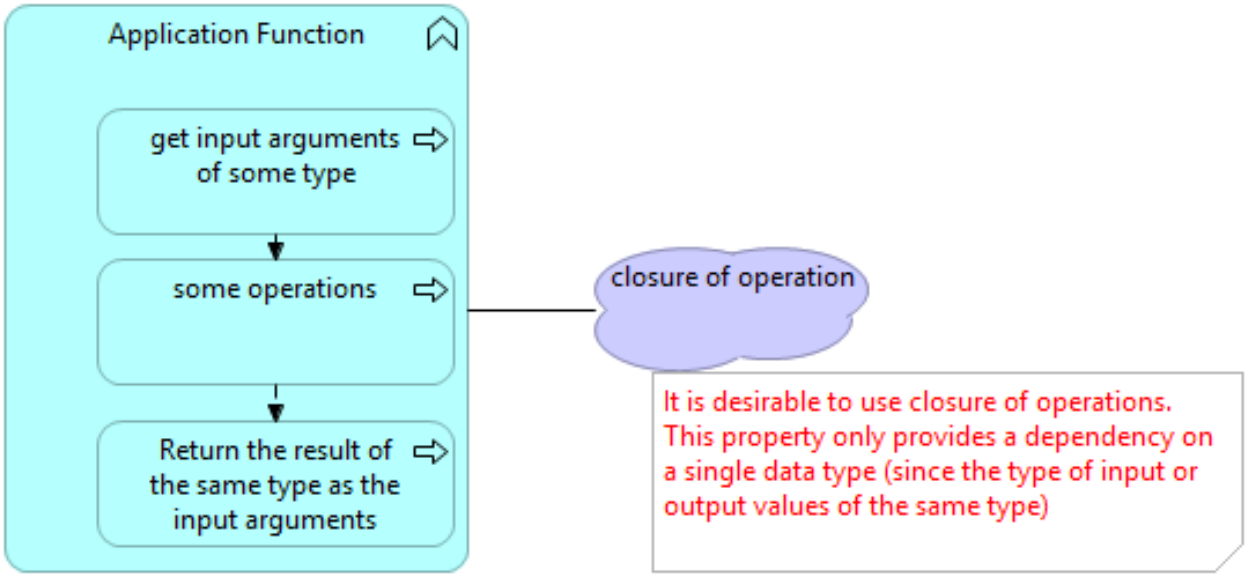
CONCEPTUAL CONTOURS



STANDALONE CLASSES



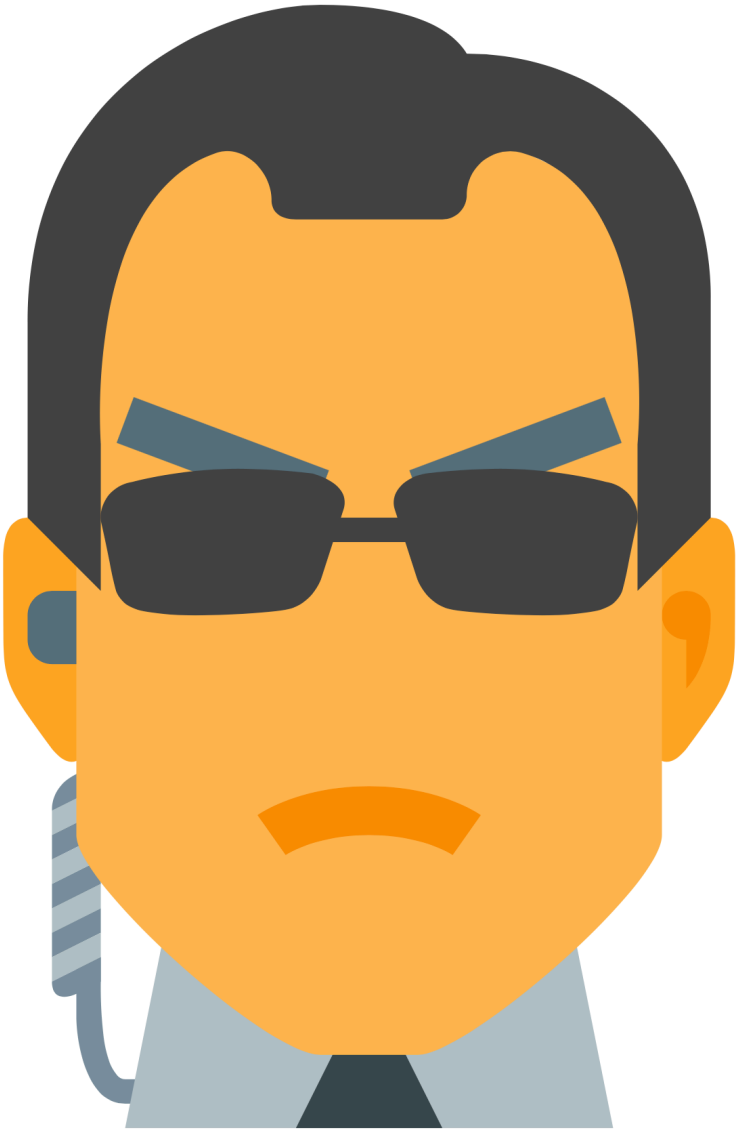
CLOSURE OF OPERATIONS



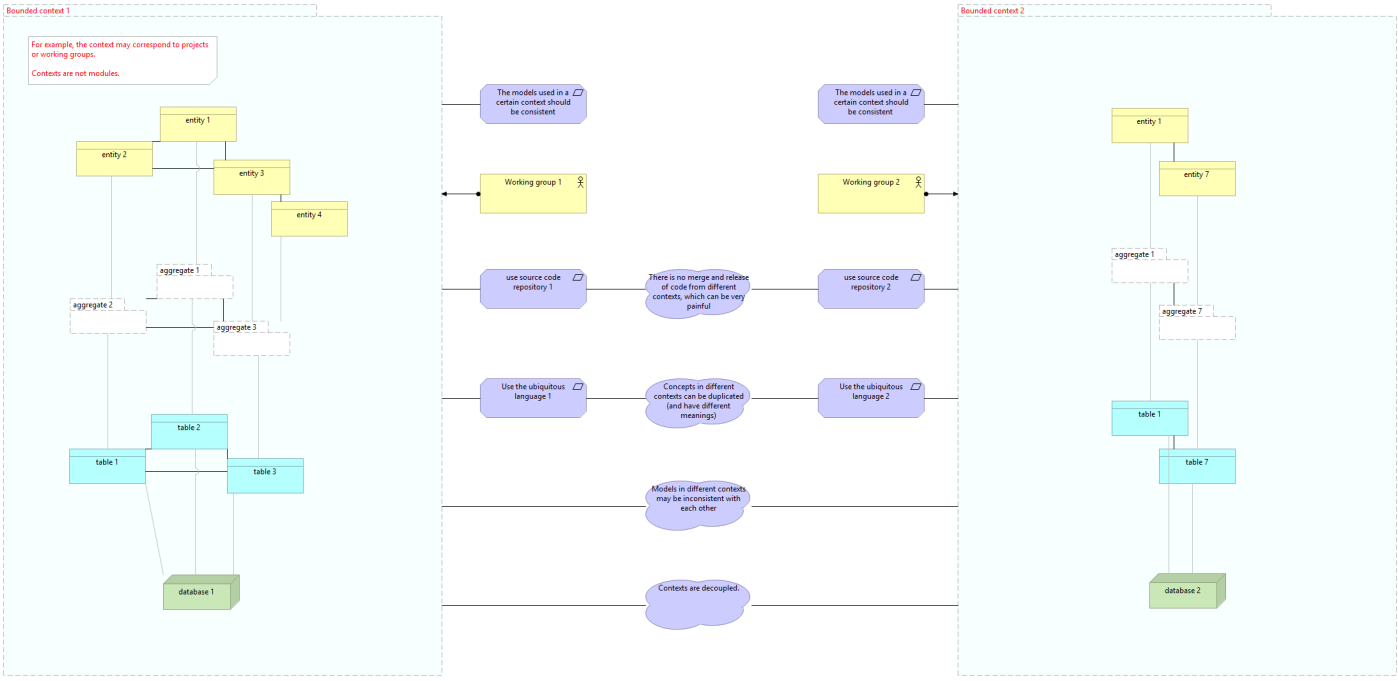
MODEL INTEGRITY AND CONTEXT

DOMAIN DRIVEN DESIGN

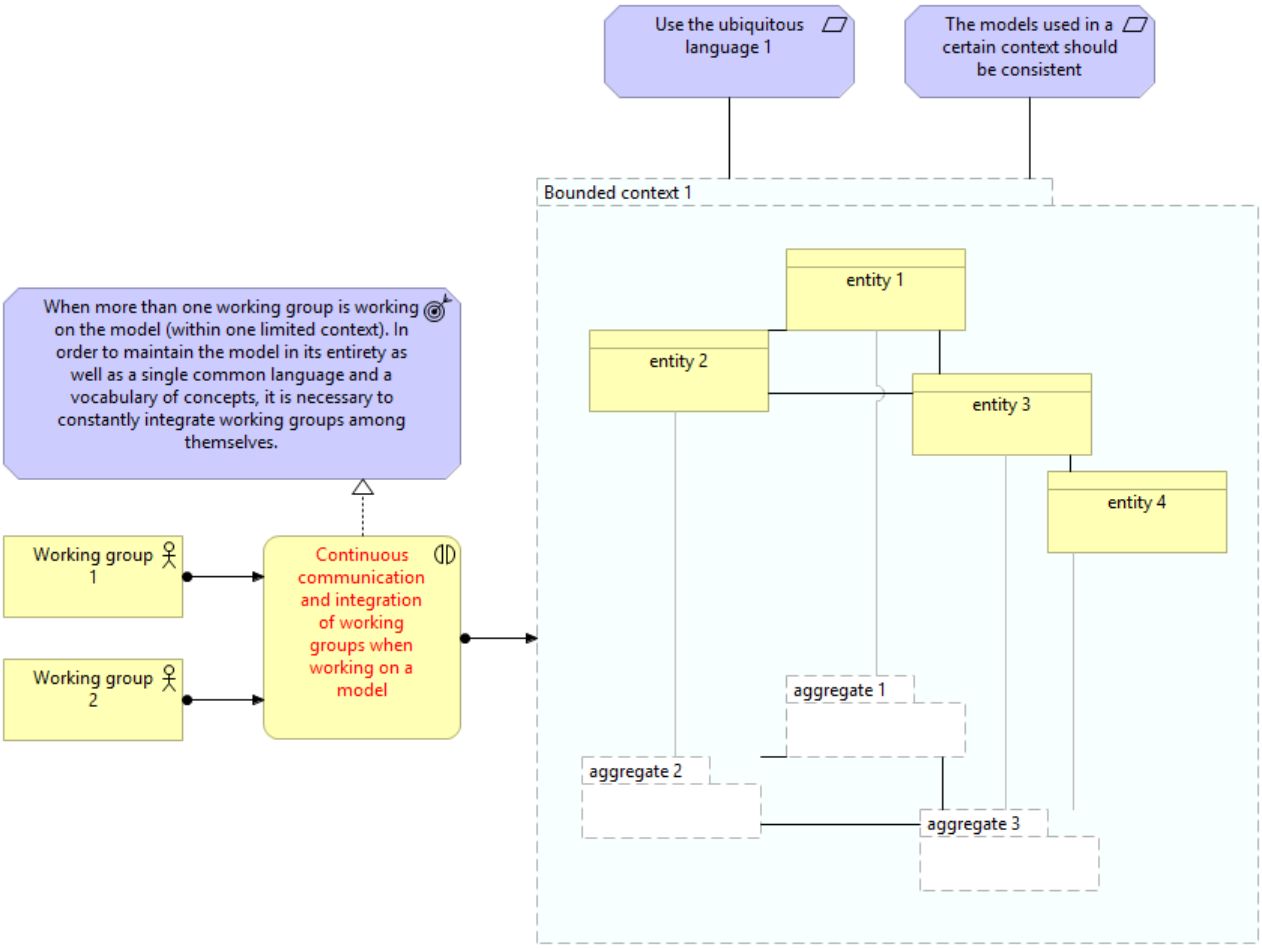
Eric Evans



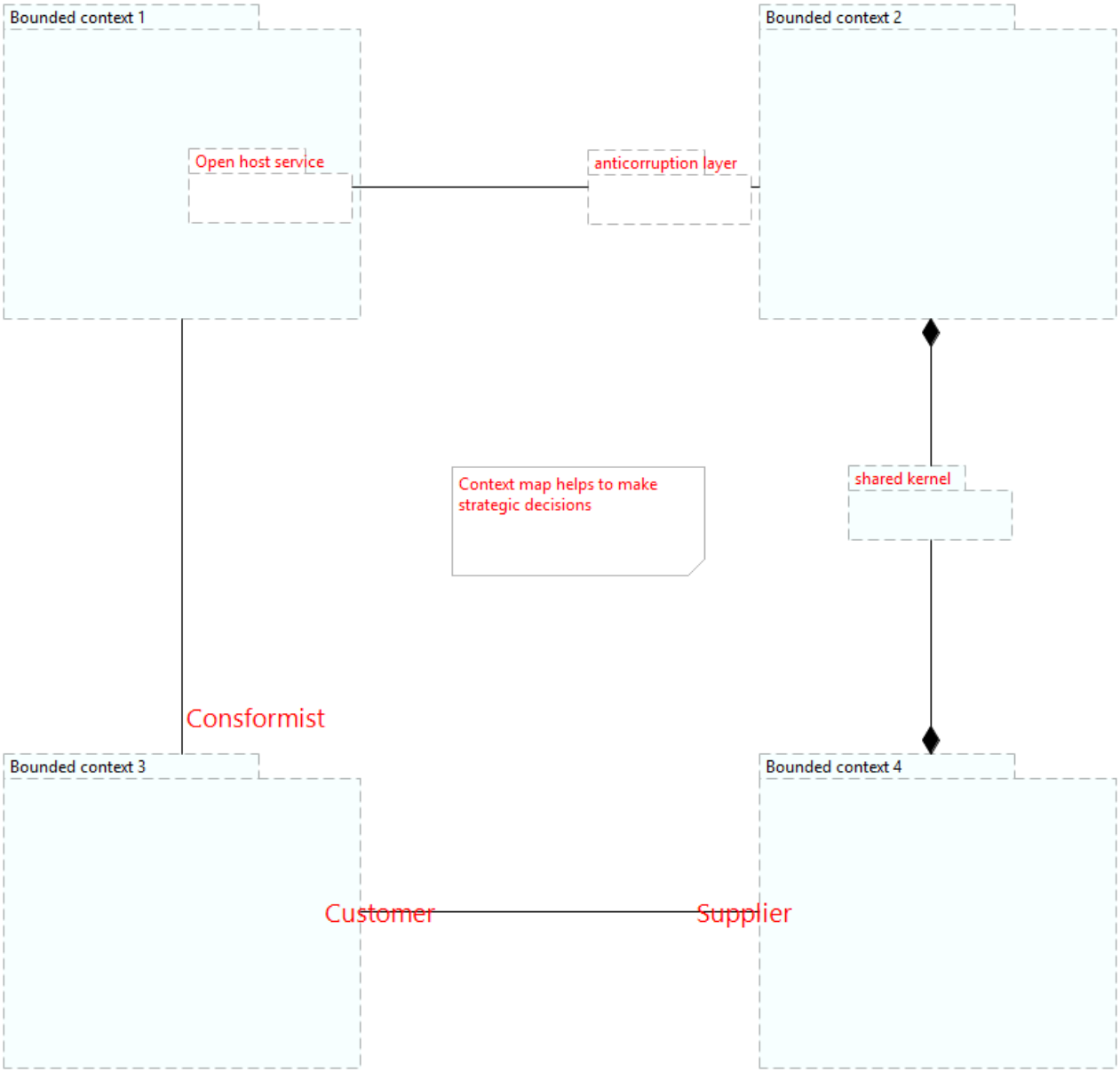
BOUNDED CONTEXT



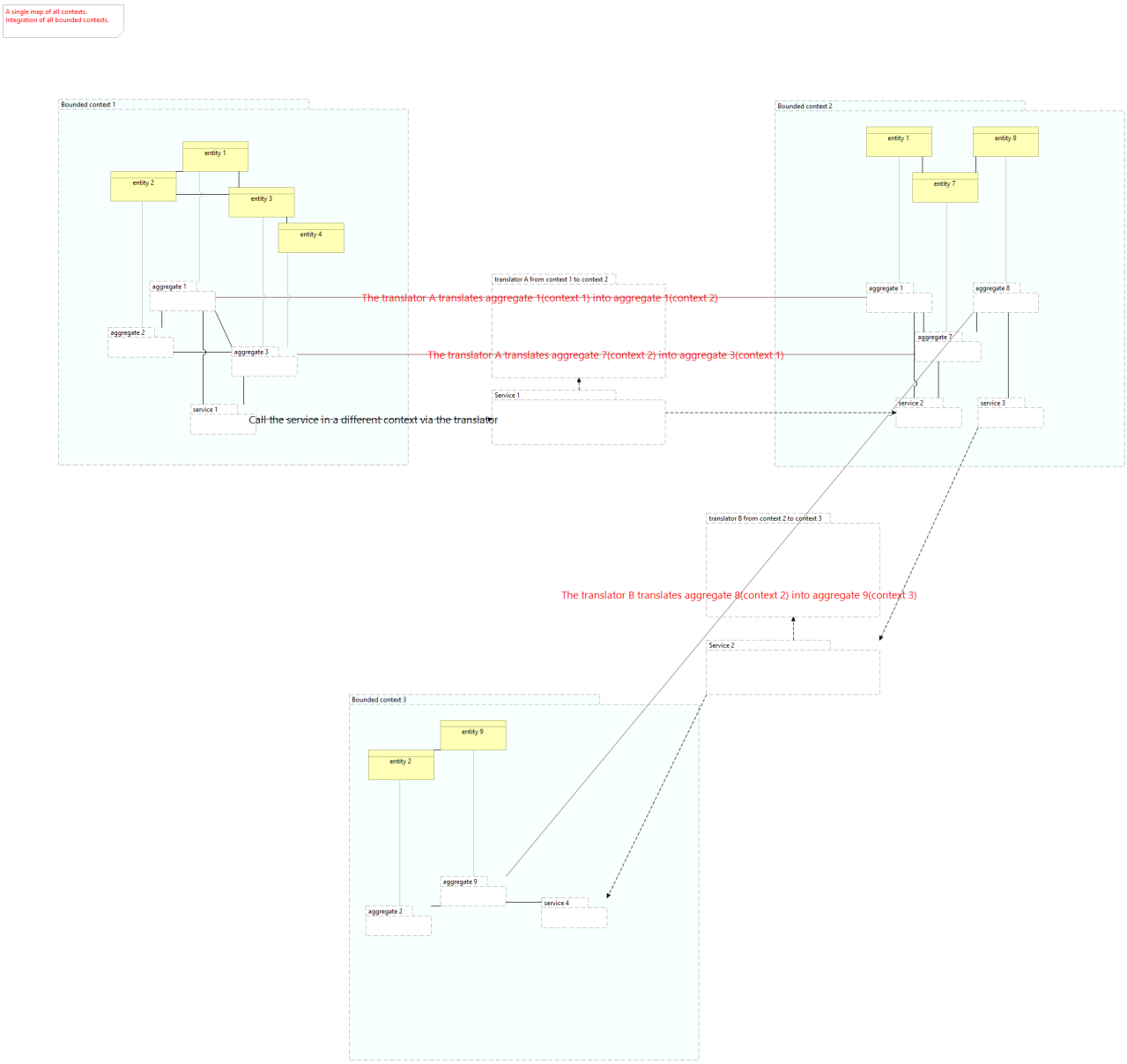
CONTINUOUS INTEGRATION



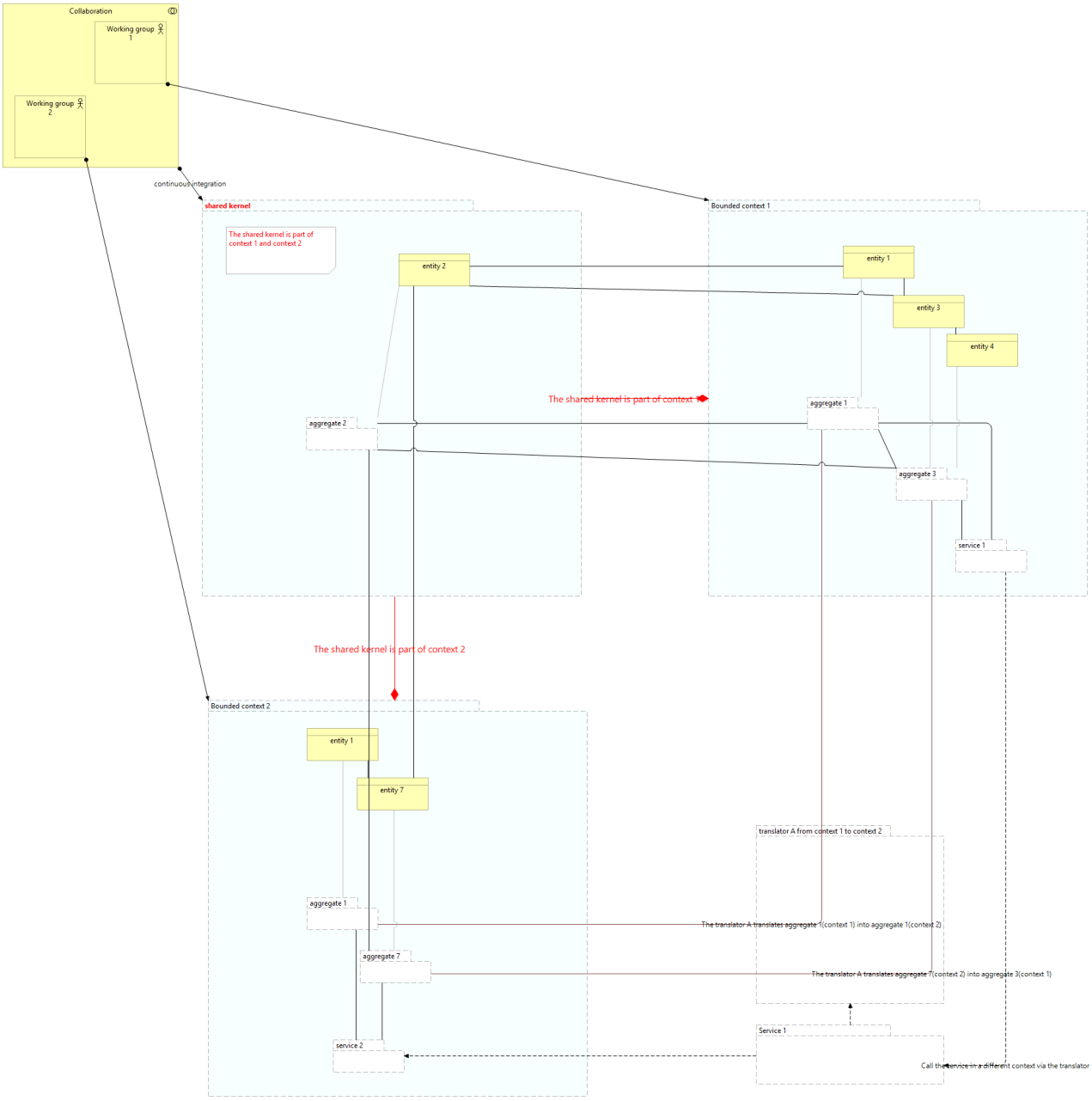
STRATEGIC CONTEXT MAP



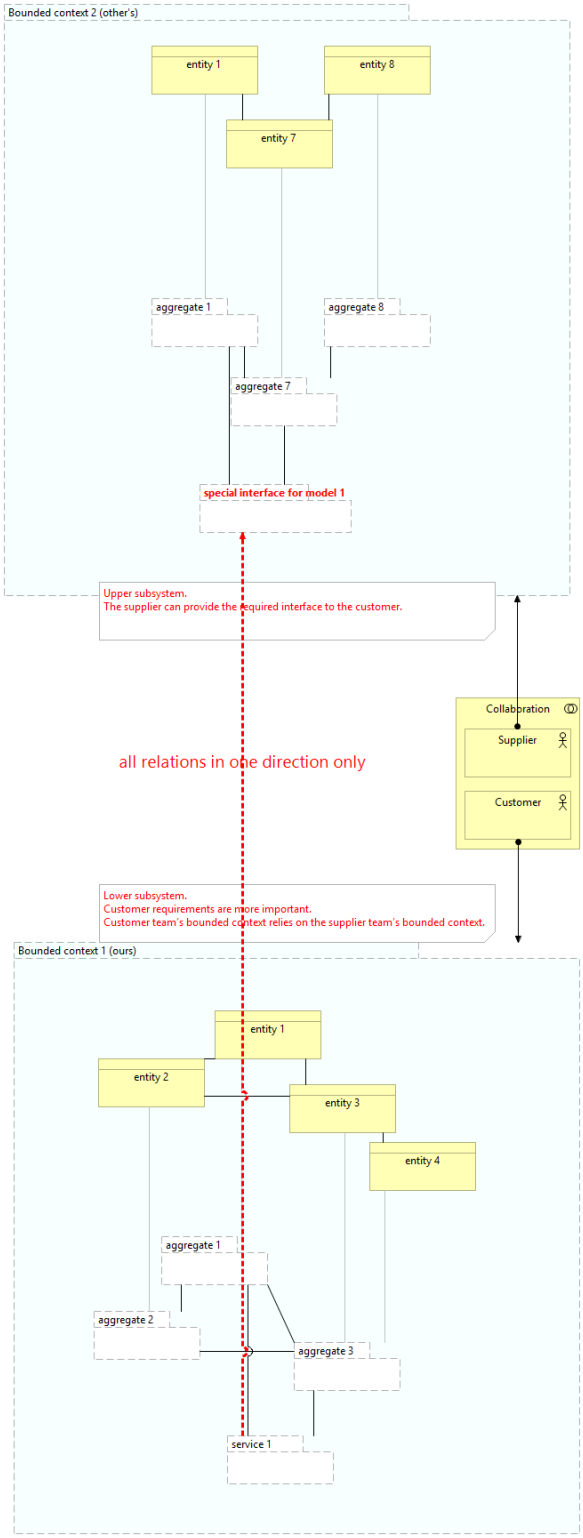
CONTEXTUAL MAP



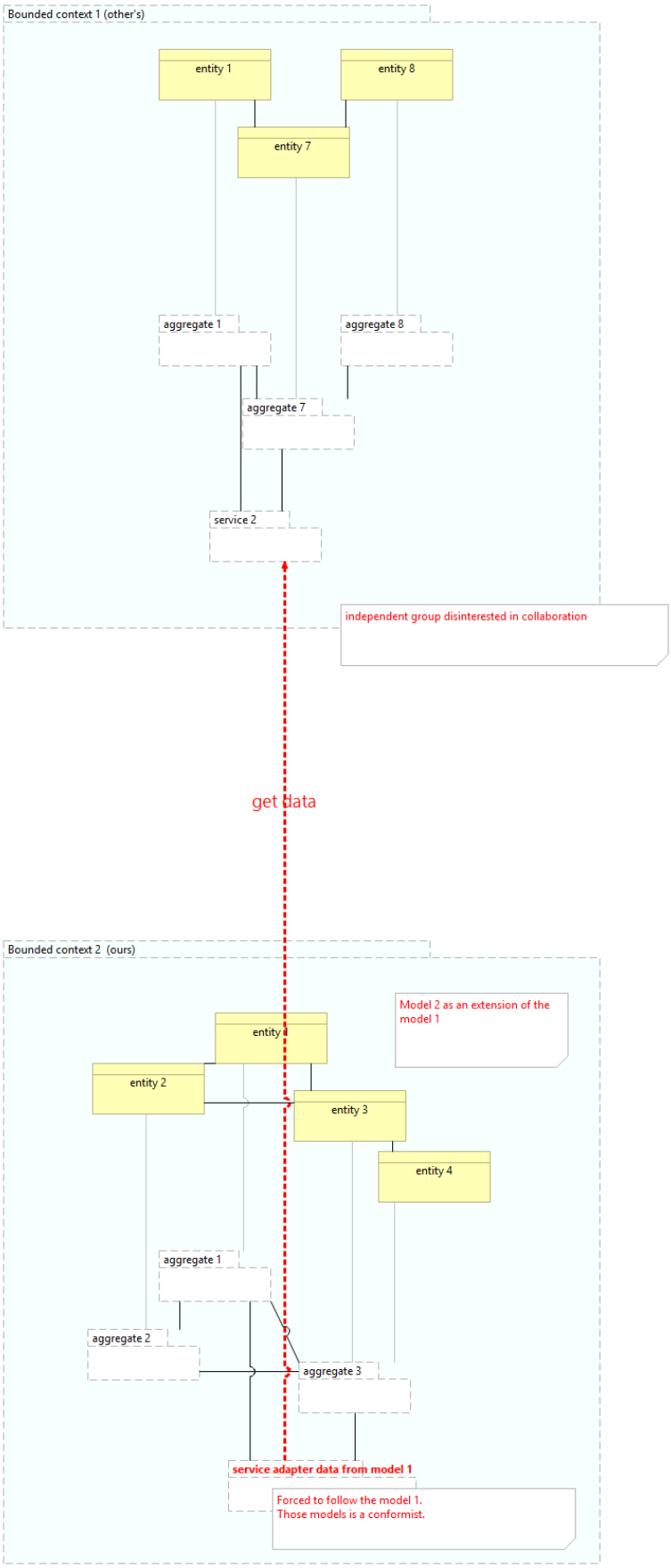
SHARED KERNEL



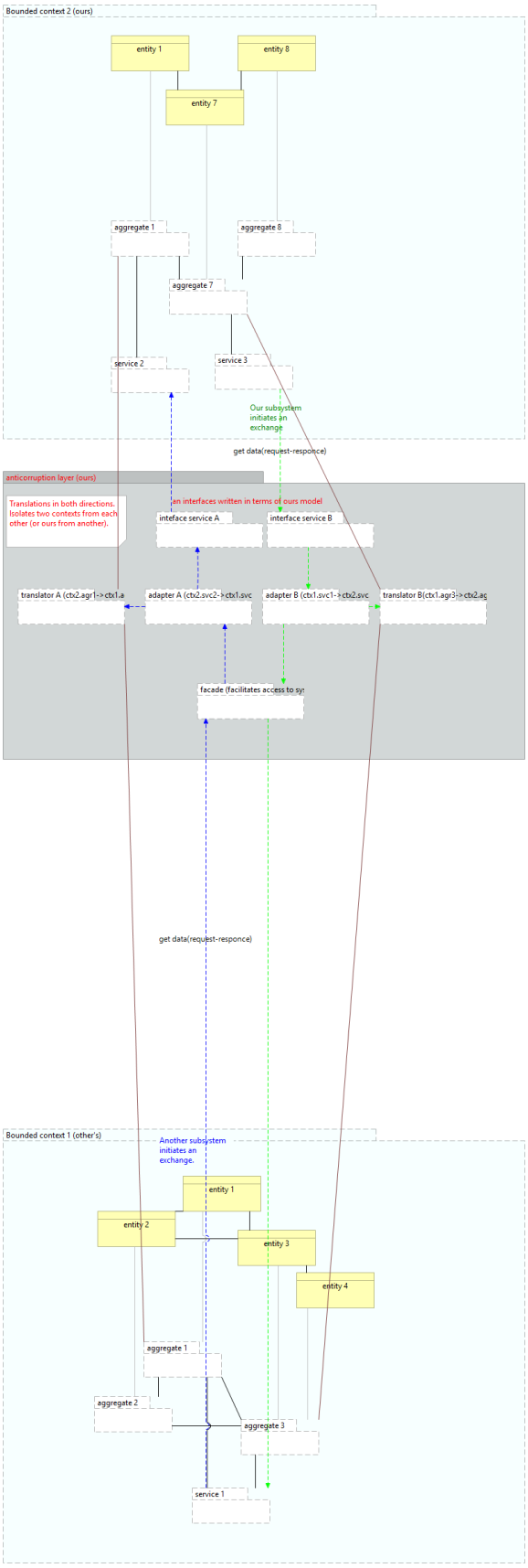
CUSTOMER-SUPPLIER TEAMS



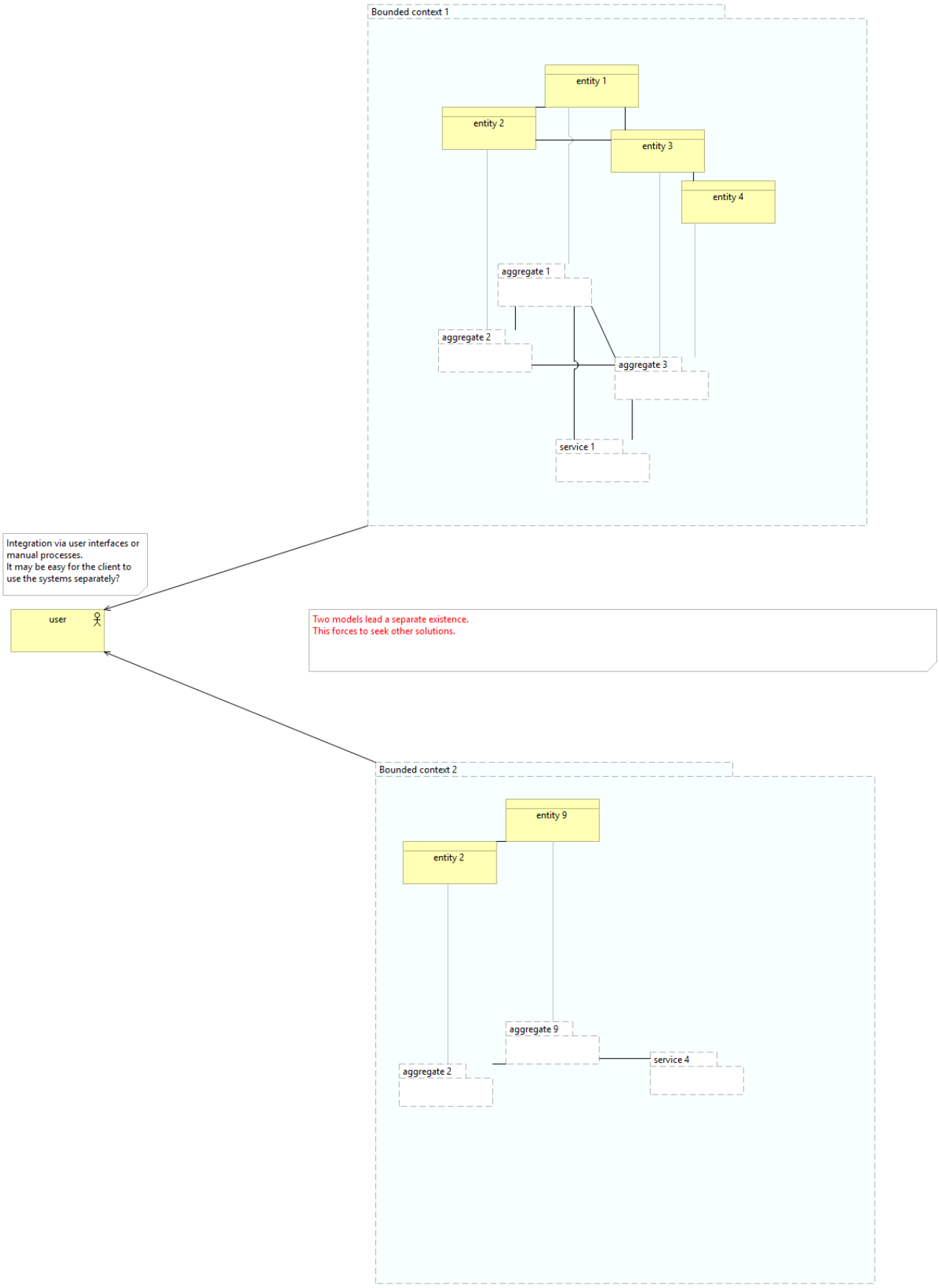
CONFORMIST


Bounded context 1 (other's)
entity 1
entity 8
entity 7
aggregate 1
aggregate 8
aggregate 7
service 2
independent group disinterested in collaboration
get data
Bounded context 2 (ours)
Model 2 as an extension of the model 1
entity 1
entity 2
entity 3
entity 4
aggregate 1
aggregate 2
aggregate 3
service adapter data from model 1
Forced to follow the model 1.
Those models is a conformist.

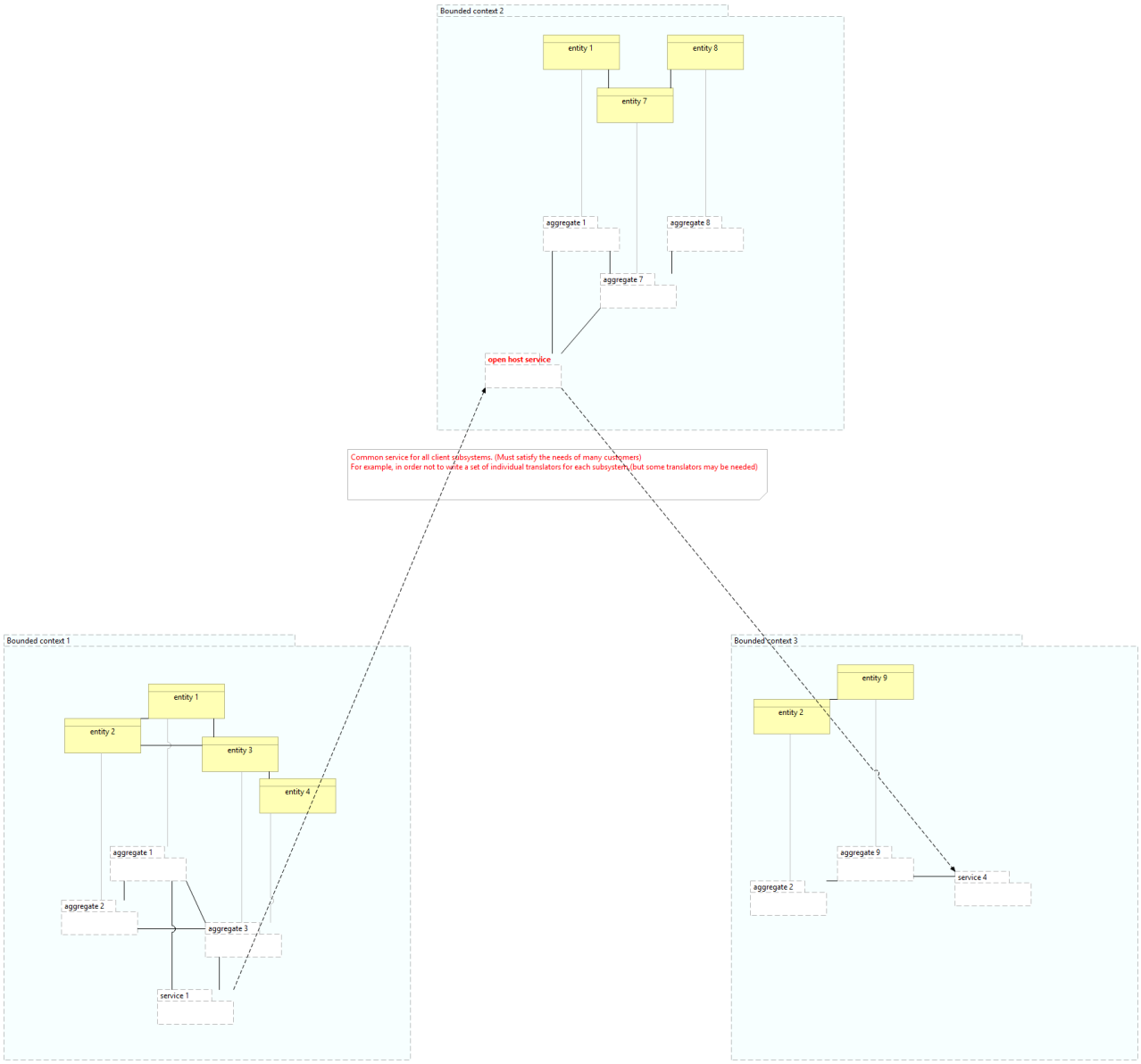
ANTICORRUPTION LAYER



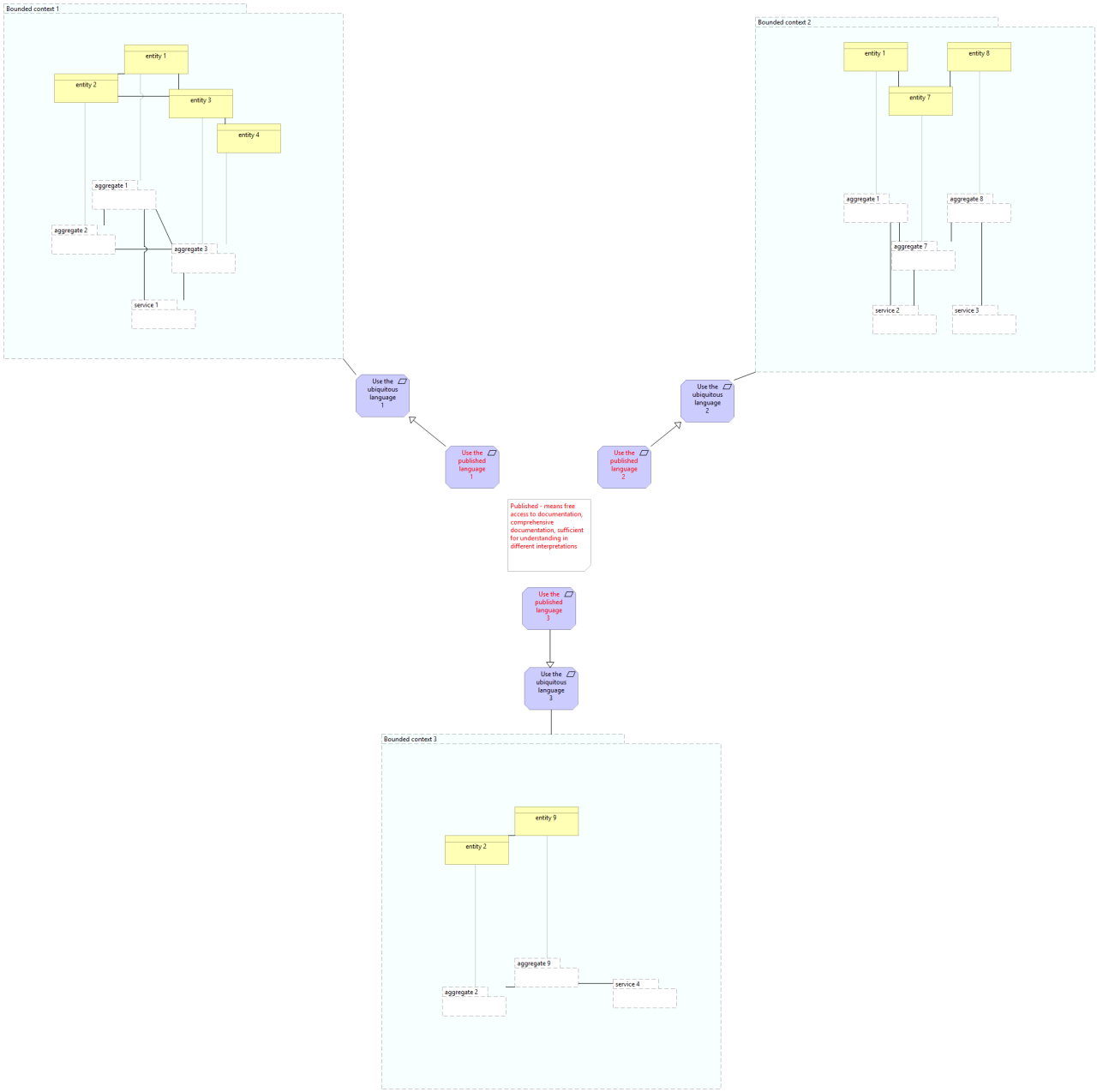
SEPARATE WAYS



OPEN HOST SERVICE



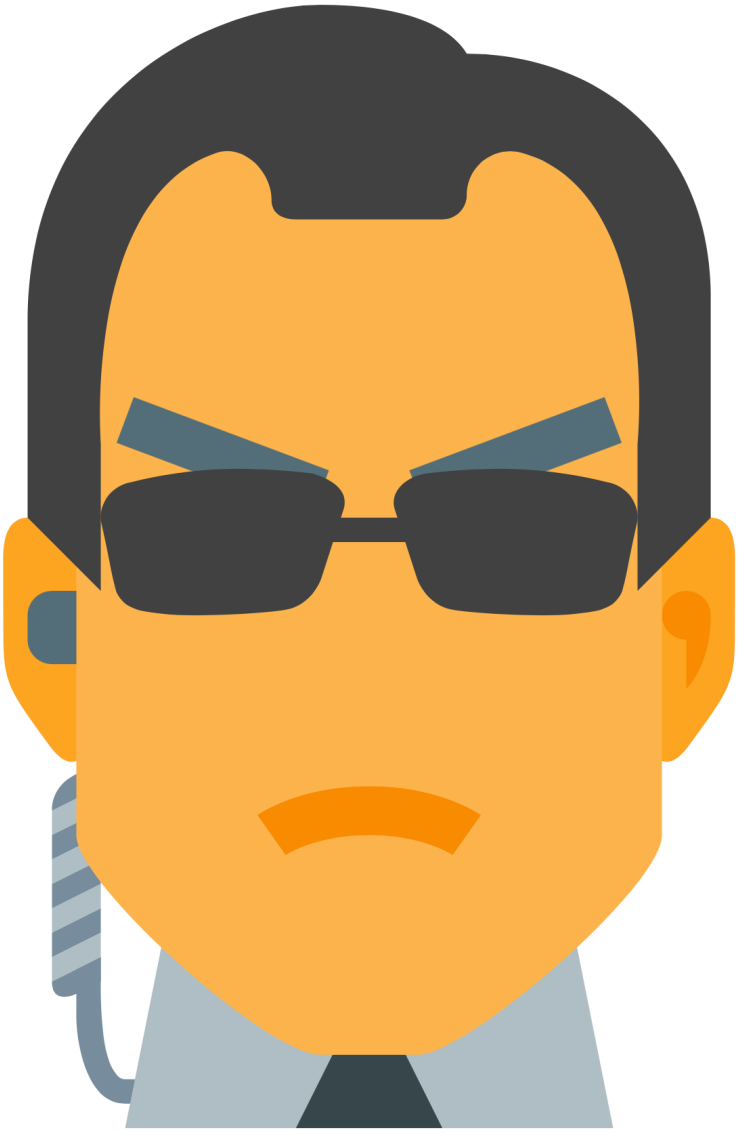
PUBLISHED LANGUAGE



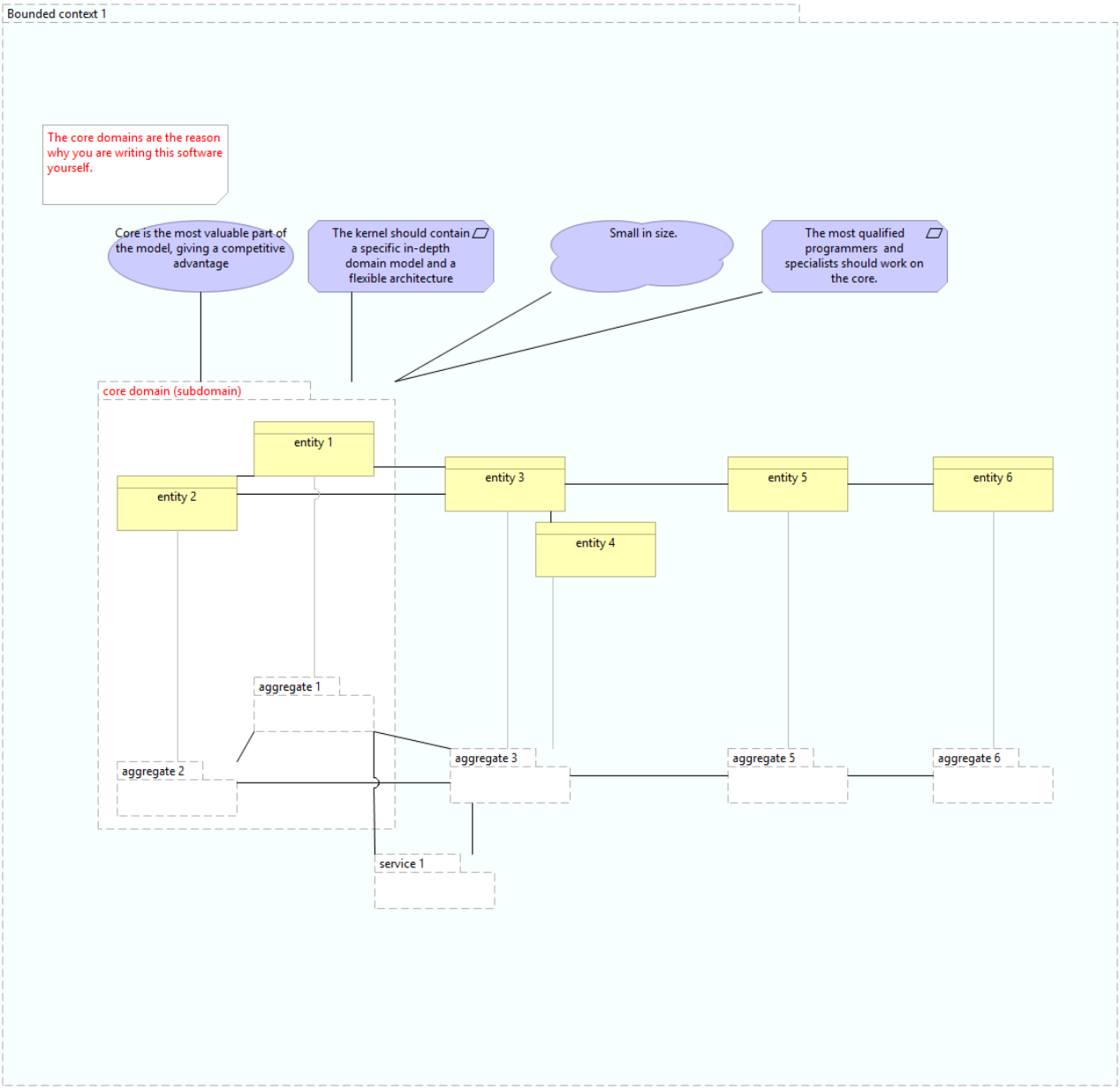
DISTILLATION

DOMAIN DRIVEN DESIGN

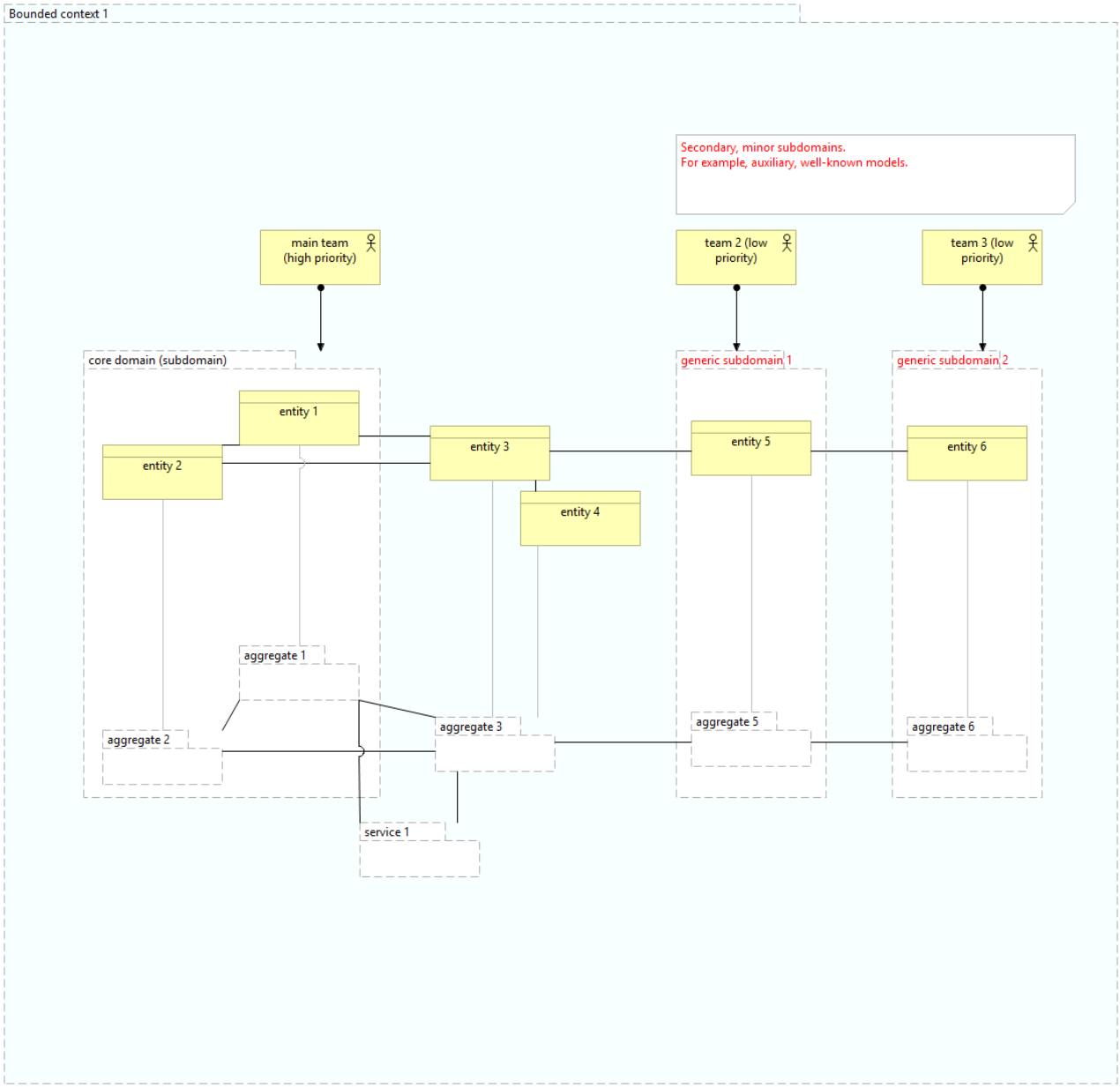
Eric Evans



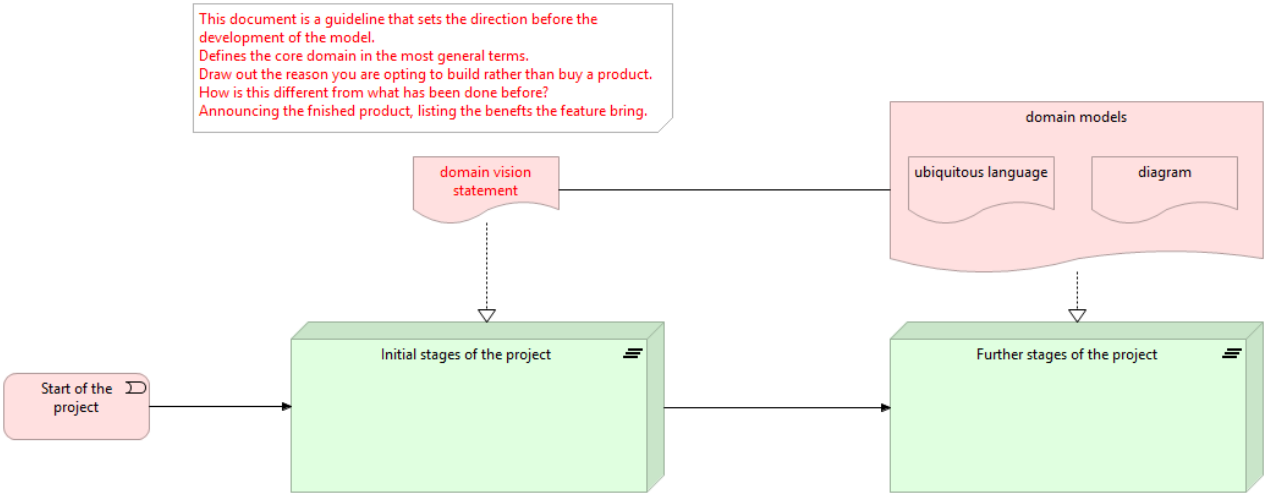
CORE DOMAIN


Bounded context 1
The core domains are the reason why you are writing this software yourself.
Core is the most valuable part of the model, giving a competitive advantage
The kernel should contain a specific in-depth domain model and a flexible architecture
Small in size.
The most qualified programmers and specialists should work on the core.
core domain (subdomain)
entity 1
entity 2
entity 3
entity 4
entity 5
entity 6
aggregate 1
aggregate 2
aggregate 3
aggregate 5
aggregate 6
service 1

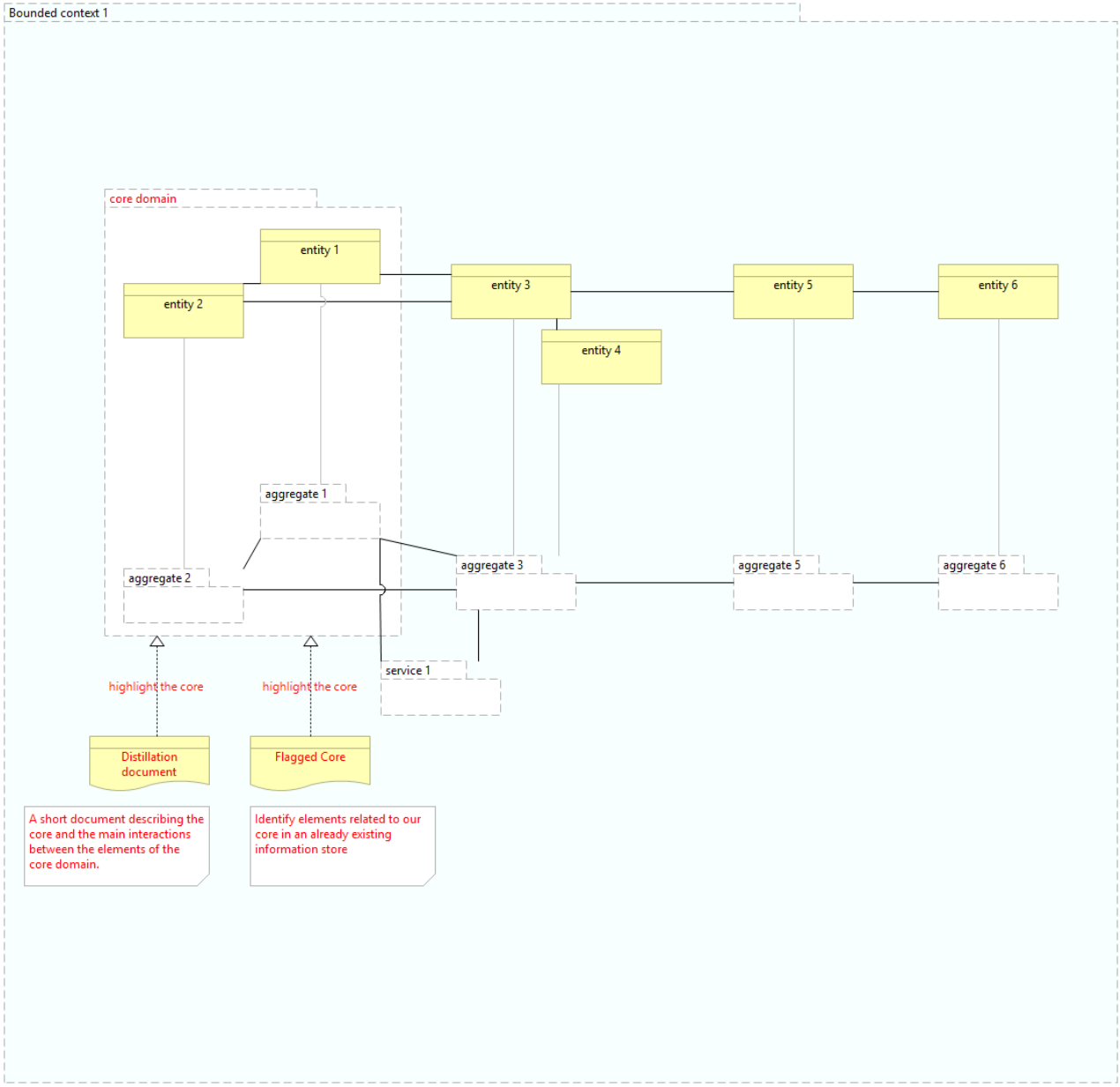
GENERIC SUBDOMAINS



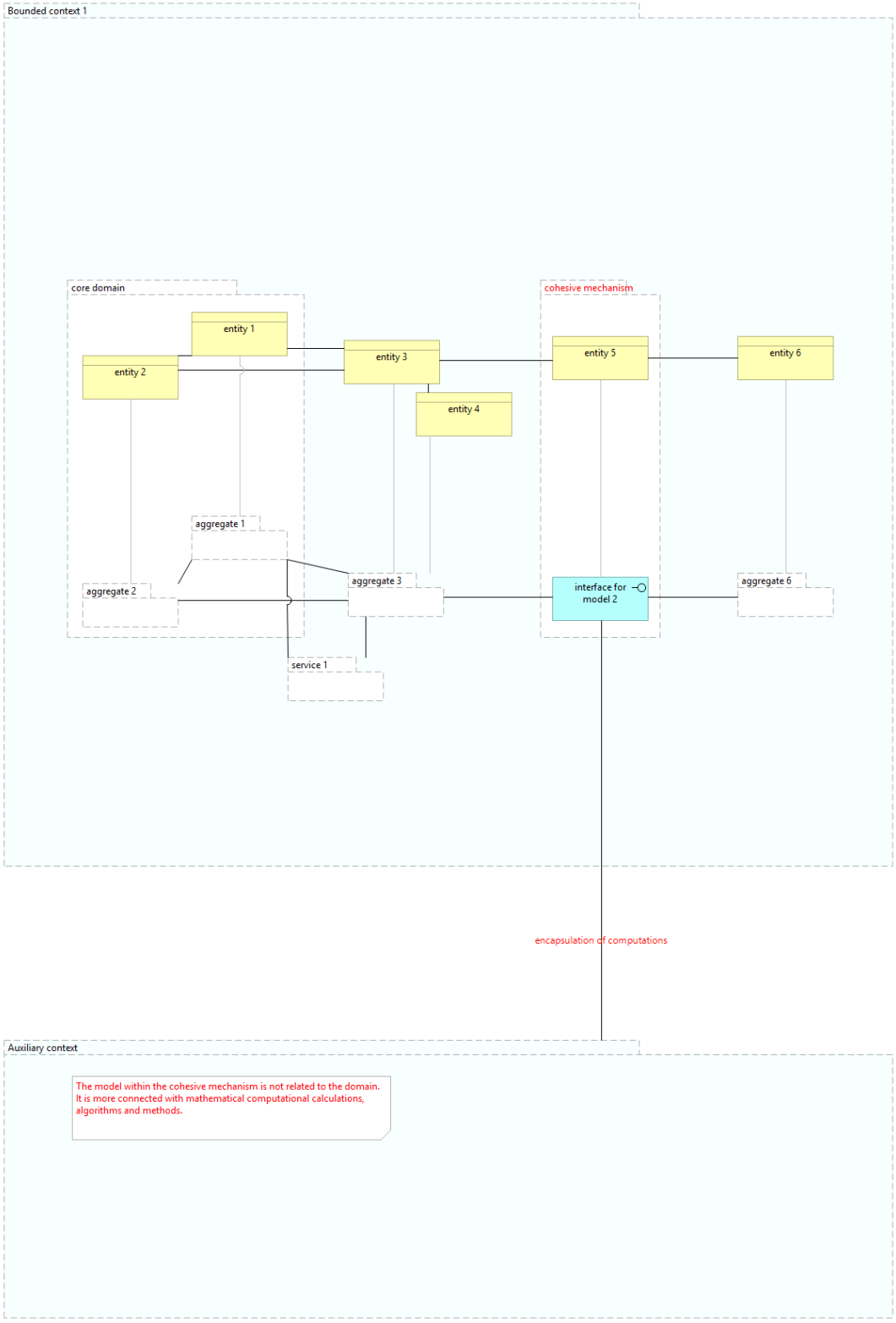
DOMAIN VISION STATEMENT



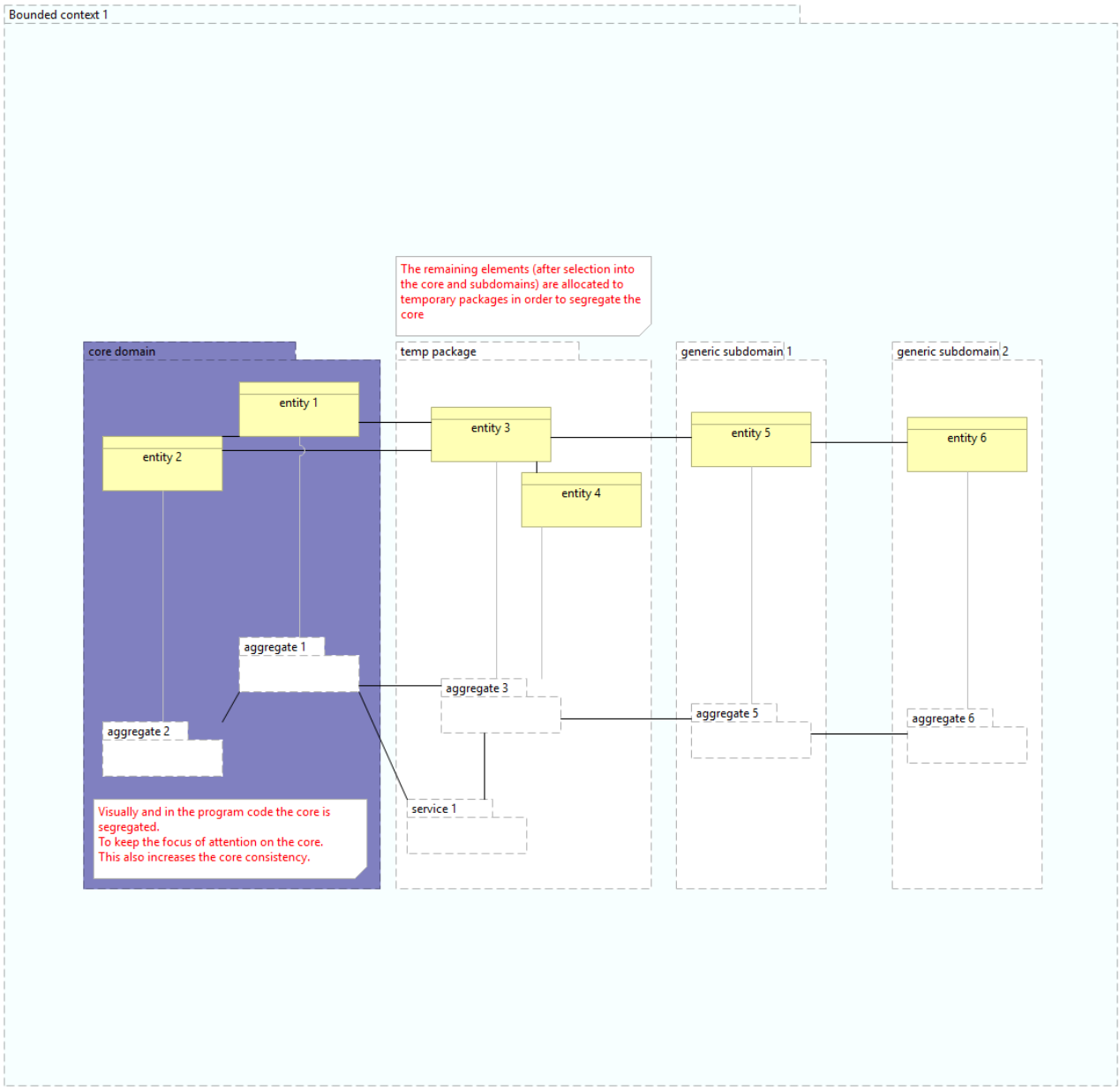
HIGHLIGHTED CORE



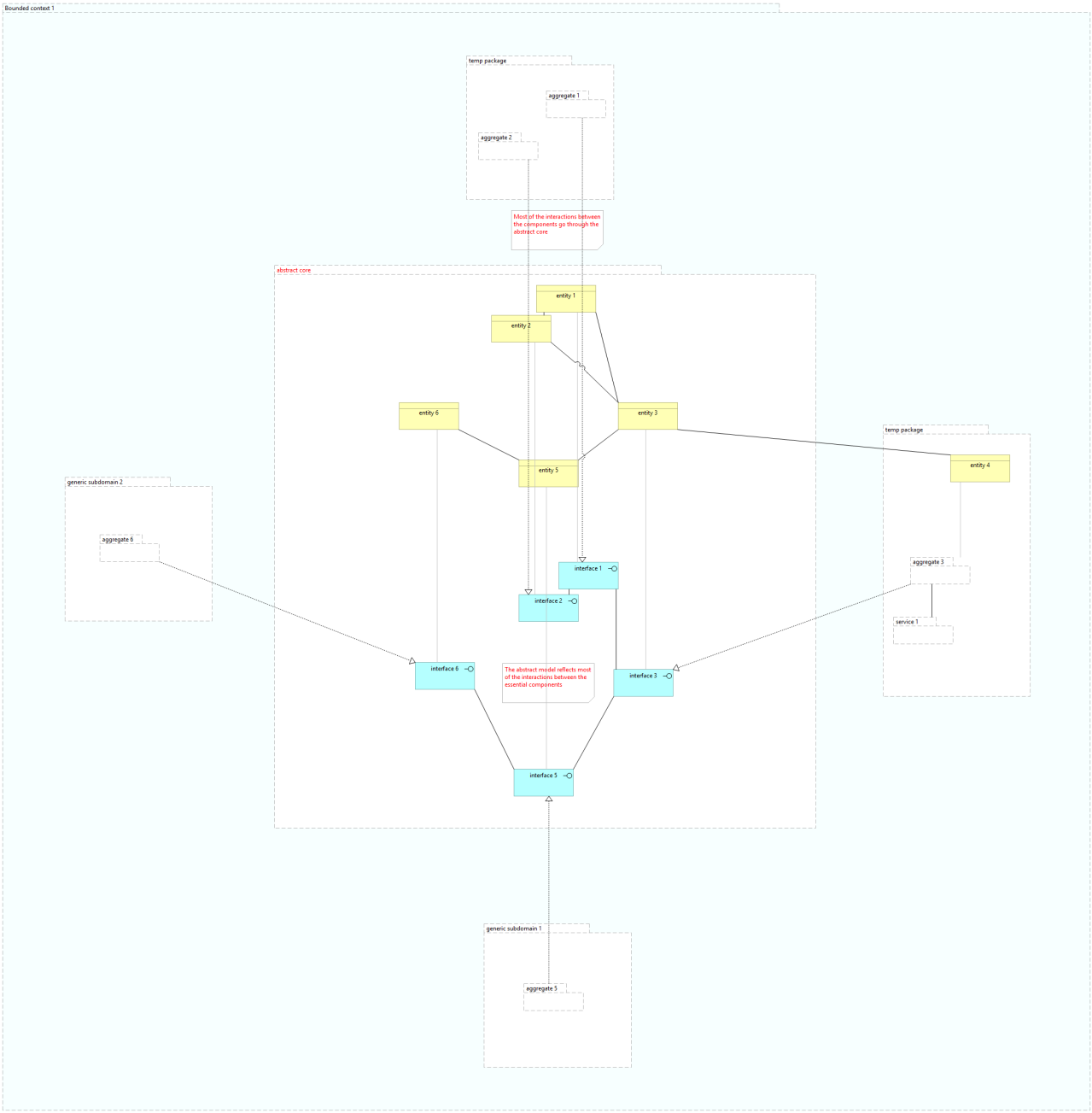
COHESIVE MECHANISMS



SEGREGATED CORE



ABSTRACT CORE

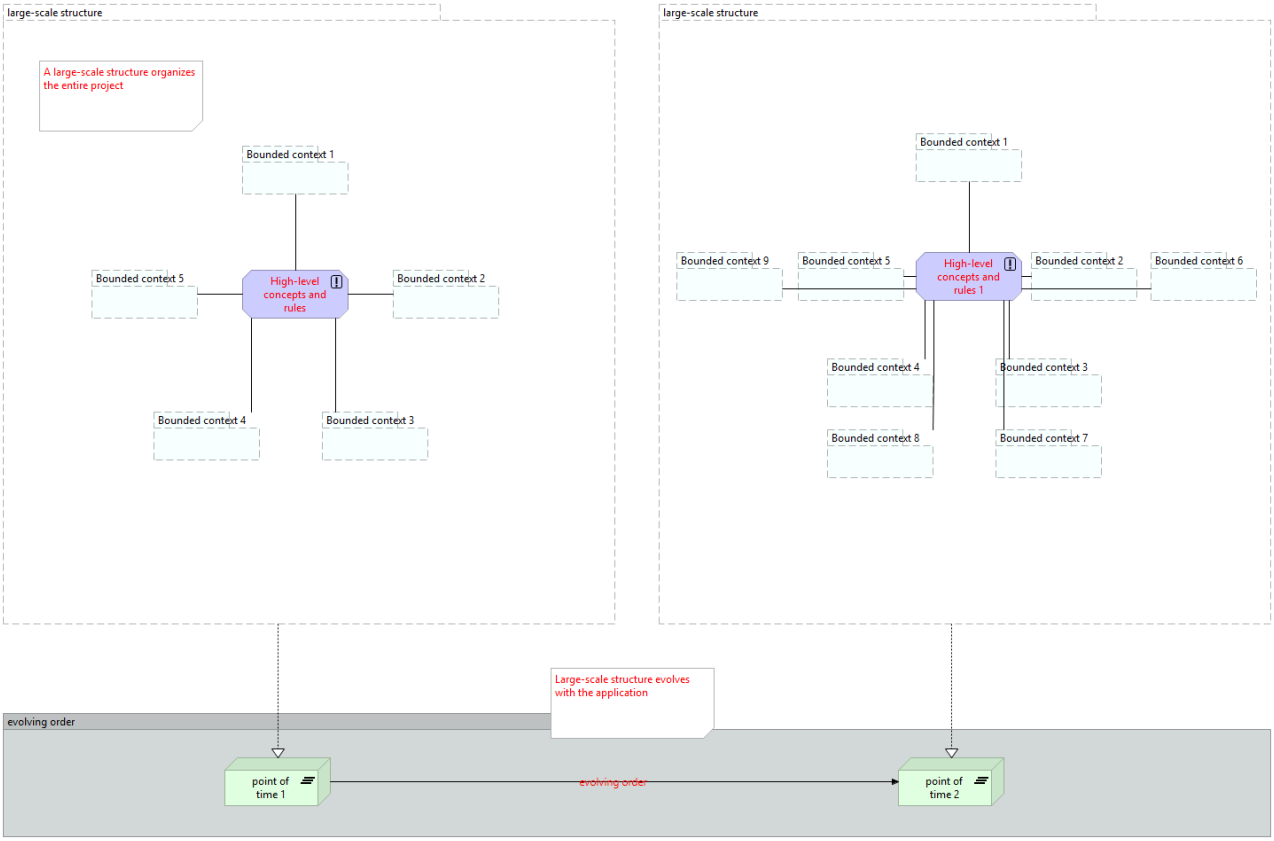


LARGE-SCALE STRUCTURE

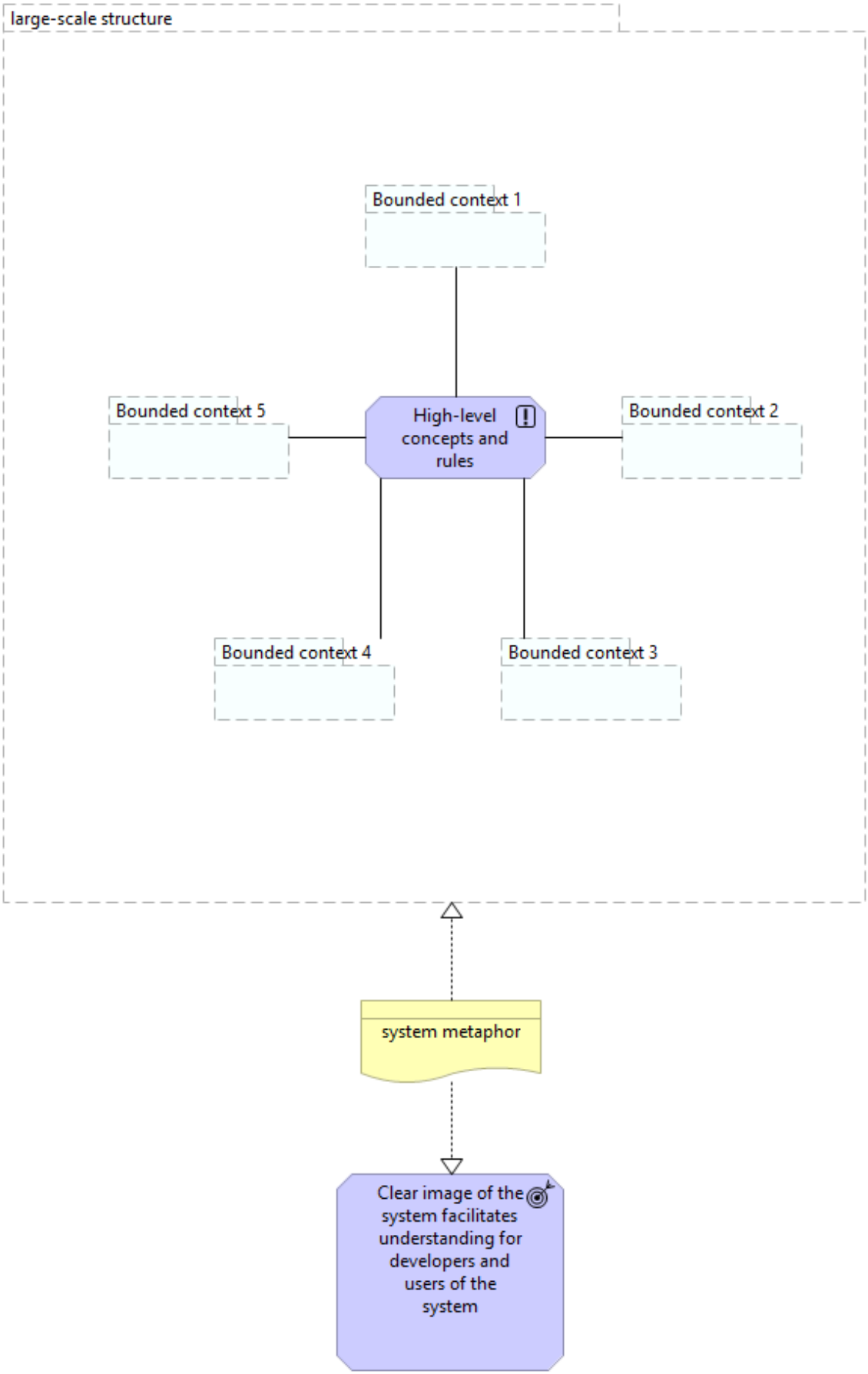
DOMAIN DRIVEN DESIGN

Eric Evans

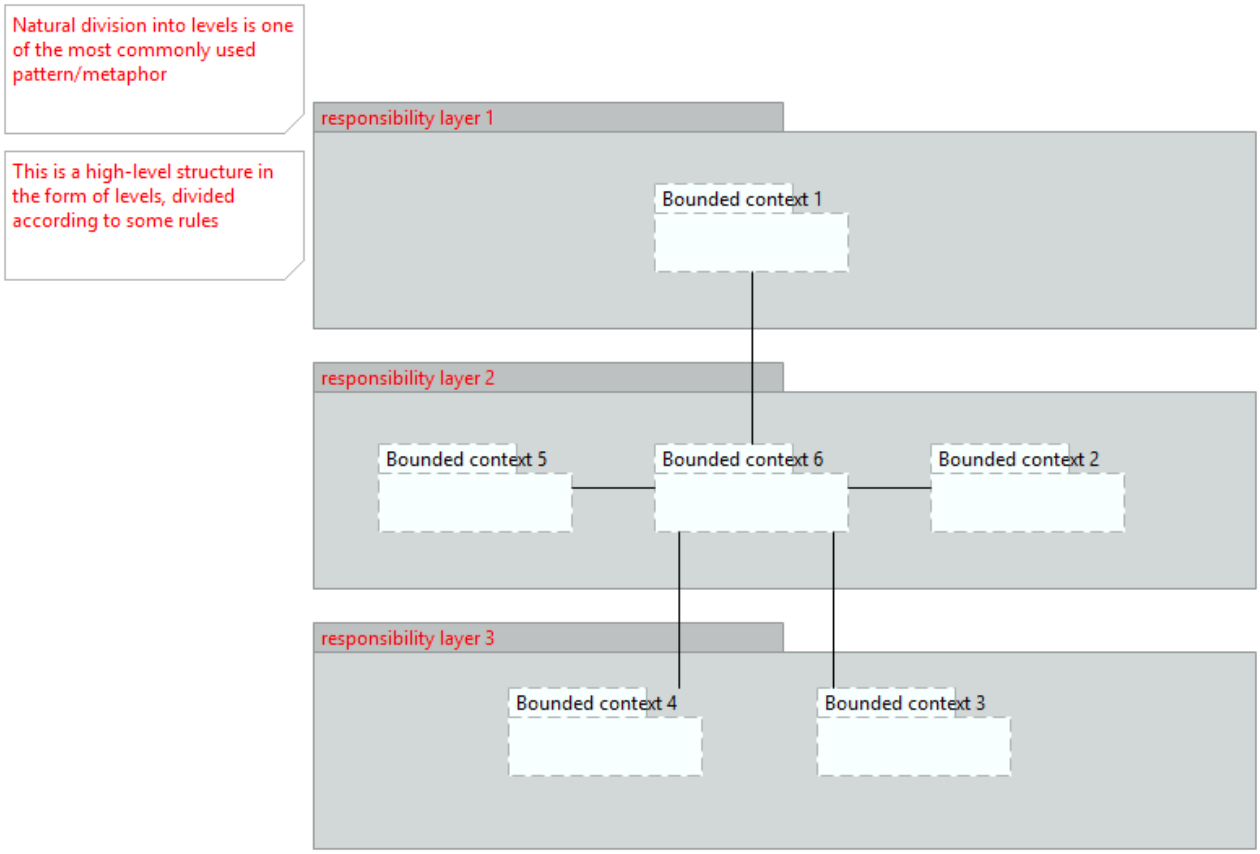
EVOLVING ORDER



SYSTEM METAPHOR



RESPONSIBILITY LAYERS



Layers based on NGOSS Framework Telecom Application Map

Market / Sales Domain

- Campaign Mangemer
- Sales Aids
- Compensation & Resu
- Mass Market Sales Ma
- Channel Sales Manag
- Corporate Sales Mana
- Sales Portal

Product Management Domain

- Product Strategy / Prc
- Product / Service Cata
- Product Lifecycle Mar
- Product Performance

Operations Support & Readiness	Fulfillment	Assurance	Billing

Customer Management Domain

- Customer Informatior
- Customer Order Mana
- Customer QoS & SLA
- Customer Service/Acc
- Receivables Managem
- Billing Account Mana
- Transactional Docume
- Customer Self Management
- Billing inquiry, dispute
- Collection Manageme
- Customer Contact Management, Retention & Loyability
- Bill Format/Render
- Bill Calculation
- Customer Service Representative Toolbox
- Product/Service Ratin
- Online Charging Appl

Service Management Domain

- Service Specification M
- Service Order Manage
- Service Level Agreeme
- Service Quality Monit
- Service Inventory Mar
- Service Problem Mana
- Service Performance M

Resource Management Domain

- Resource Lifecycle Ma
- Resource Order Mana
- Resource Assurance M
- Receivables Managem
- Resource Process Management (Workflow/Integration)
- Voucher Management
- Real time Billing Medi
- Resource Inventory Management Application
- Resource Domain Management Applications

Supplier / Partner Domain

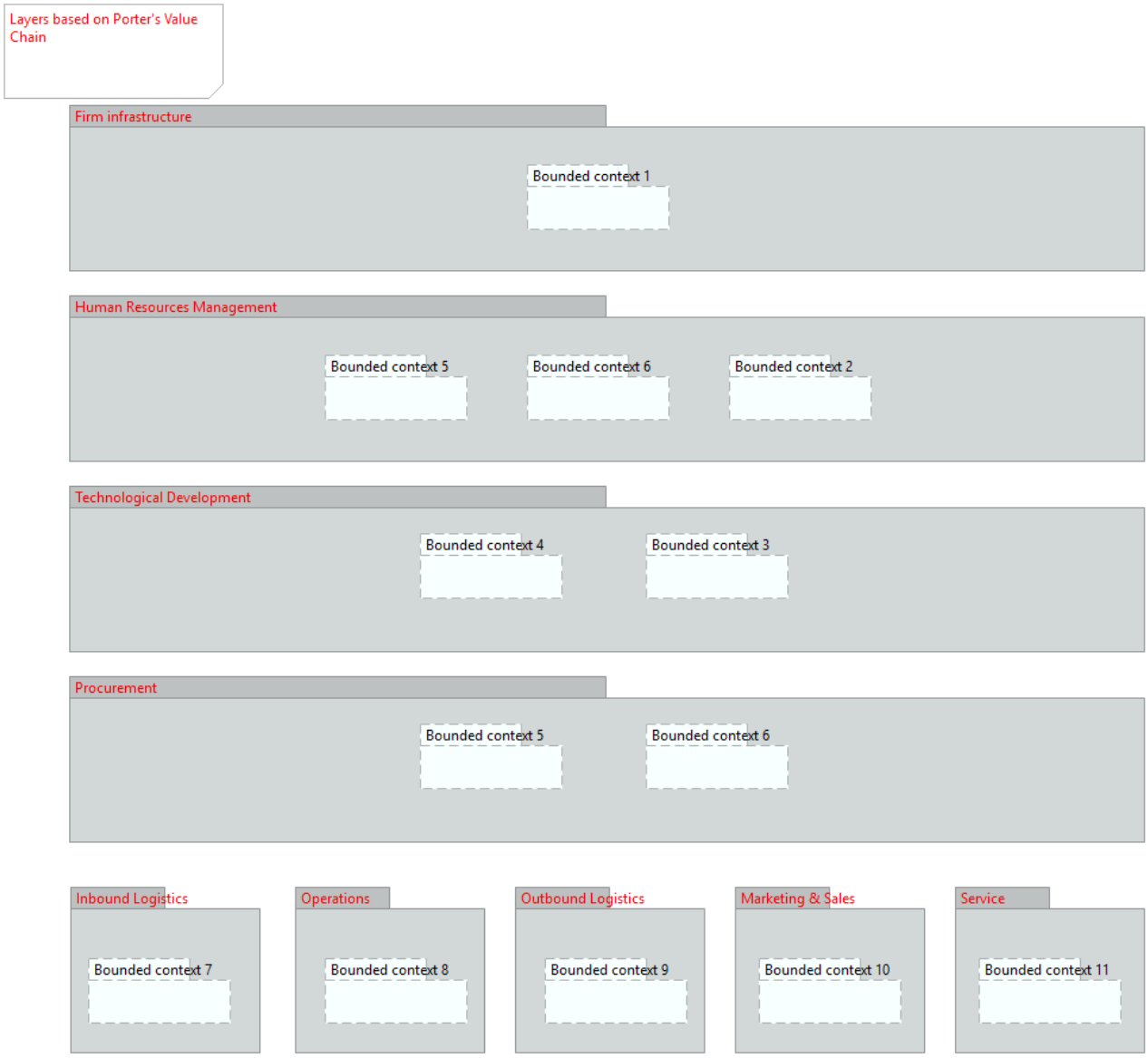
- Partner Management
- Suuply Chain Manage

Enterprise Domain

- Revenue Assurance M
- HR Management
- Financial Managemer
- Asset Management
- Security Management
- Knowlege Manageme
- Fraud Management

Integration Infrastructure

- bus technology / mid


Layers based on Porter's Value Chain
Firm infrastructure
Bounded context 1
Human Resources Management
Bounded context 5
Bounded context 6
Bounded context 2
Technological Development
Bounded context 4
Bounded context 3
Procurement
Bounded context 5
Bounded context 6
Inbound Logistics
Bounded context 7
Operations
Bounded context 8
Outbound Logistics
Bounded context 9
Marketing & Sales
Bounded context 10
Service
Bounded context 11

The Banking Industry Architecture Network BIAN.org

Legend

REST API exists	Reserved	Gap exists

REFERENCE DATA

Party

- Party Data Management
- Customer Profile

External Agency

- Information Provider Administration
- Syndicate Management
- Interbank Relationship Management
- Correspondent Bank Relationship Management
- Correspondent Bank Administration
- Sub Custodian Agreement
- Product Service Agency
- Product Broker Operations
- Contractor/Supplier Agreement

Market Data

- Information Provider Operation
- Market Information Management
- Financial Market Analysis
- Financial Market Research
- Quant Model
- Market Data Switch Administration
- Market Data Switch Operation
- Financial Instrument Reference Data Management
- Counterparty Administration
- Public Reference Data Management
- Position Data Management

Product Management

- Product Design
- Product Deployment
- Product Training
- Product Quality Assurance
- Discount Pricing
- Product Directory
- Special Pricing Conditions

SALES & SERVICE / ПРОДАЖИ И ОБСЛУЖИВАНИЕ КЛИЦАМ

Channel Specific

- Branch Location Management
- Branch Network Management
- Contact Center Management
- Contact Center Operations
- E-Branch Management
- E-Branch Operations
- ATM Network Management
- ATM Network Operations
- Branch Location Operations
- Branch Currency Management
- Branch Currency Distribution
- Advanced Voice Services Management
- Advanced Voice Services Operations
- Product Inventory Item Management
- Product Inventory Distribution
- Card Terminal Operation
- Card Terminal Administration

Cross Channel

- Point of Sale Operations
- Transaction Authorization
- Point of Service
- Servicing Item History
- Servicing Activity Analysis
- Channel Manager
- Contact Routing
- Agent Desktop
- Interactive Help
- Customer Workbench

Marketing

- Business Development
- Event Management
- Advertising
- Promotional Events
- Prospect Campaign Management
- Prospect Campaign Design
- Customer Campaign Management
- Customer Campaign Design
- Customer Survey

Sales

- Prospect Campaign Execution
- Customer Campaign Execution
- Prospect Management (RELATED)
- Prospect Management
- Sales Activity Management Order
- Sales Planning
- Underwriting
- Underwriter Office
- Product Marketing
- Commission Agreement
- Commissions
- Product Expert Sales Support
- Product Sales Support
- Sales Product

Customer Management

- Customer Relationship Management
- Customer Event History
- Customer Profile Management
- Customer Profile
- Customer Proposition
- Customer Product Service Eligibility
- Customer Access Certification
- Customer Agreement
- Sales Product Agreement
- Customer Behavioral Insights
- Customer Credit Rating
- Account Receivable

Servicing

- Customer Case
- Customer Case Management
- Case Root Cause Analysis
- Customer Order
- Servicing Issue
- Servicing Order (B2B Customer Order)
- Payment Initiation(NON)
- Servicing Mandate (NON)

OPERATIONS & EXECUTION / ОПЕРАЦИИ И ИСПОЛНЕНИЕ

PRODUCT SPECIFIC FULFILLMENT

Loans & Deposits

- Loan
- Leasing
- Corporate Loan
- Corporate Lease
- Merchandising Loan
- Mortgage
- Corporate Deposits
- Current Account
- Savings Account
- Corporate Current Account
- Fiduciary Agreement

Cards

- Card Processing Unit
- Card Authorization
- Card Capture
- Card Billing & Payments
- Merchant Acquiring Facility
- Card Network Participant Facility
- Merchant Relations / Торговые отношения

Consumer Services

- Remittance
- Currency Exchange
- Bank Drafts & Traveler's Checks
- Brokered Product
- Consumer Investments
- Customer Tax Handling
- Service Product
- Consumer Advisory Services
- Trust Services
- Corporate Trust Services

Investment Management

- Investment Portfolio Planning
- Investment Portfolio Analysis
- Investment Portfolio Management
- eTrading Workbench

Wholesale Trading

- Trading Book Oversight
- Trading Models
- Program Trading
- Traded Position Management
- Dealer Workbench
- Suitability Checking
- Credit Risk Operations
- Quote Management
- Market Making
- ECM / DCM
- Market Order
- Market Order Execution

Market Operations

- Mutual Fund Administration
- Hedge Fund Administration
- Unit Trust Administration
- Custody Administration
- Trade Confirmation Matching
- Trade Price Reporting
- Order Allocation
- Corporate Events
- Financial Instrument Valuation
- Securities Fails Processing
- Securities Delivery & Receipt Management
- Settlement Obligation Management

Trade Banking

- Letter of Credit
- Syndicated Loan
- Bank Guarantee
- Trade Finance
- Project Finance
- Cash Management
- Credit Facility
- Direct Debit
- Direct Debit Mandate
- Limit & Exposure Management
- Cash Management & Account Services
- Cheque Lock Box
- Factoring

Corporate Financing & Advisory Services

- Corporate Finance
- Corporate Tax Advisory
- M&A Advisory
- Public Offering
- Private Placement

CROSS PRODUCT OPERATIONS

Payments

- Payment Operations
- Financial Gateway
- Financial Message Analysis
- Correspondent Bank
- Cheque Processing
- Card e-Commerce Gateway
- Central Cash Handling
- Financial Order
- Payment Order
- Servicing Issue

Collateral Administration

- Collateral Allocation Management
- Collateral Asset Administration
- Collections

Account Management

- Account Opening
- Position Management
- Customer Position
- Client Money Pool
- Accounts Receivable
- Account Reconciliation
- Reward Points Account
- Fraud Detection
- Fraud Case
- Transaction Engine
- Product Combination

Operational Services

- Event Management
- Event Notice Tracking
- Customer Billing
- Rewards Points Awards & Redemption
- Exchange
- Channel Activity History
- Channel Activity Analysis
- Leasing Item Administration
- Spot Procurement (NON)
- Open Item Management

RISK & COMPLIANCE

Bank Portfolio & Treasury

- Corporate Finance
- Corporate Treasury Analytics
- Bank Portfolio Administration
- Bank Portfolio Analysis
- Asset & Liability Management
- Stock Lending Repos
- Asset Securitization

Models

- Business Risk Models
- Market Risk Models
- Credit Risk Models
- Liquidity Risk Models
- Production Risk Models
- Operational Risk Models
- Pricing Models
- Contribution Models
- Customer Behavior Models
- Financial Instrument Valuation Models
- Gap Analysis
- Economic Capital
- Credit Margin Management

Business Analytics

- Product Portfolio
- Customer Portfolio
- Branch Portfolio
- Channel Portfolio
- Segment Direction
- Market Research
- Competitor Analysis
- Market Models
- Contribution Analysis

Regulations & Compliance

- Regulatory Compliance
- Guideline Compliance
- Compliance Reporting
- Fraud/AML Assurance
- Financial Accounting

BUSINESS SUPPORT

IT Management

- Systems Operations
- Systems Administration
- Systems Assurance
- Platform Operations
- Development Environment
- System Development
- System Deployment
- Production Release
- IT Systems Direction
- IT Standards & Guidelines
- Systems Help Desk
- Internal Network Operation

Non IT & HR: Enterprise Services

- Legal Compliance
- Internal Audit
- Security Advisory
- Security Assurance
- Approved Supplier Directory
- Procurement
- Company Billing & Payments
- Fixed Asset Register

Buildings, Equipment and Facilities

- Property Portfolio
- Building Maintenance
- Utilities Administration
- Site Administration
- Site Operations
- Equipment Administration
- Equipment Maintenance

Business Command & Control

- Organization Direction
- Business Unit Direction
- Business Unit Management
- Business Unit Financial Analysis
- Business Unit Financial Operations
- Business Unit Accounting

Finance

- Financial Statements
- Financial Control
- Financial Compliance
- Enterprise Tax Administration

Human Resource Management

- Human Resources Direction
- Employee Assignment
- Employee Data Management
- Recruitment
- Employee/Contractor Contract
- Employee Access
- Workforce Training
- Employee Certification
- Employee Evaluation
- Employee Payroll and Incentives
- Employee Benefits
- Travel and Expenses

Knowledge & IP Management

- Management Manual
- Intellectual Property Portfolio
- Knowledge Exchange

Corporate Relations

- Corporate Communications
- Corporate Alliance Stakeholder
- Corporate Relationship
- Investor Relations
- Regulatory and Legal Authority

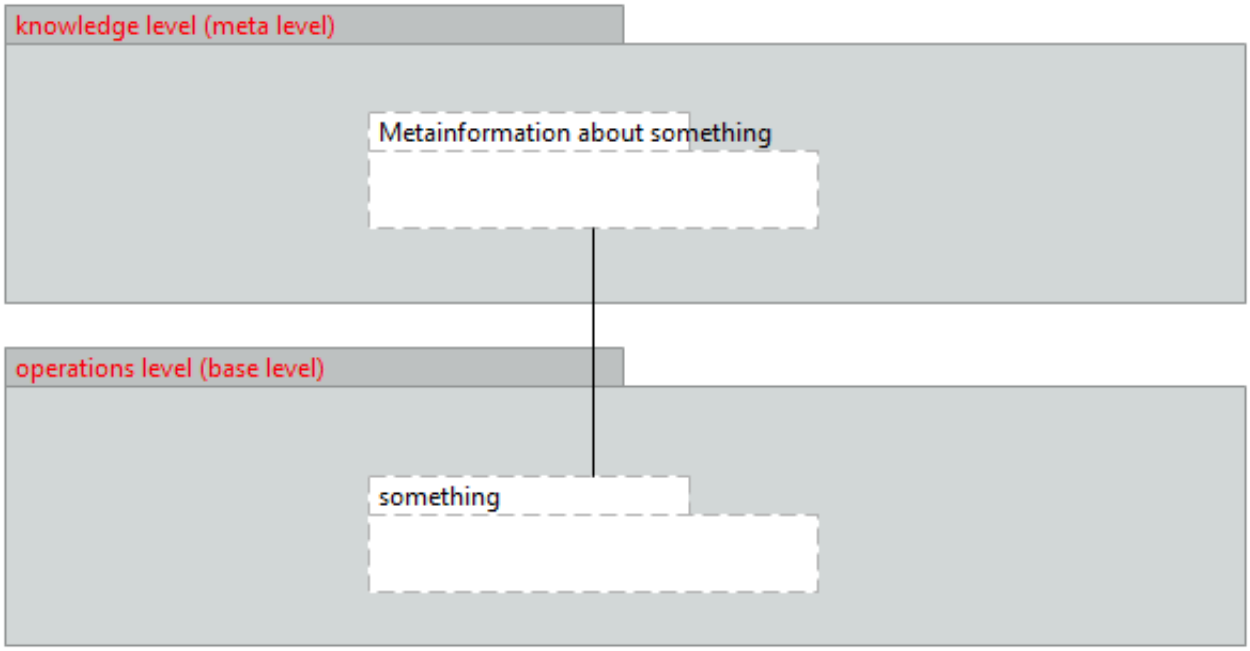
Business Direction

- Corporate Strategy
- Corporate Policies
- Product & Services Direction
- Business Architecture
- Community Planning

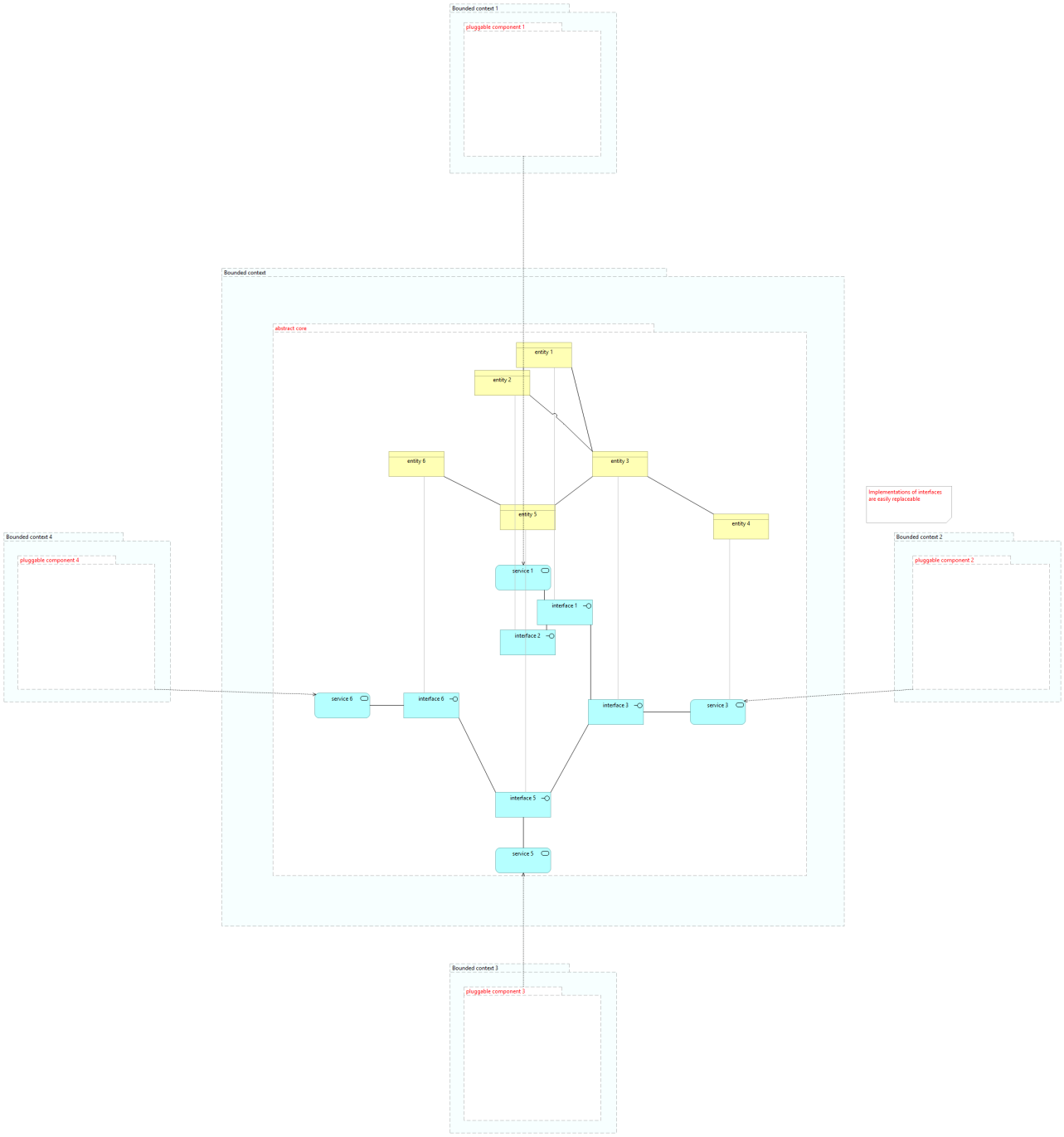
Document Management & Archive

- Document Management
- Archive Services
- Correspondence

KNOWLEDGE LEVEL



PLUGGABLE COMPONENT FRAMEWORK

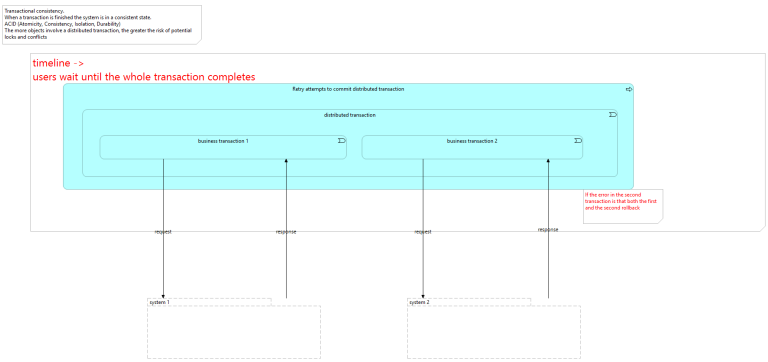


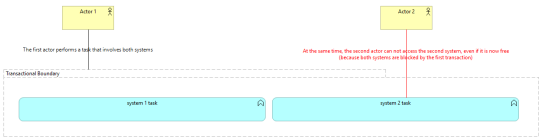
ADDITIONAL PATTERNS

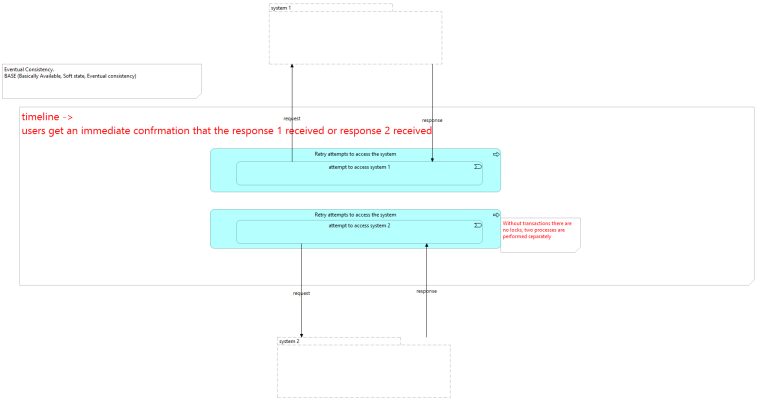
DOMAIN DRIVEN DESIGN

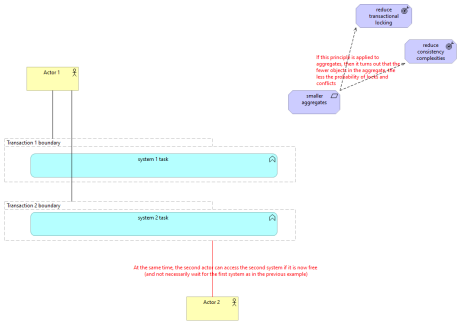
Eric Evans

TYPES OF CONSISTENCY

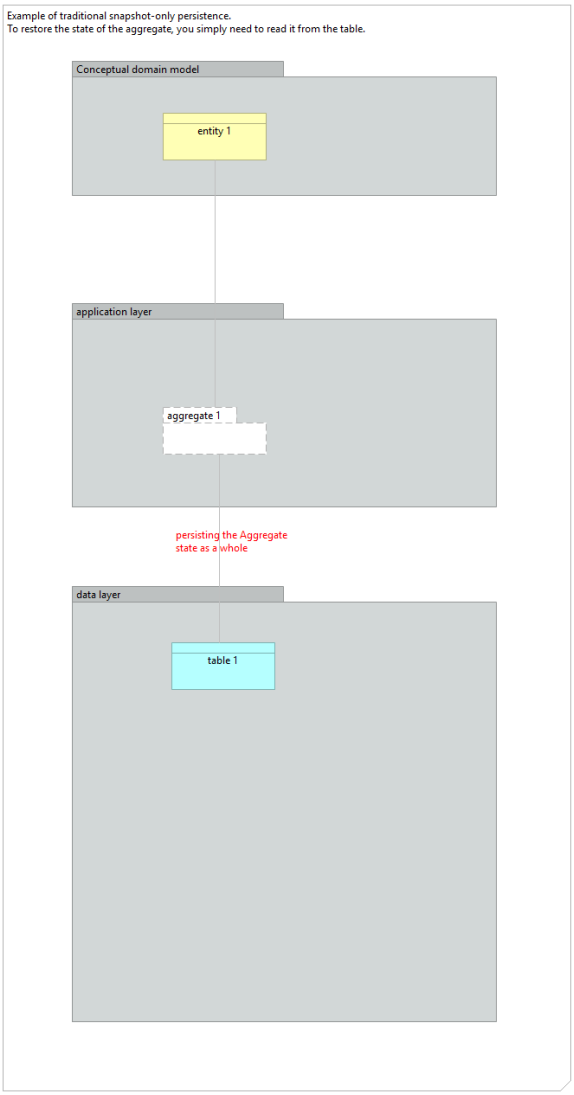


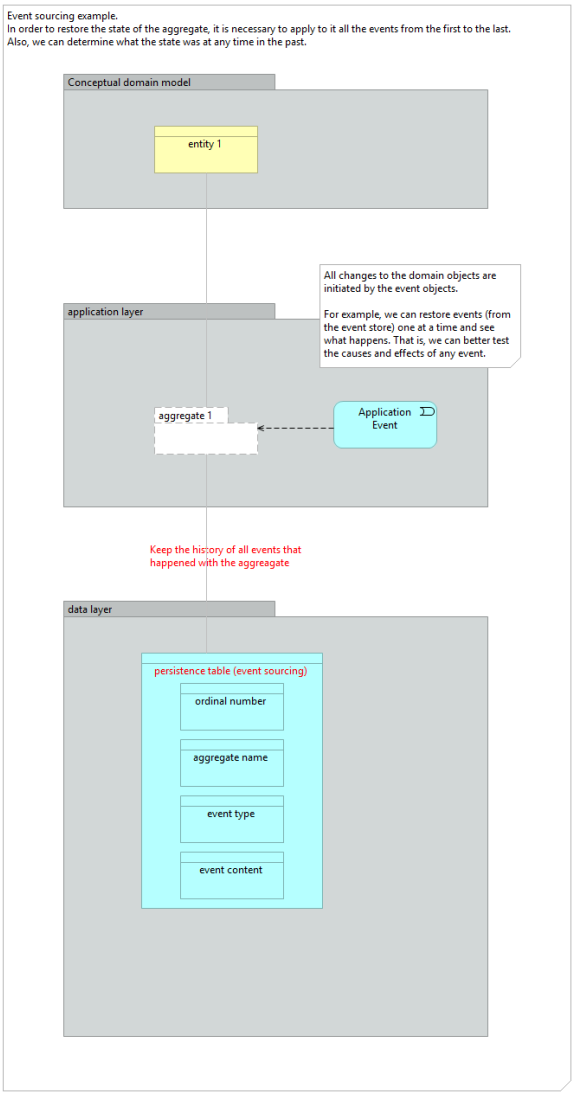




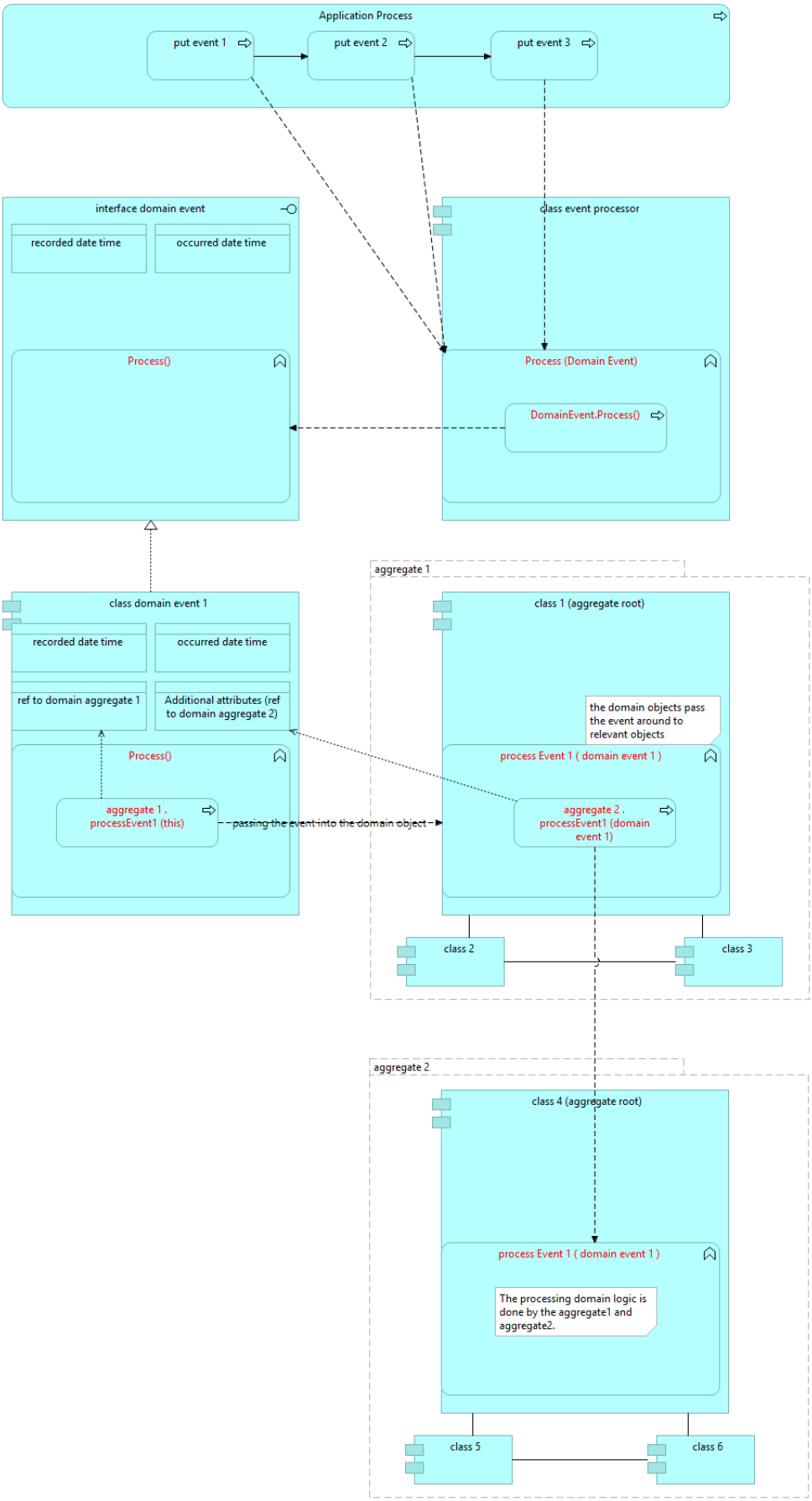


EVENT SOURCING

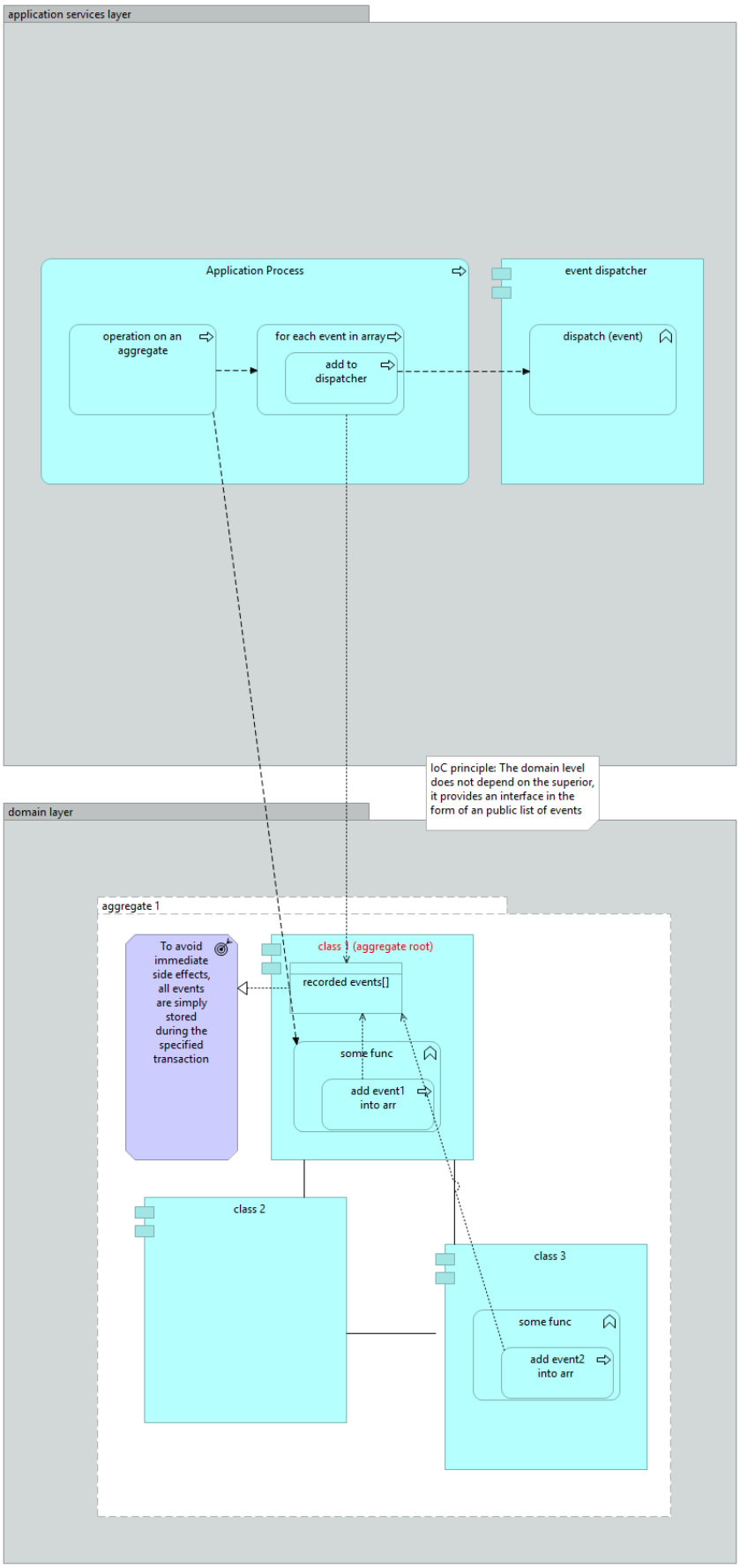




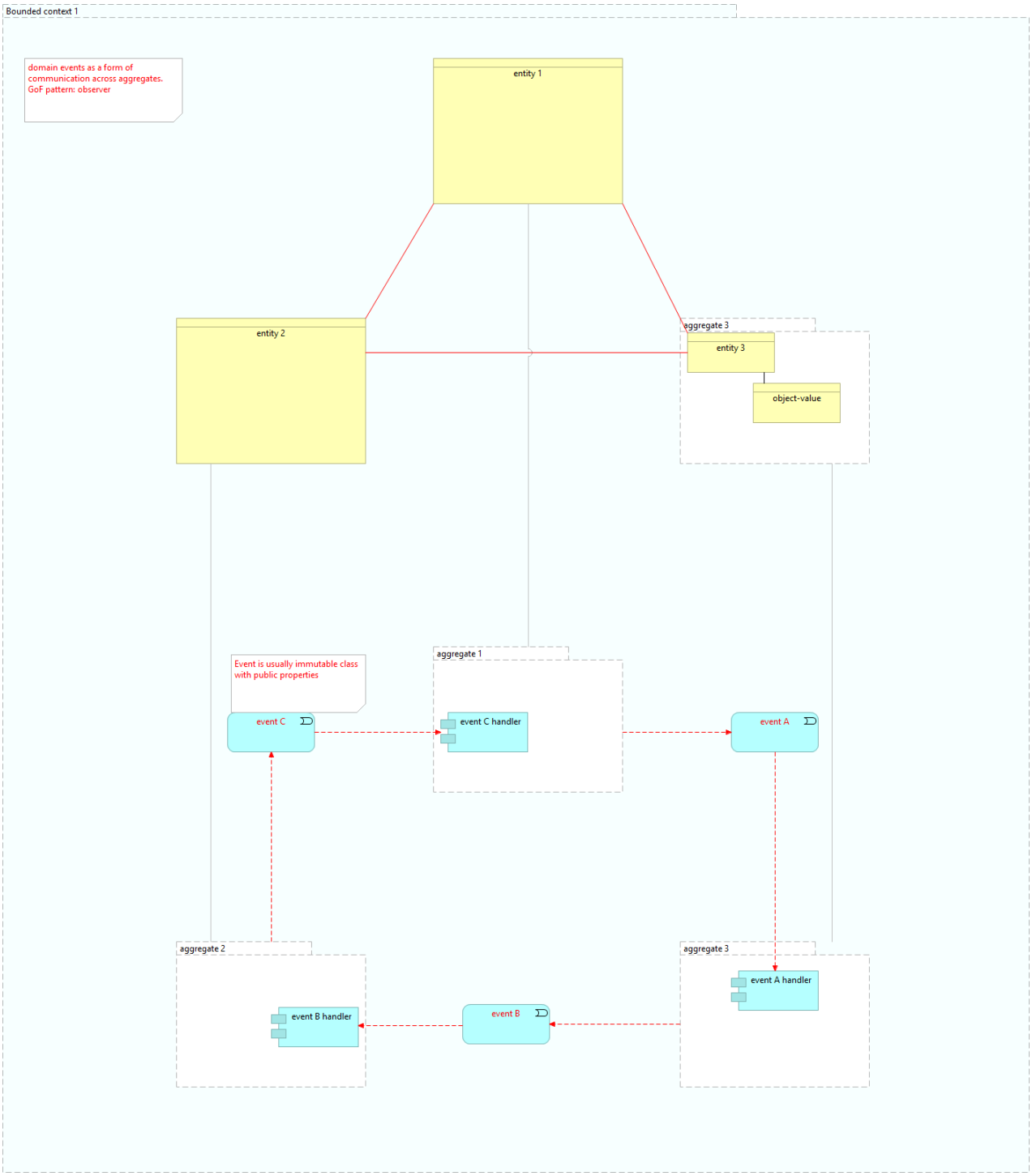
EVENT PROCESSOR



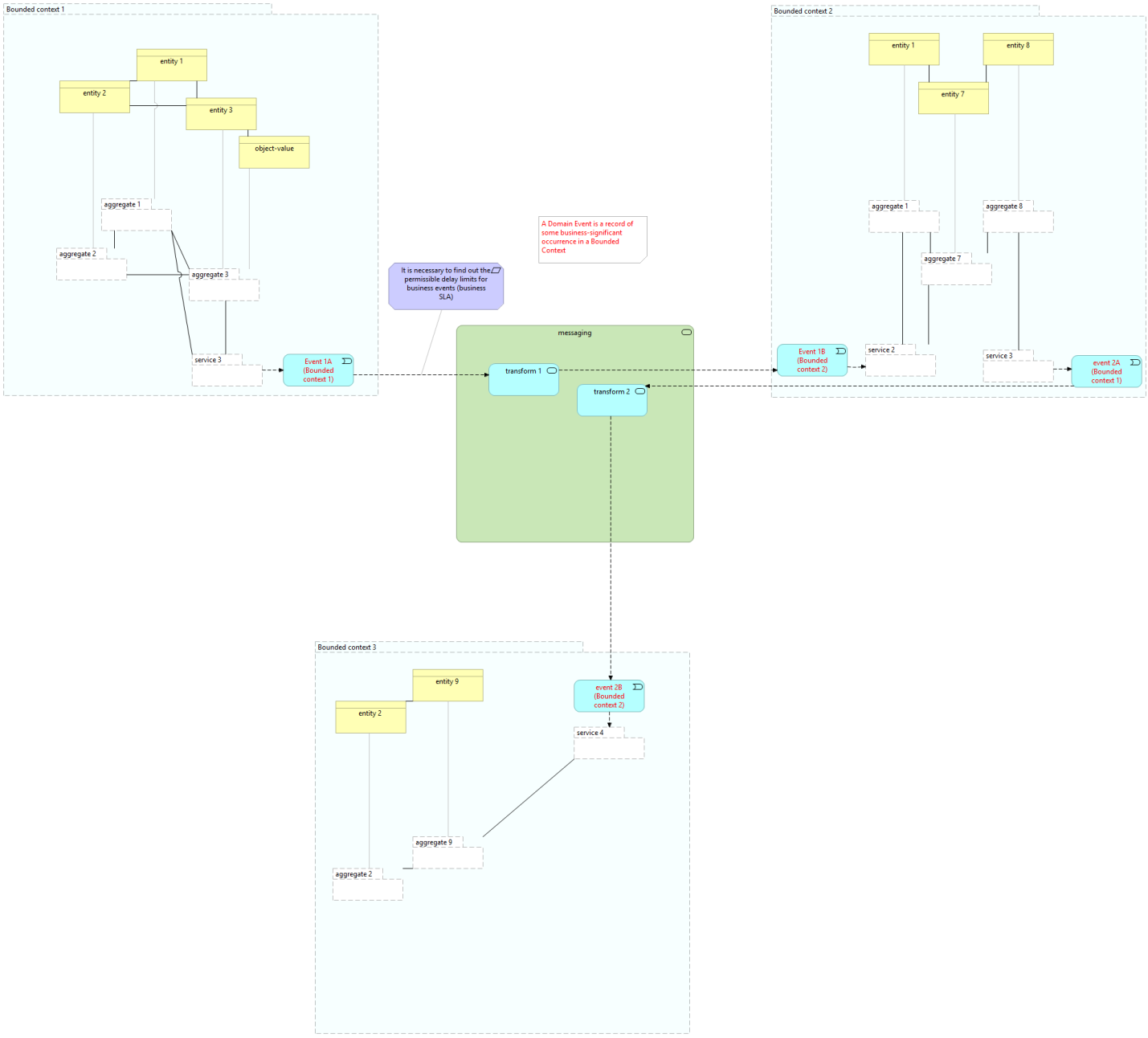
EVENT DISPATCHER



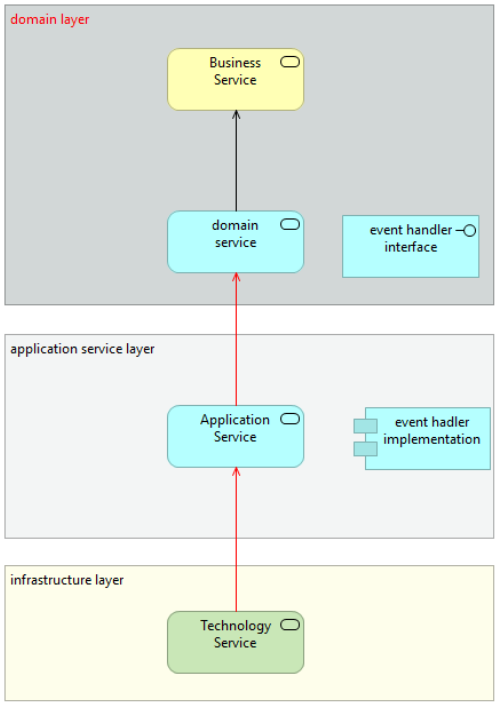
INTERNAL DOMAIN EVENTS

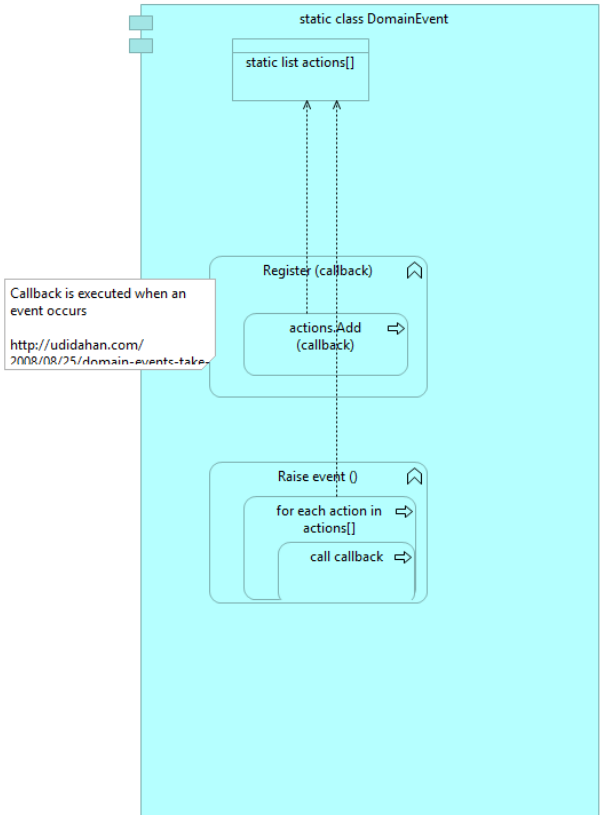


EXTERNAL DOMAIN EVENTS, TRANSFER BETWEEN CONTEXTS

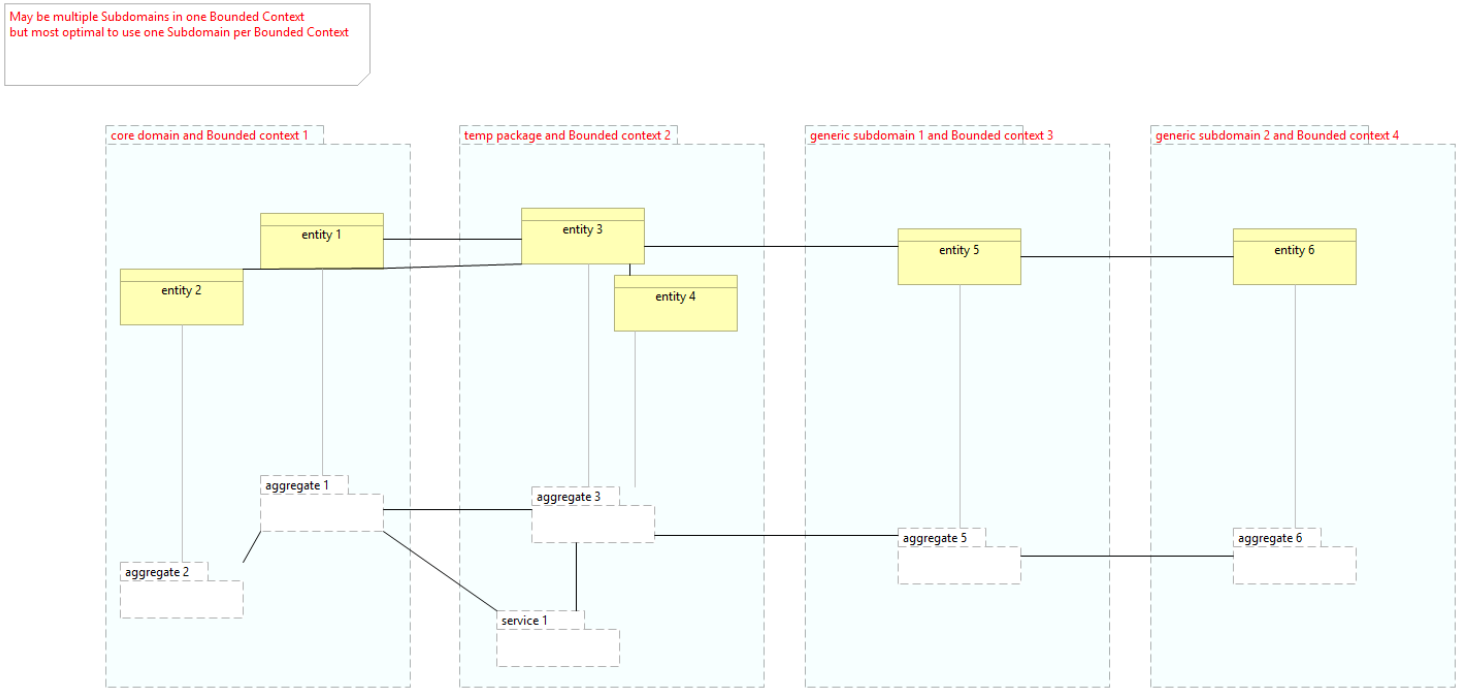


STATIC DOMAIN EVENTS CLASS

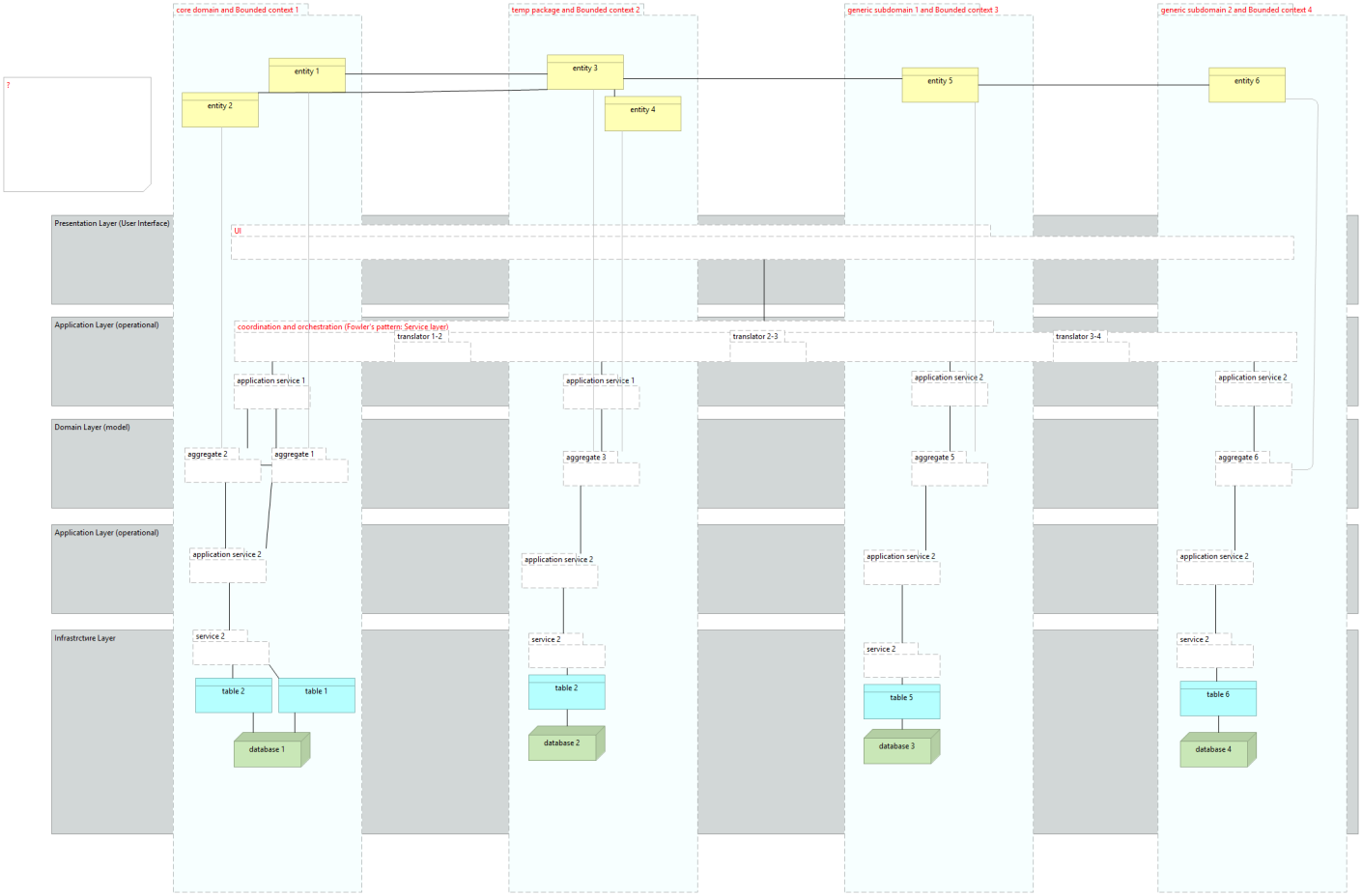




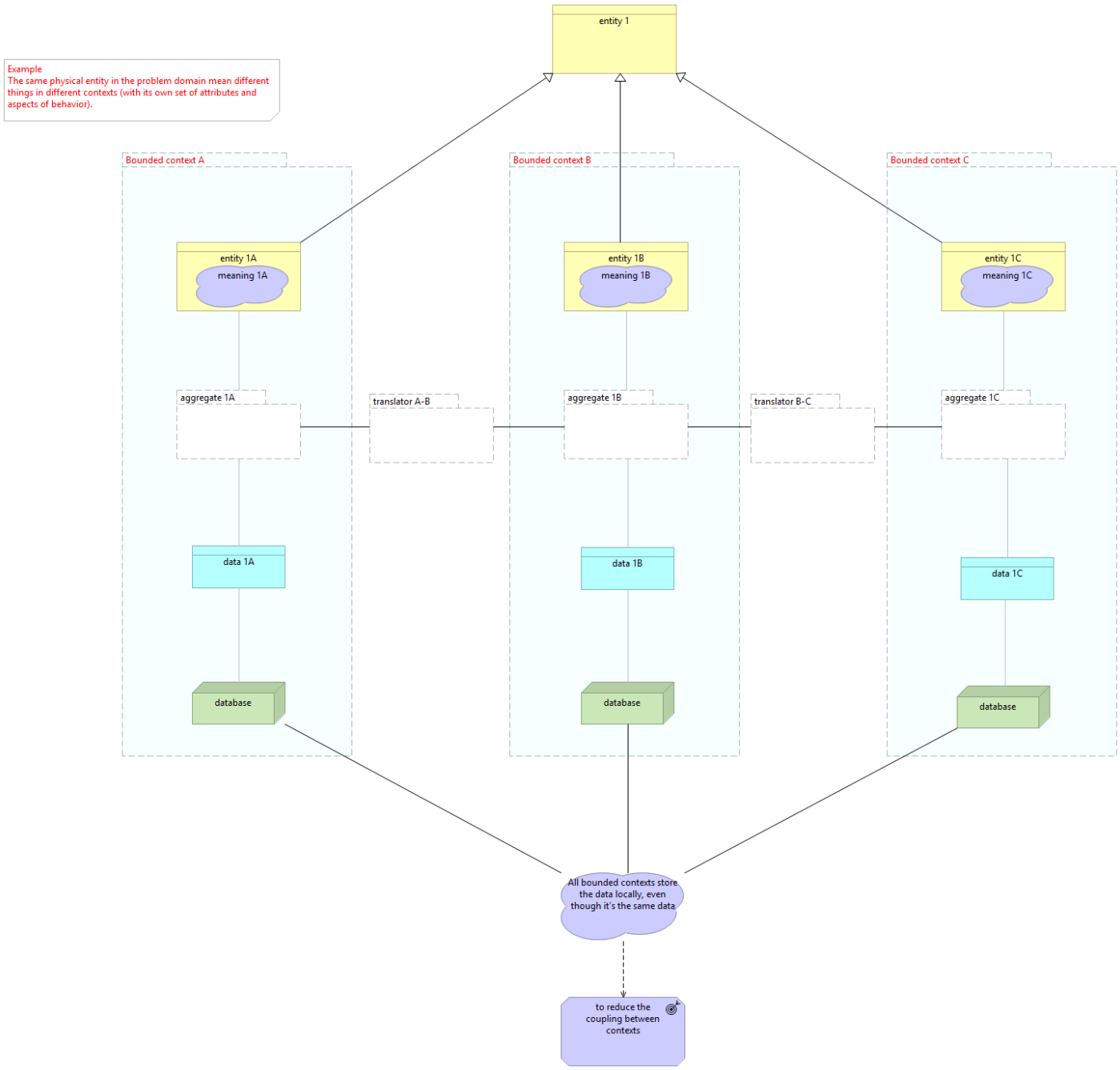
ONE SUBDOMAIN PER BOUNDED CONTEXT



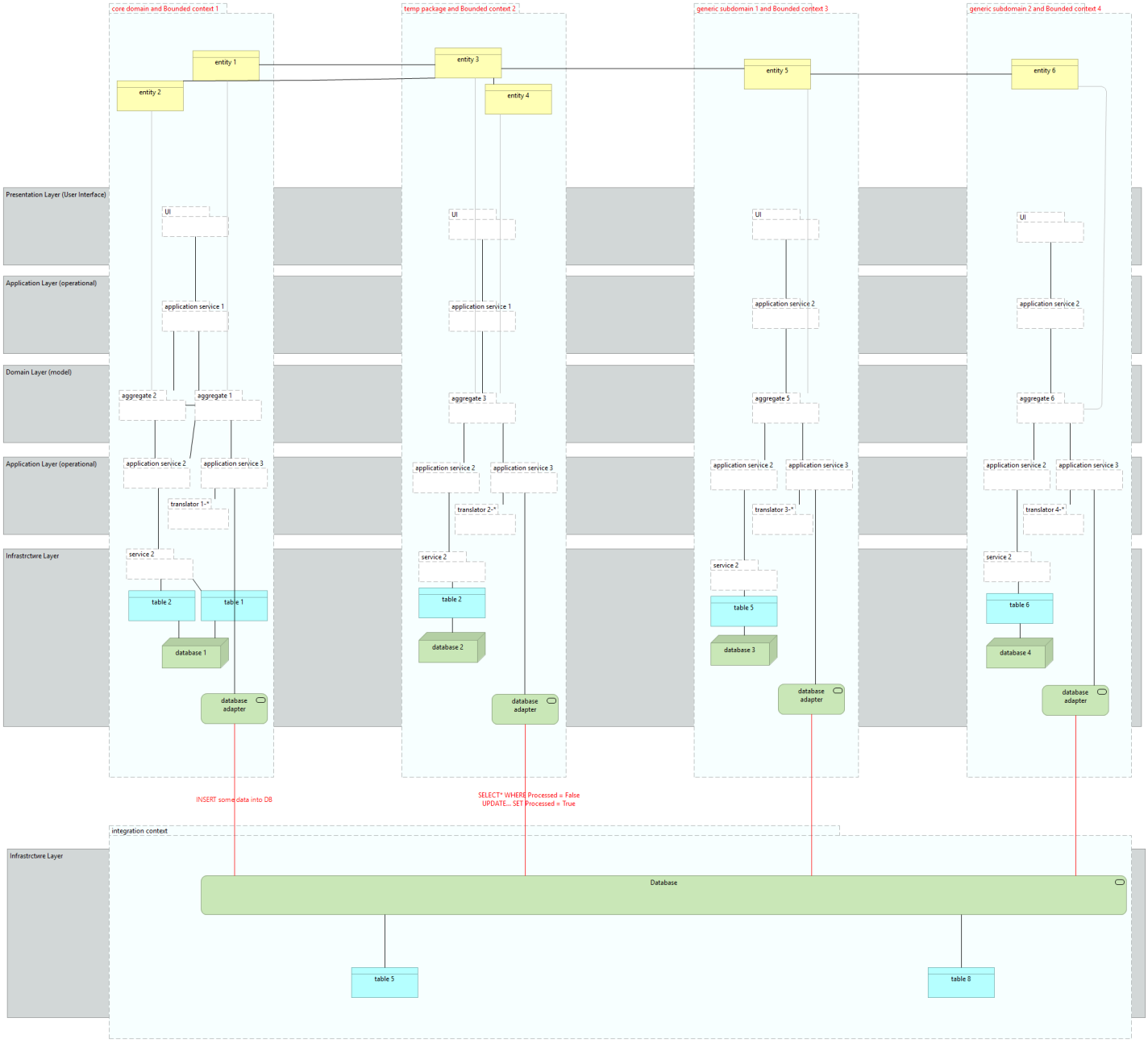
THE APPLICATION LAYER COORDINATES THE WORK BETWEEN CONTEXTS



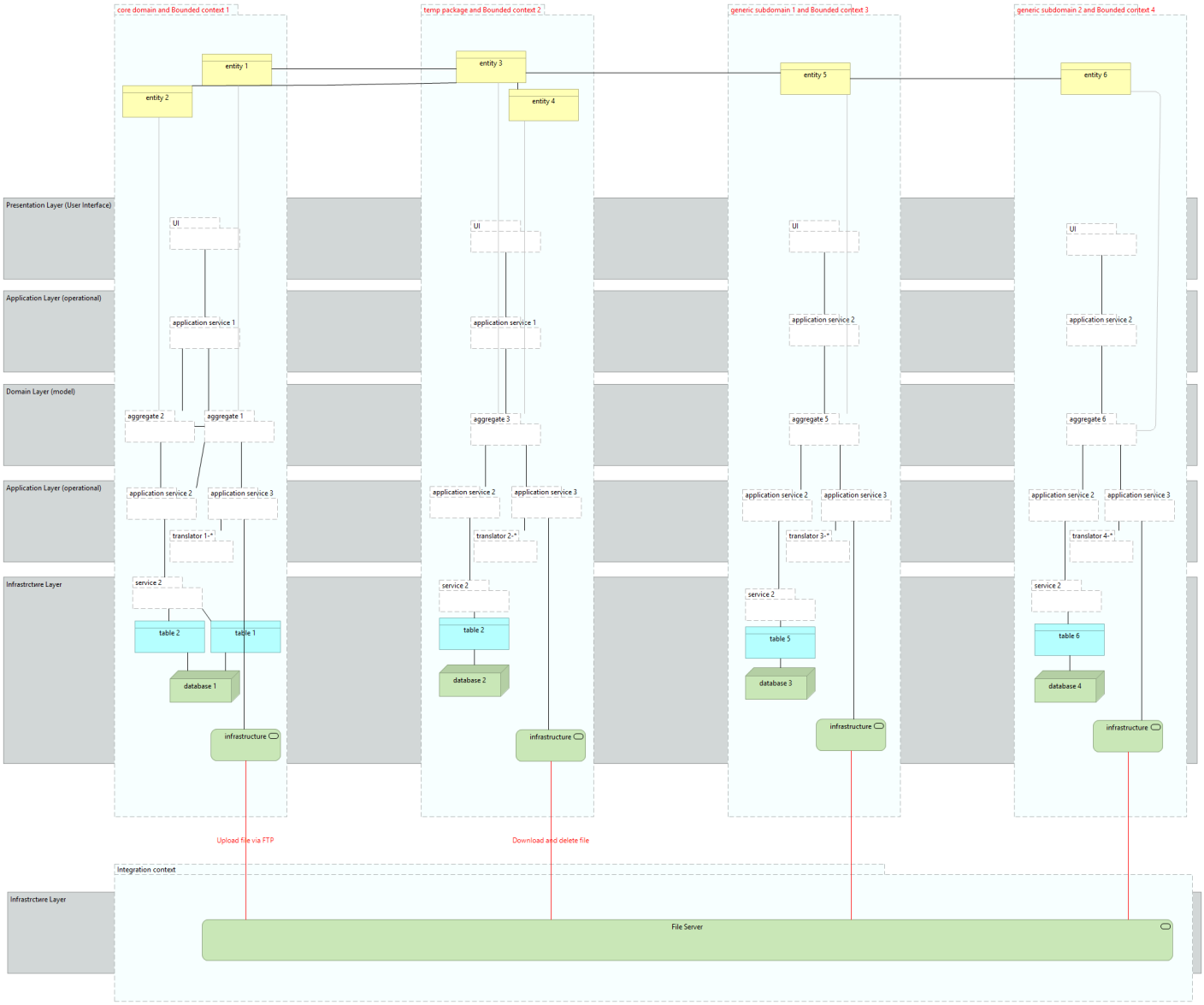
THE SAME PHYSICAL ENTITY IN DIFFERENT CONTEXTS



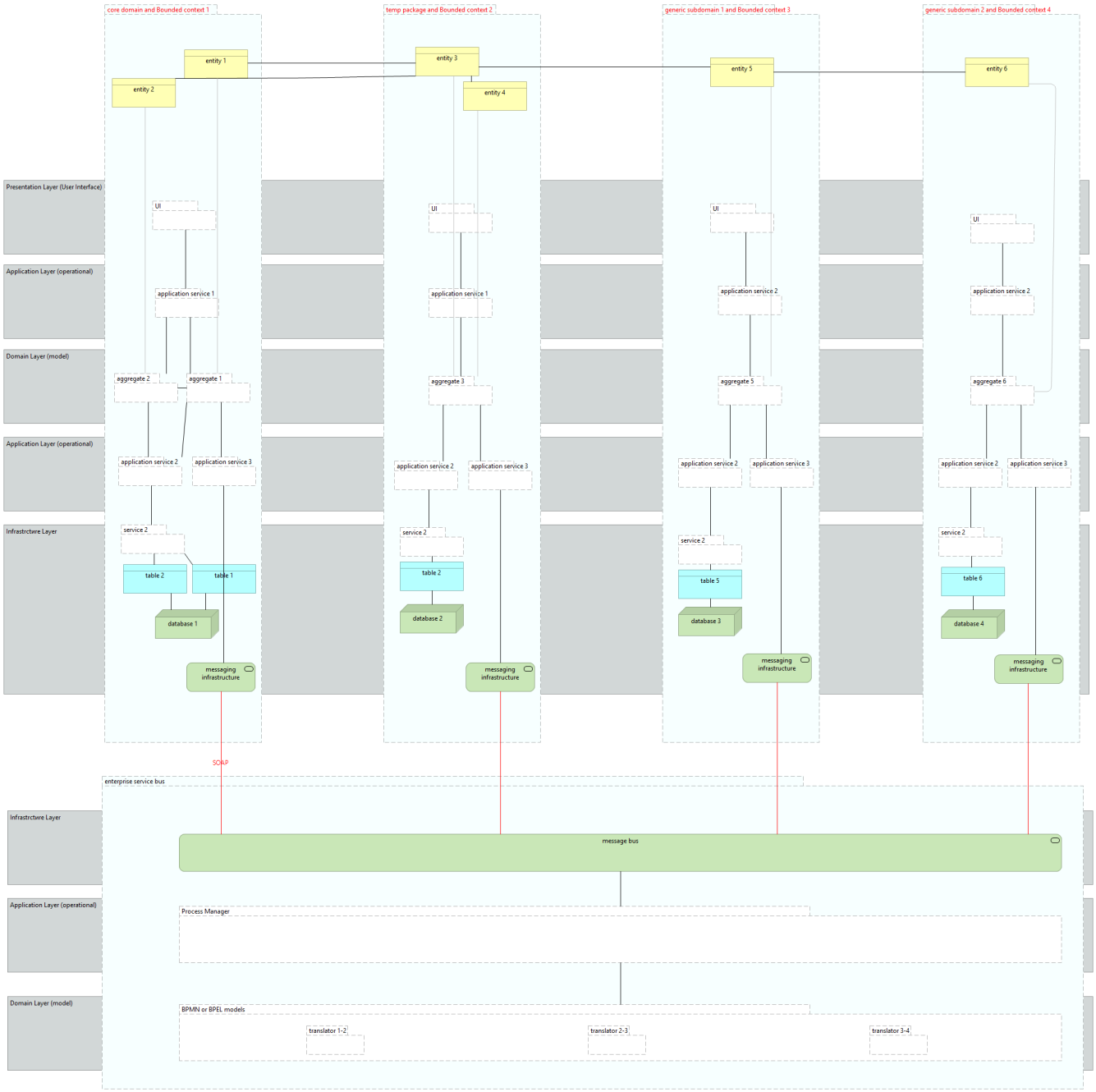
INTEGRATION OF BOUDED CONTEXTS THROUGH DATABASE



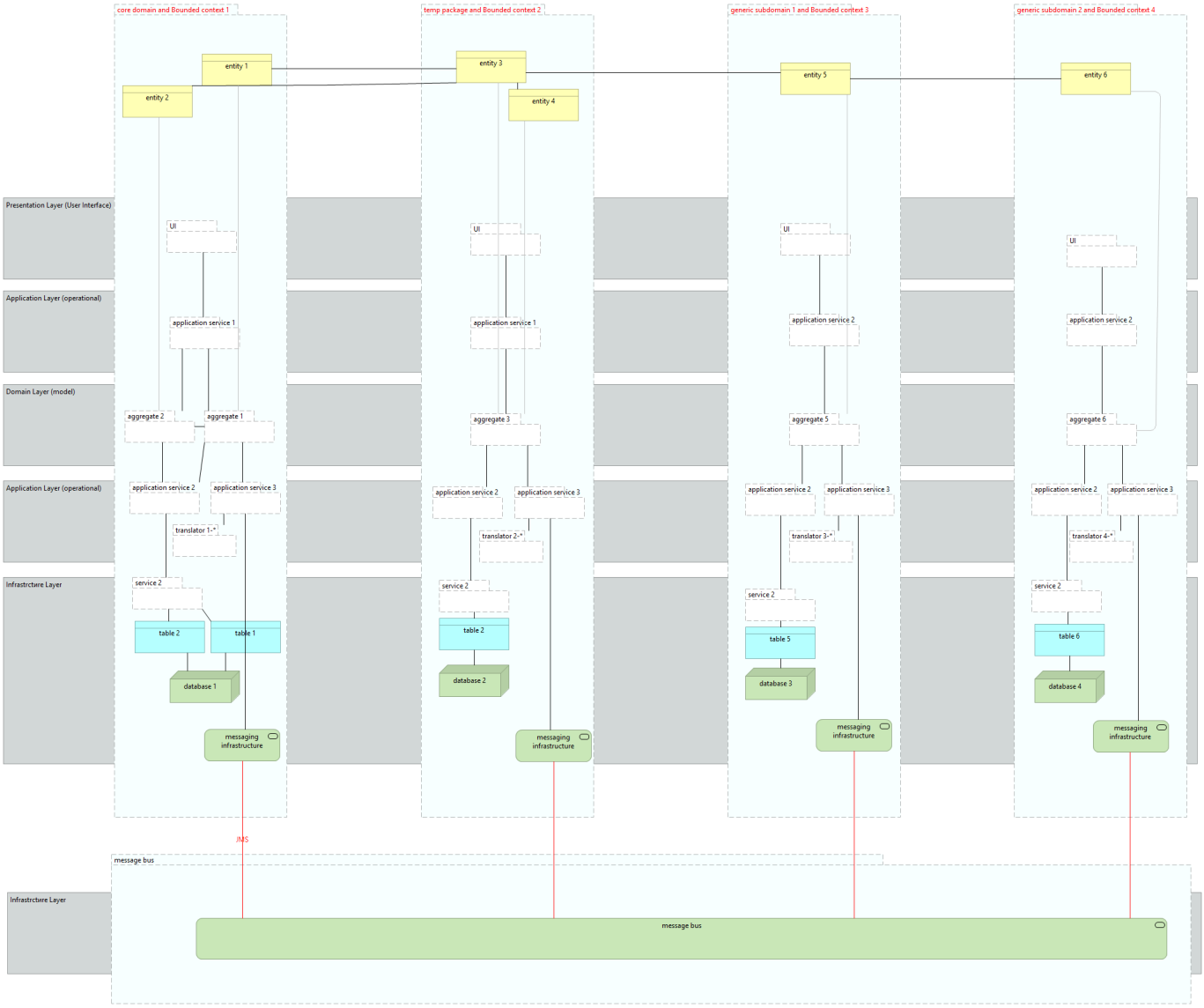
INTEGRATION OF BOUDED CONTEXTS THROUGH FLAT FILES



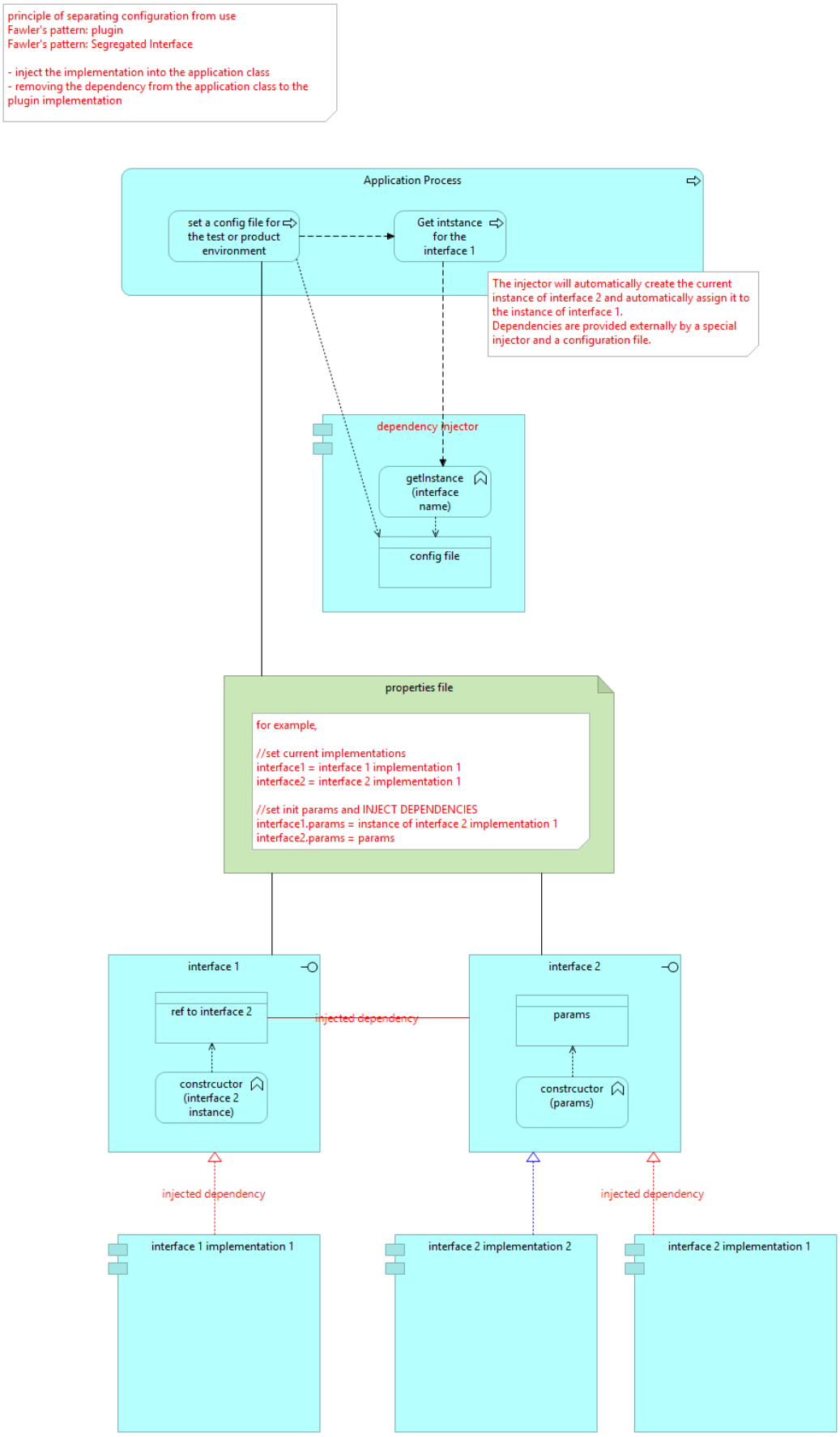
INTEGRATION OF BOUDED CONTEXTS THROUGH ENTERPRISE SERVICE BUS

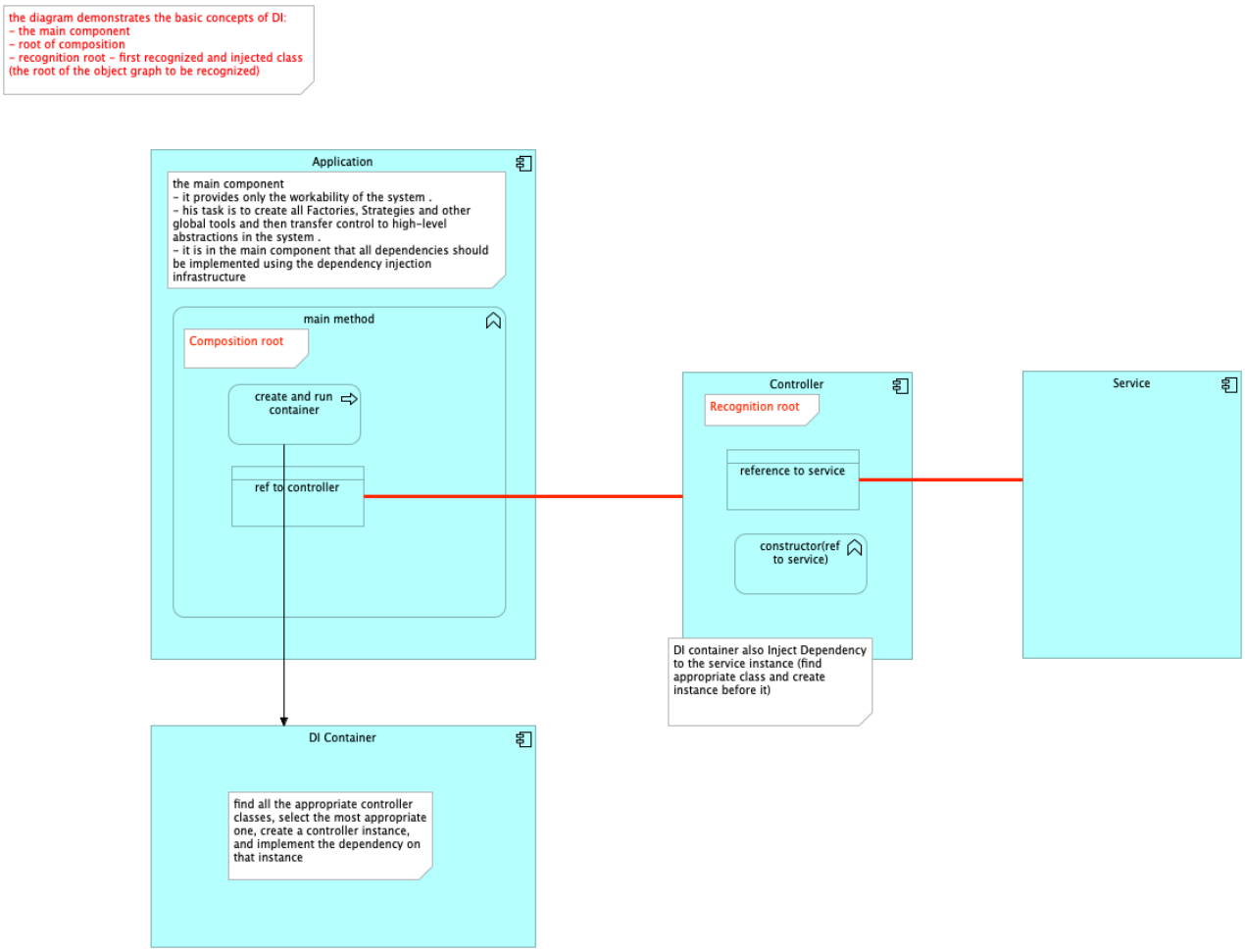


INTEGRATION OF BOUDED CONTEXTS THROUGH MESSAGE QUEUE

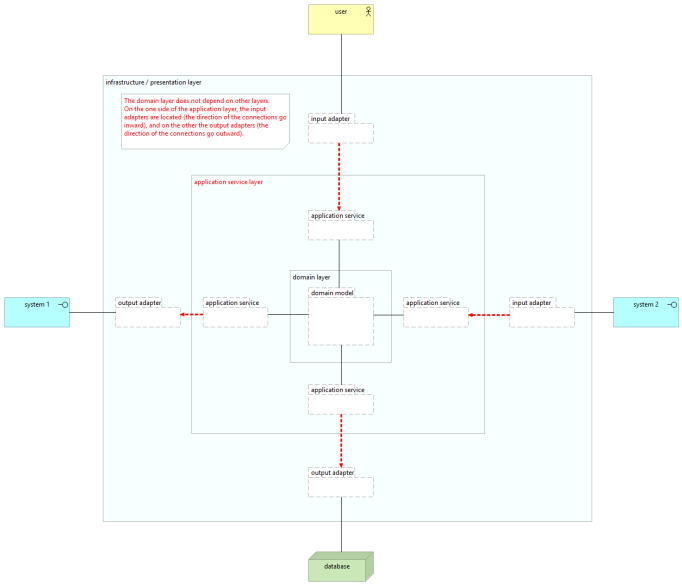


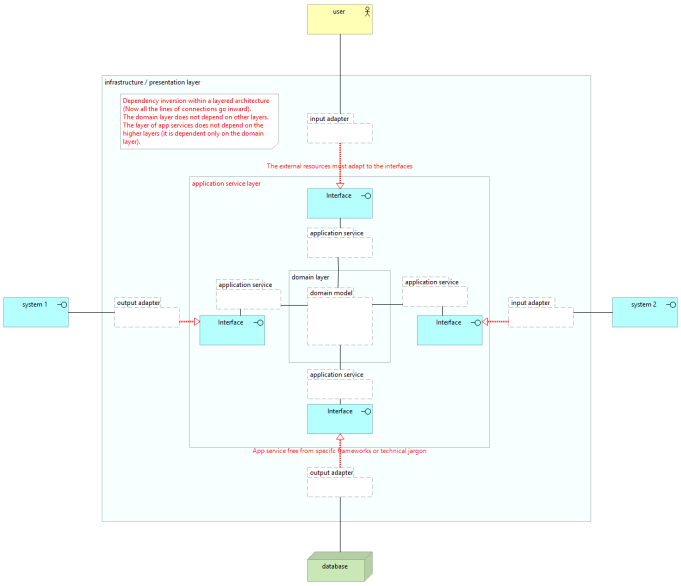
DEPENDENCY INJECTION



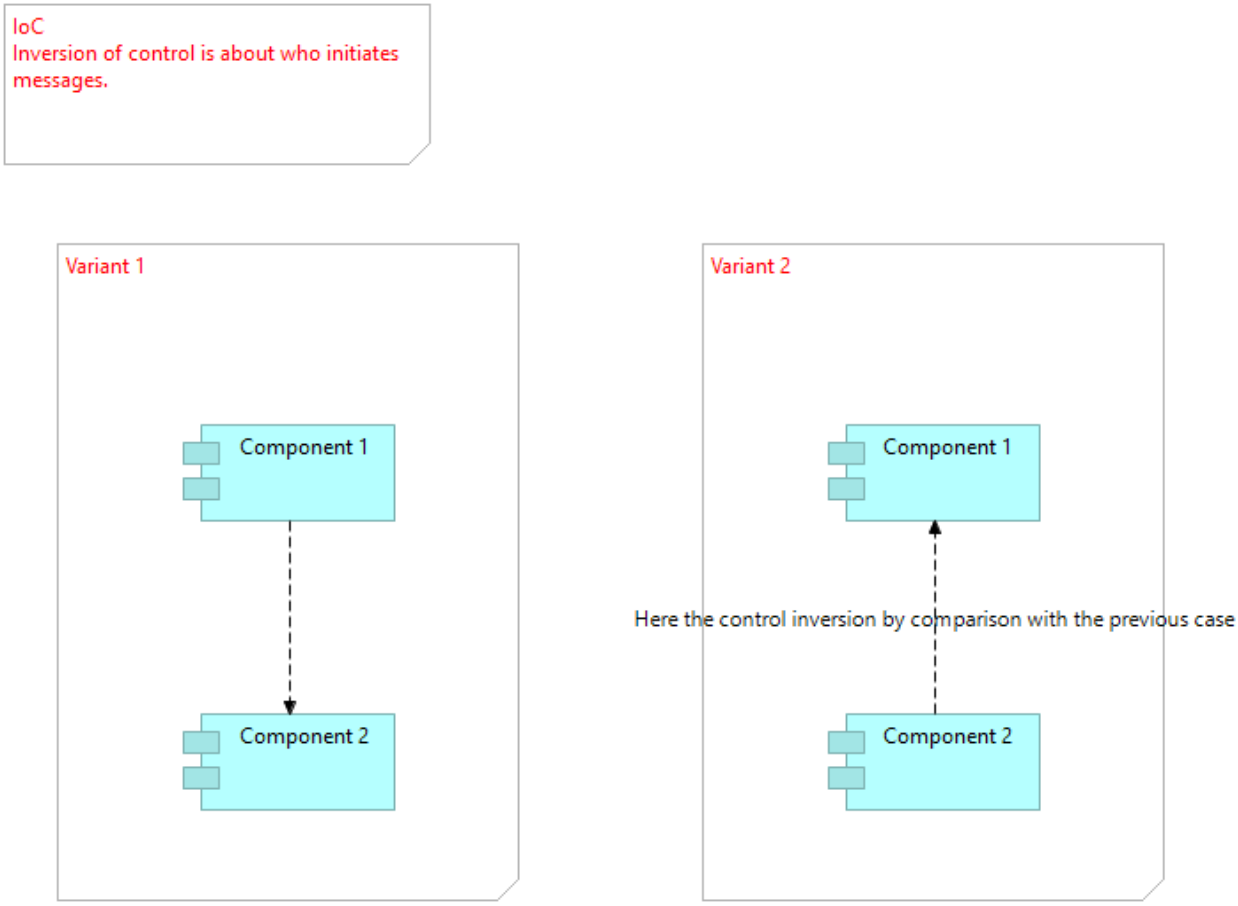

the diagram demonstrates the basic concepts of DI:
- the main component
- root of composition
- recognition root - first recognized and injected class (the root of the object graph to be recognized)
Application
the main component
- it provides only the workability of the system .
- his task is to create all Factories, Strategies and other global tools and then transfer control to high-level abstractions in the system .
- it is in the main component that all dependencies should be implemented using the dependency injection infrastructure
main method
Composition root
create and run container
ref to controller
Controller
Recognition root
reference to service
constructor(ref to service)
DI container also Inject Dependency to the service instance (find appropriate class and create instance before it)
Service
DI Container
find all the appropriate controller classes, select the most appropriate one, create a controller instance, and implement the dependency on that instance

DEPENDENCY INVERSION

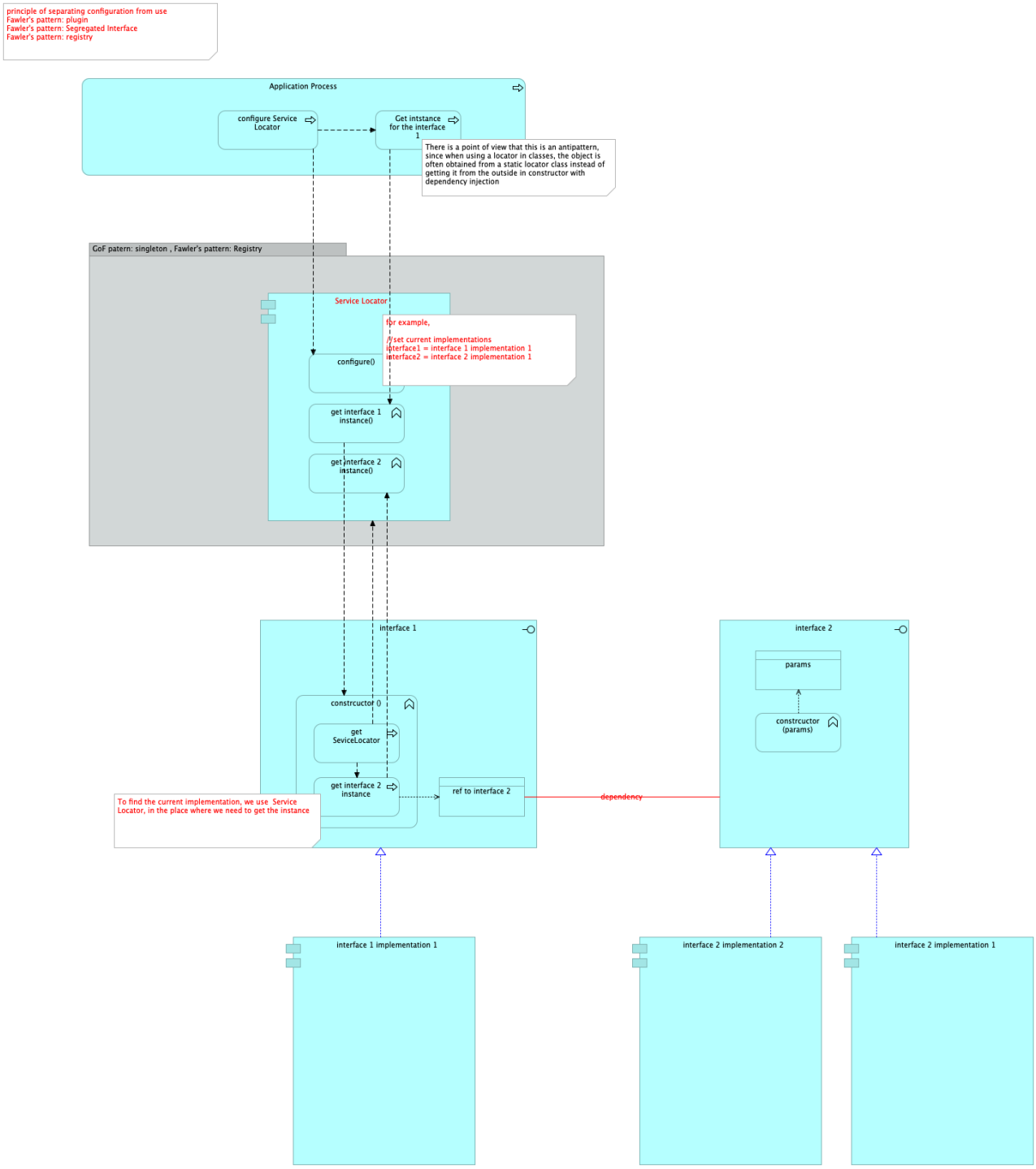




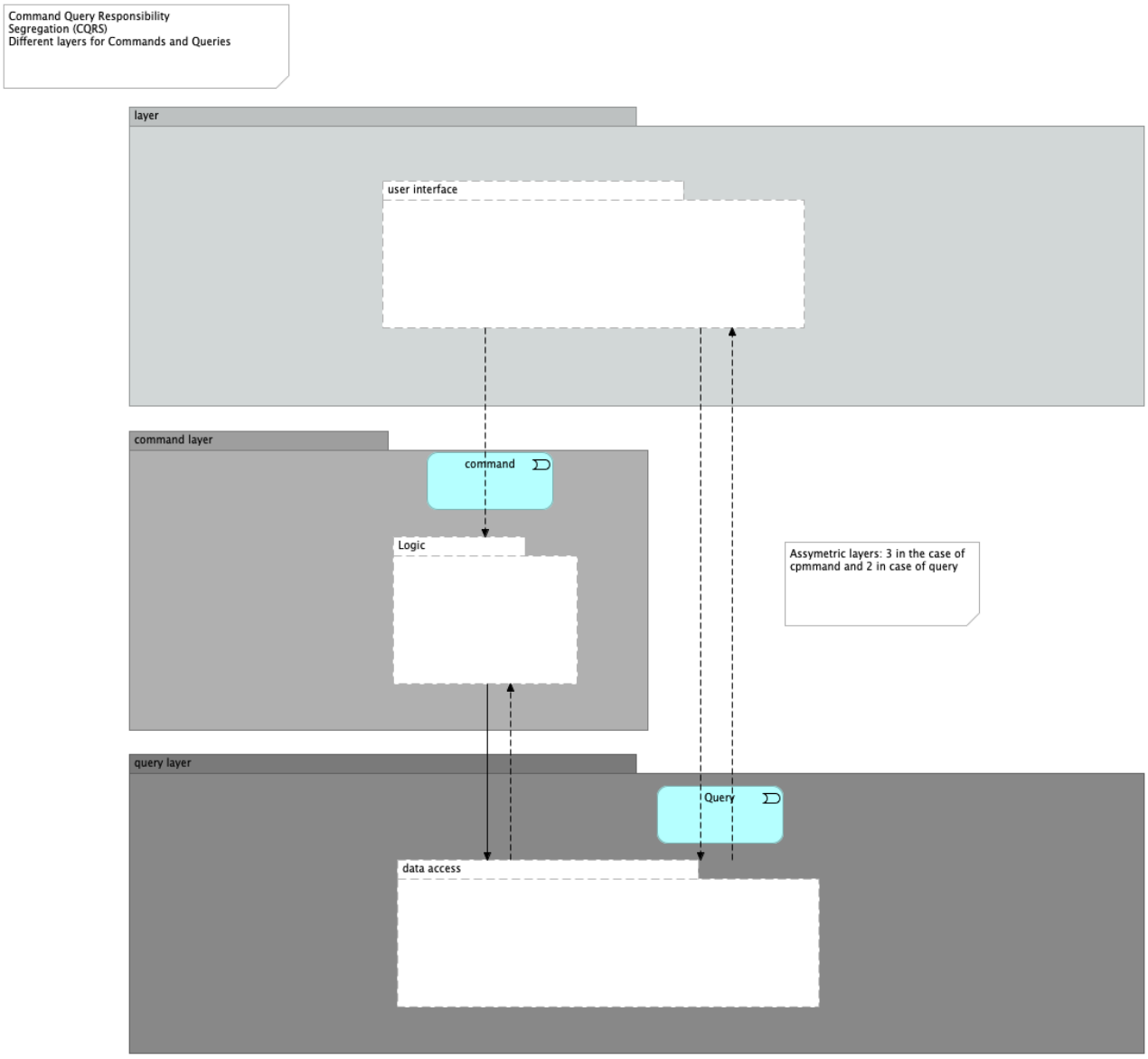
INVERSION OF CONTROL



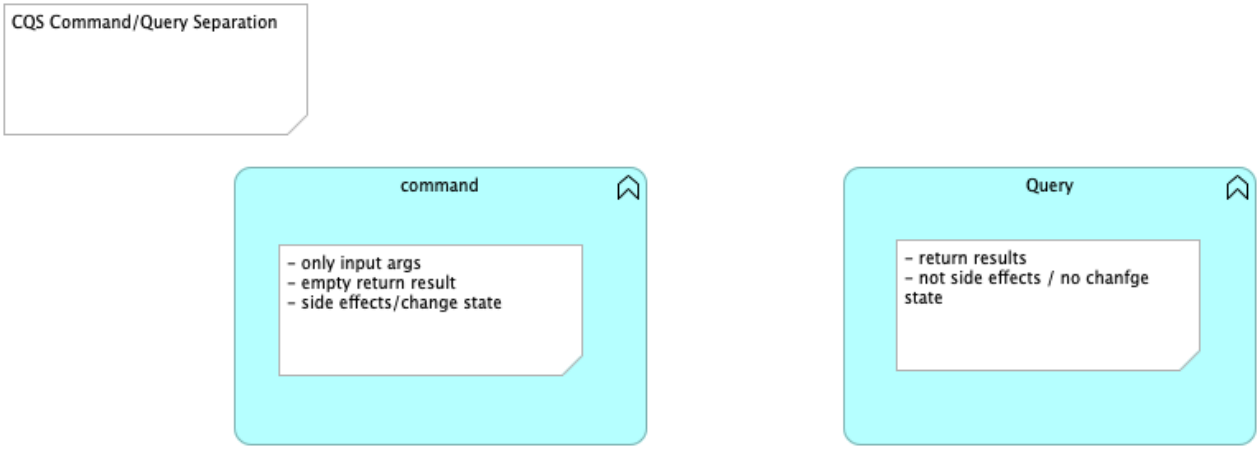
SERVICE LOCATOR



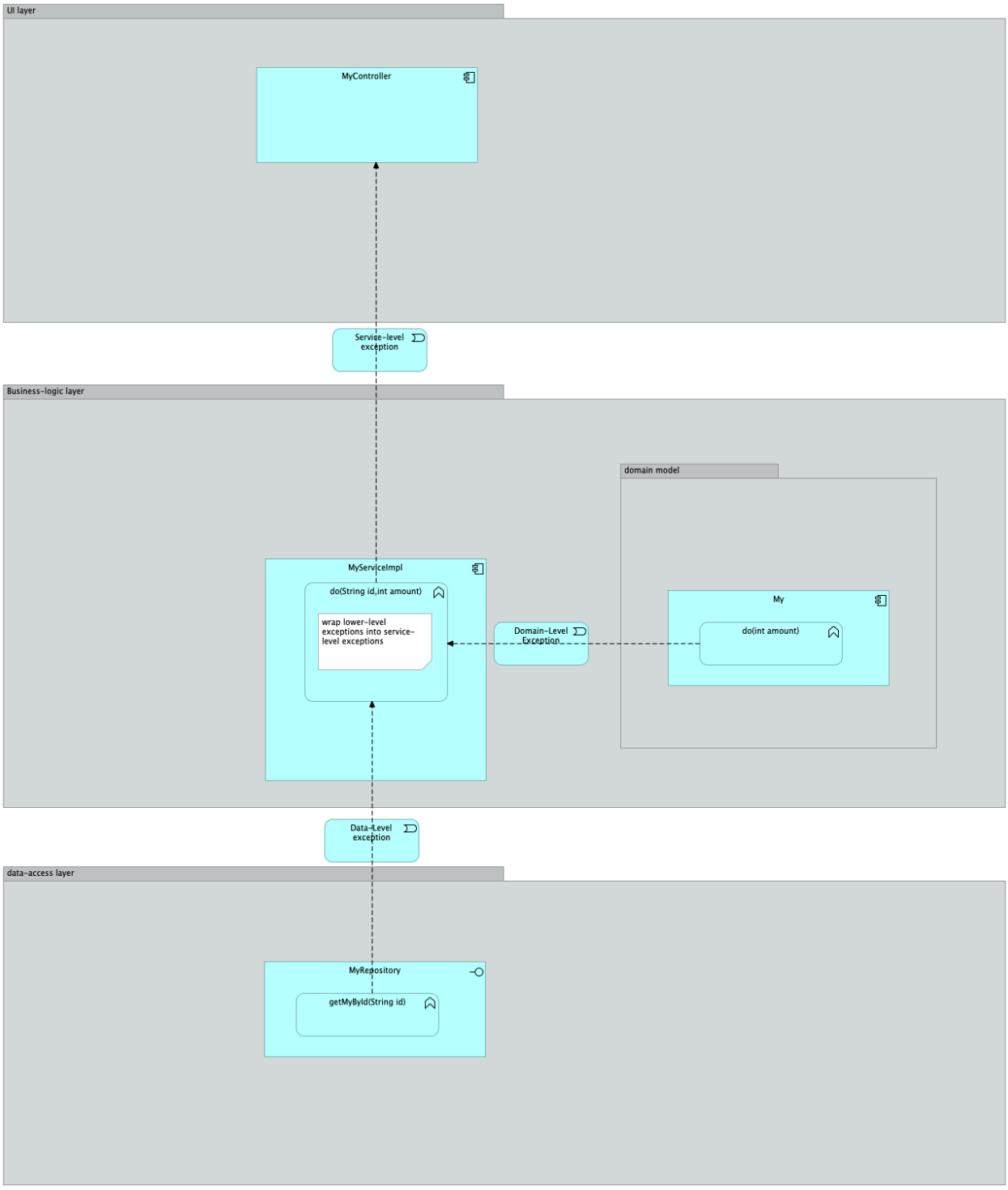
CQRS


Command Query Responsibility
Segregation (CQRS)
Different layers for Commands and Queries
layer
user interface
command layer
command
Logic
Assymetric layers: 3 in the case of
cpmmand and 2 in case of query
query layer
Query
data access

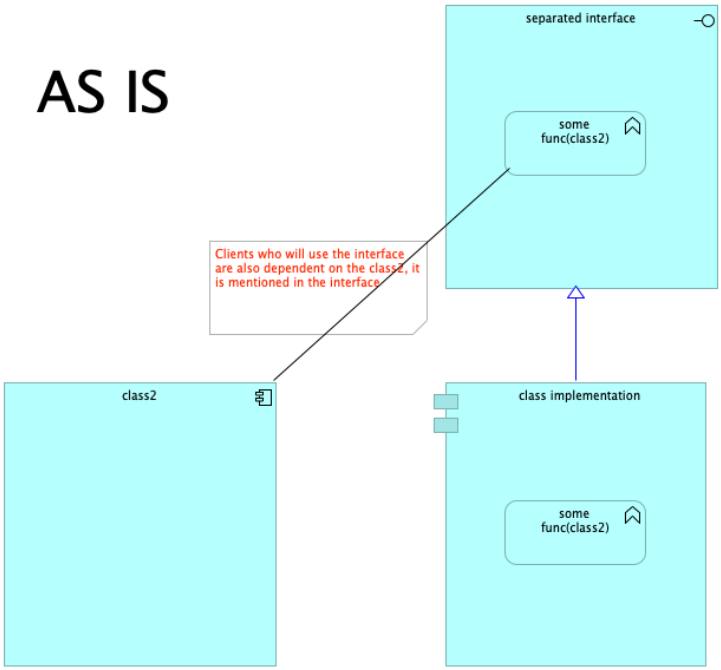
CQS

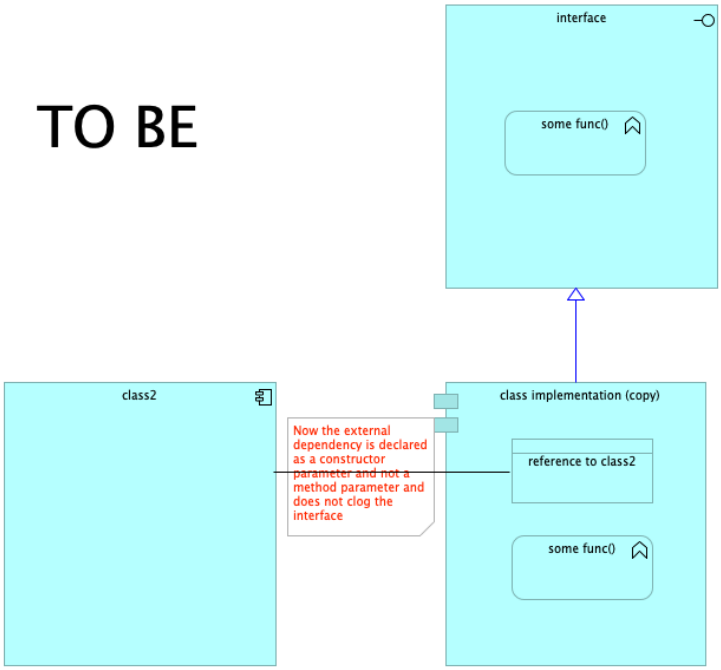

CQS Command/Query Separation
command
- only input args
- empty return result
- side effects/change state
Query
- return results
- not side effects / no chanfge state

WRAP LOW-LEVEL EXCEPTIONS

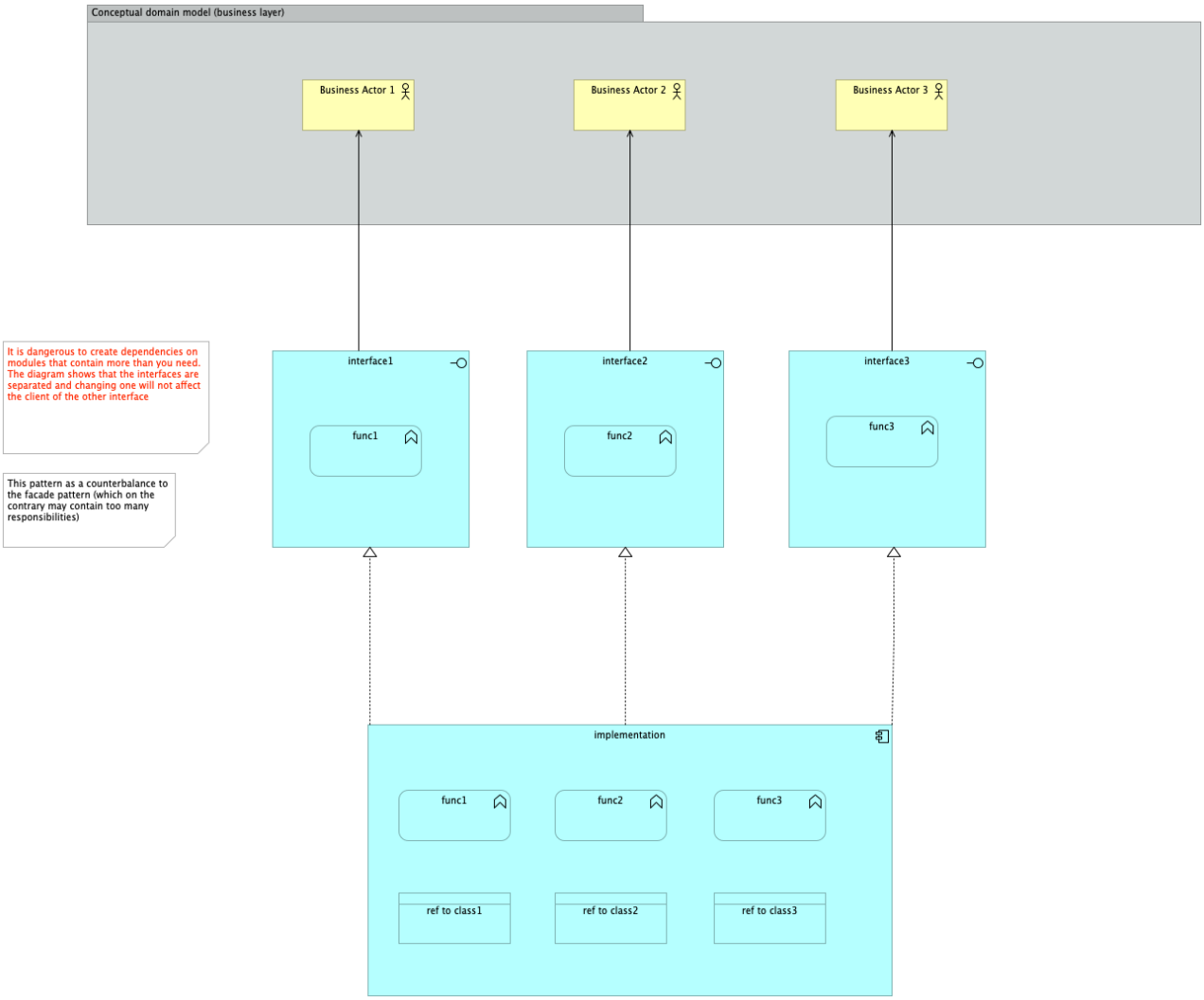


EXTRACT DEPENDENCY FROM INTERFACE TO COSNTRUCTOR

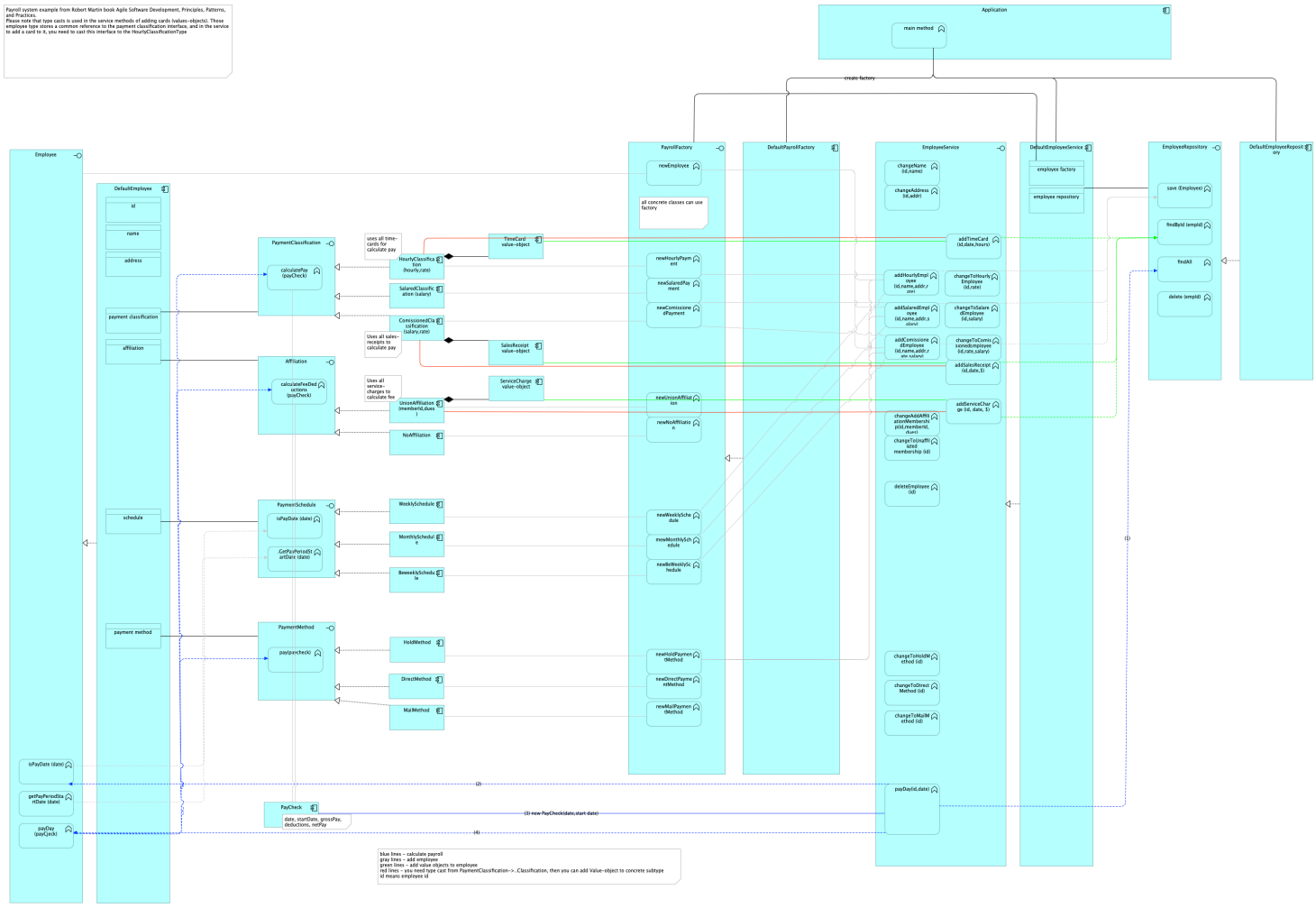




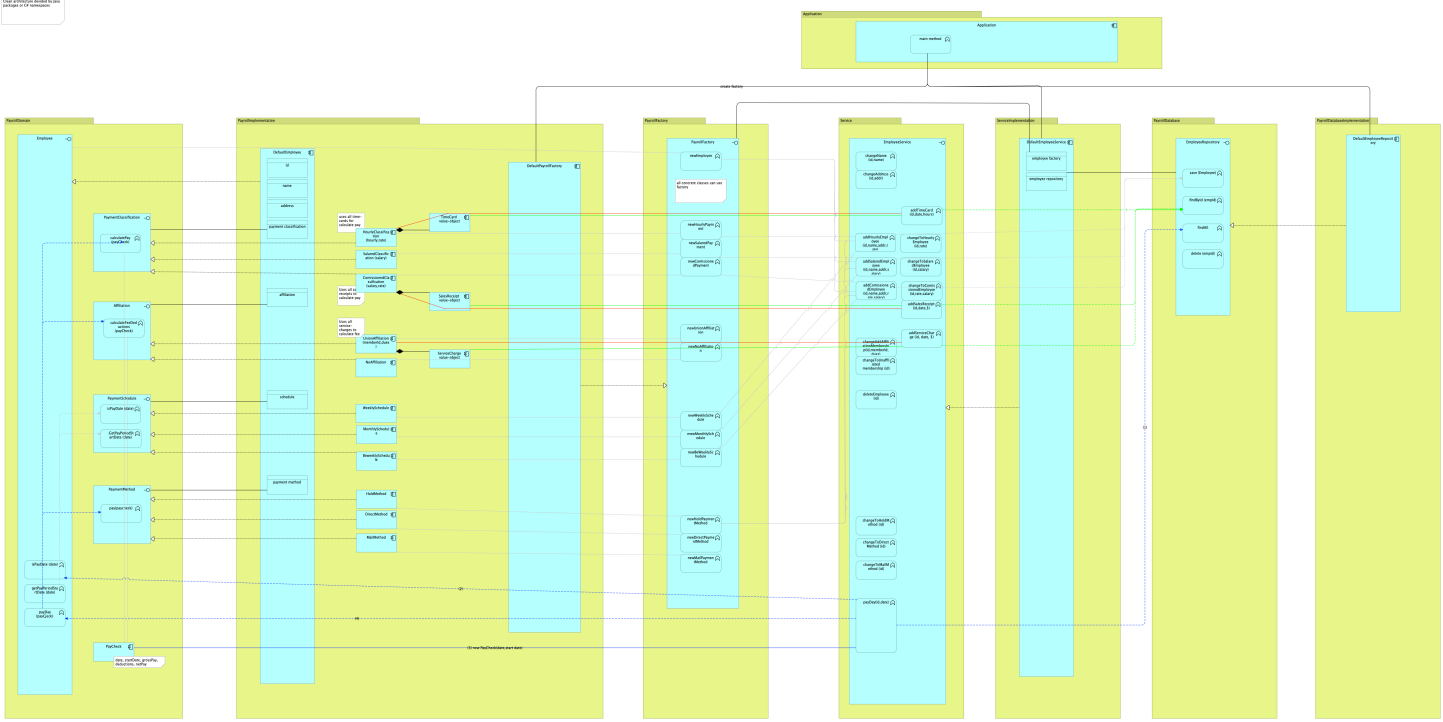
INTERFACE SEGREGATION



CLEAN ARCHITECTURE



Clean architecture divided by into packages in C# namespaces



SOME ARCHITECTURAL DIAGRAMS ARE CONSTRUCTED USING JARCHITECT, COURTESY OF CODEGEARS / CPPDEPEND

COMPLETE CATALOG OF ALL CLASSICAL PATTERNS IN THE ARCHIMATE LANGUAGE (ARCHITOOL USED) THE VERSION INCLUDES ALL 155+ PATTERNS COMPLETED (278+ MODELS). IT'S GREAT OPPORTUNITY TO USE BEST PRACTICES IN YOUR MICRO SERVICE ARCHITECTURE (ALSO AVIALABLE AT HTTPS://GITHUB.COM/WILMERKRISP/BIAN)

KrispWilmer

CC BY 4.0 LICENSE 2019

https://github.com/wilmerkrisp/patterns