Before?

BEYOND DESIGN PATTERNS & PRINCIPLES

WRITING GOOD OO CODE

ibuildings
WEB & MOBILE APP DEVELOPMENT

Matthias Noback
@matthiasnoback

My goals

- To give you some vocabulary for code discussions and reviews
- To bring back our focus to the basic ideas behind OOP

Abstract factory

Mediator

Proxy

Builder

Composite

Chain of responsibility

Adapter

Strategy

Command

Factory method

Facade

Observer

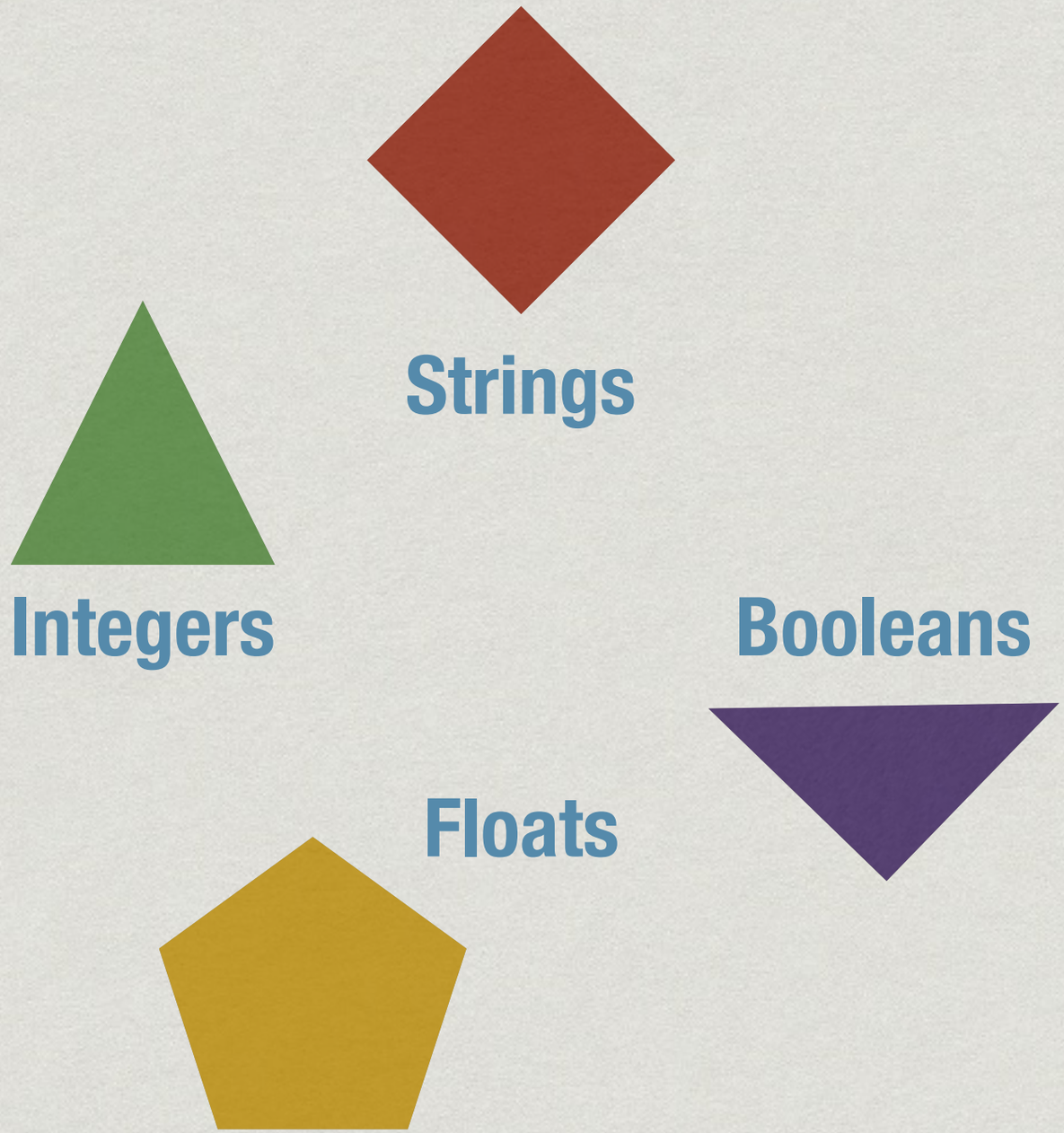
Bridge

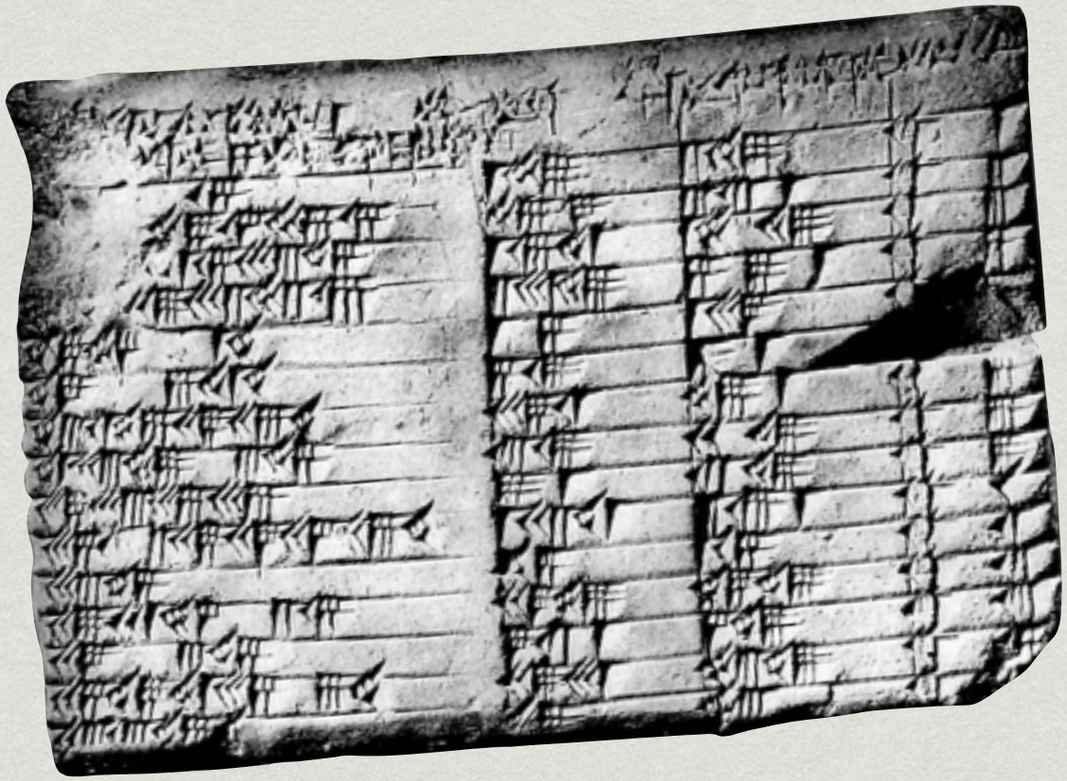
Singleton

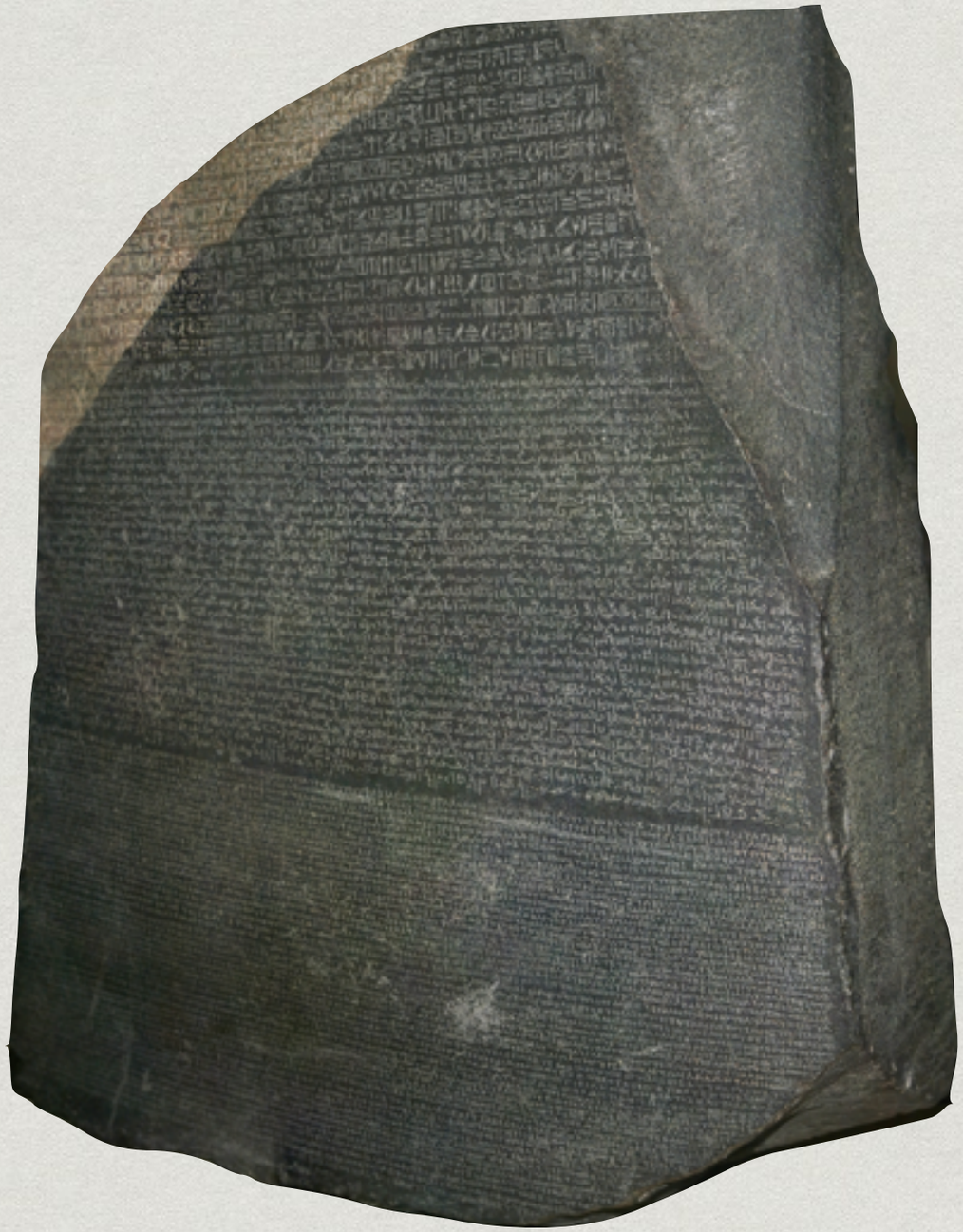
Single responsibility principle
Open/closed principle
Liskov substitution principle
Interface segregation principle
Dependency inversion principle

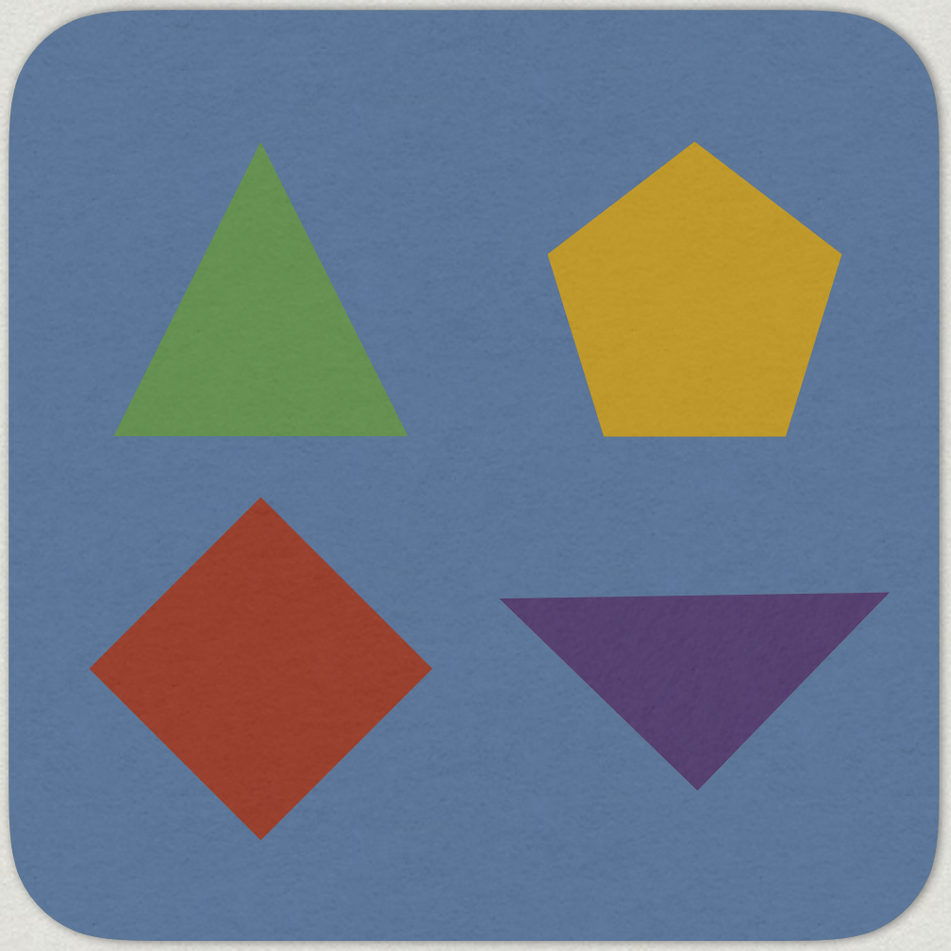
EVERYTHING IS AN OBJECT

Alan Kay


Strings
Integers
Booleans
Floats

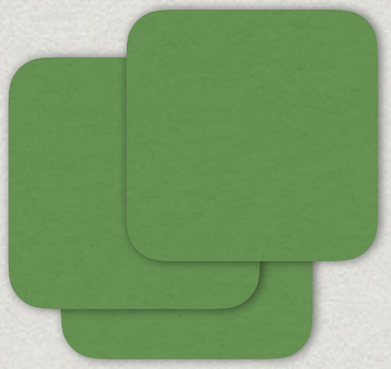






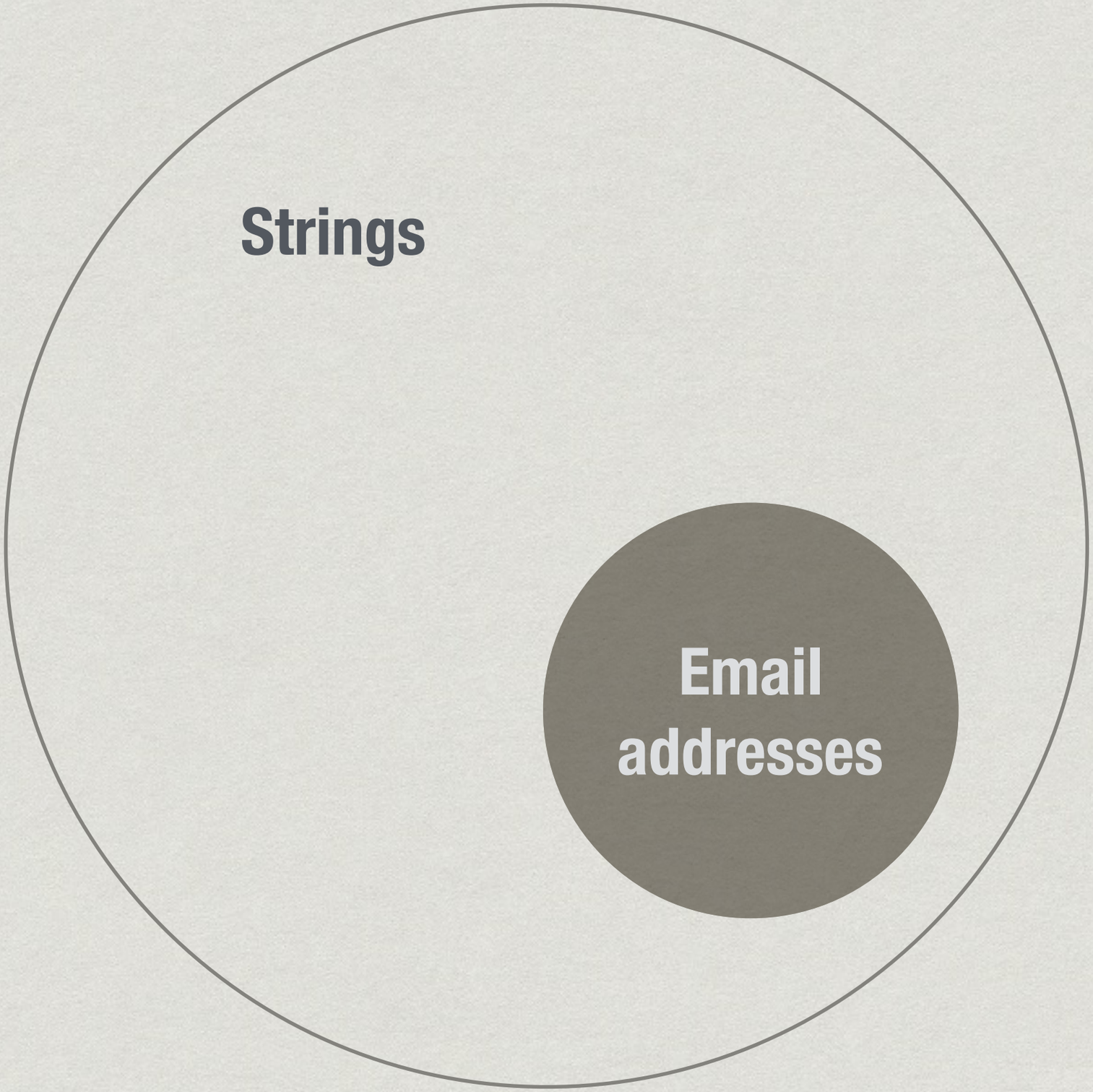
Domain concepts

Infra-stuff



Use cases

Objects introduce meaning

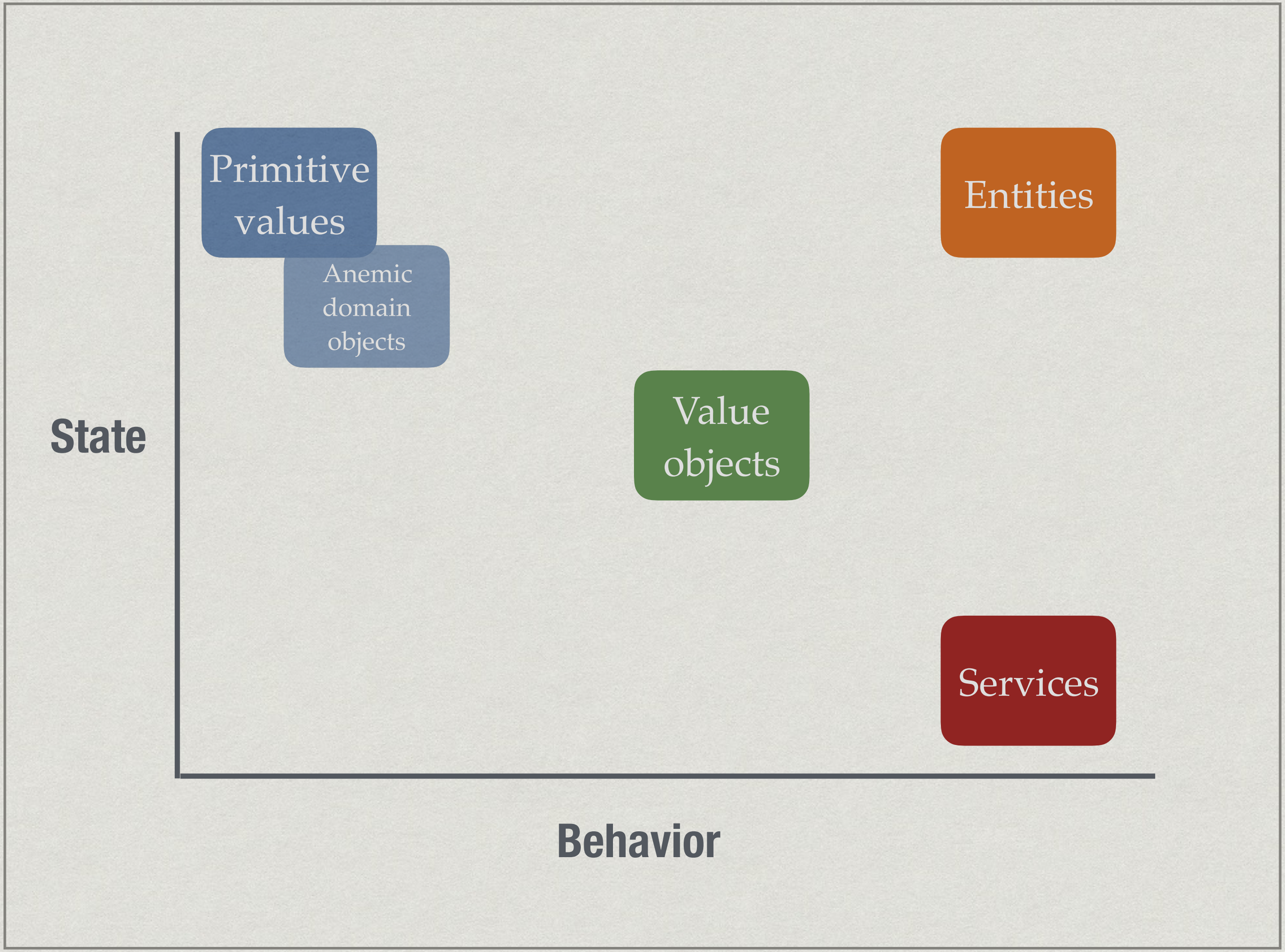

Strings
Email addresses

enclose

OBJECTS ENCAPSULATE STATE AND BEHAVIOR

```
a = ...
b = ...
```

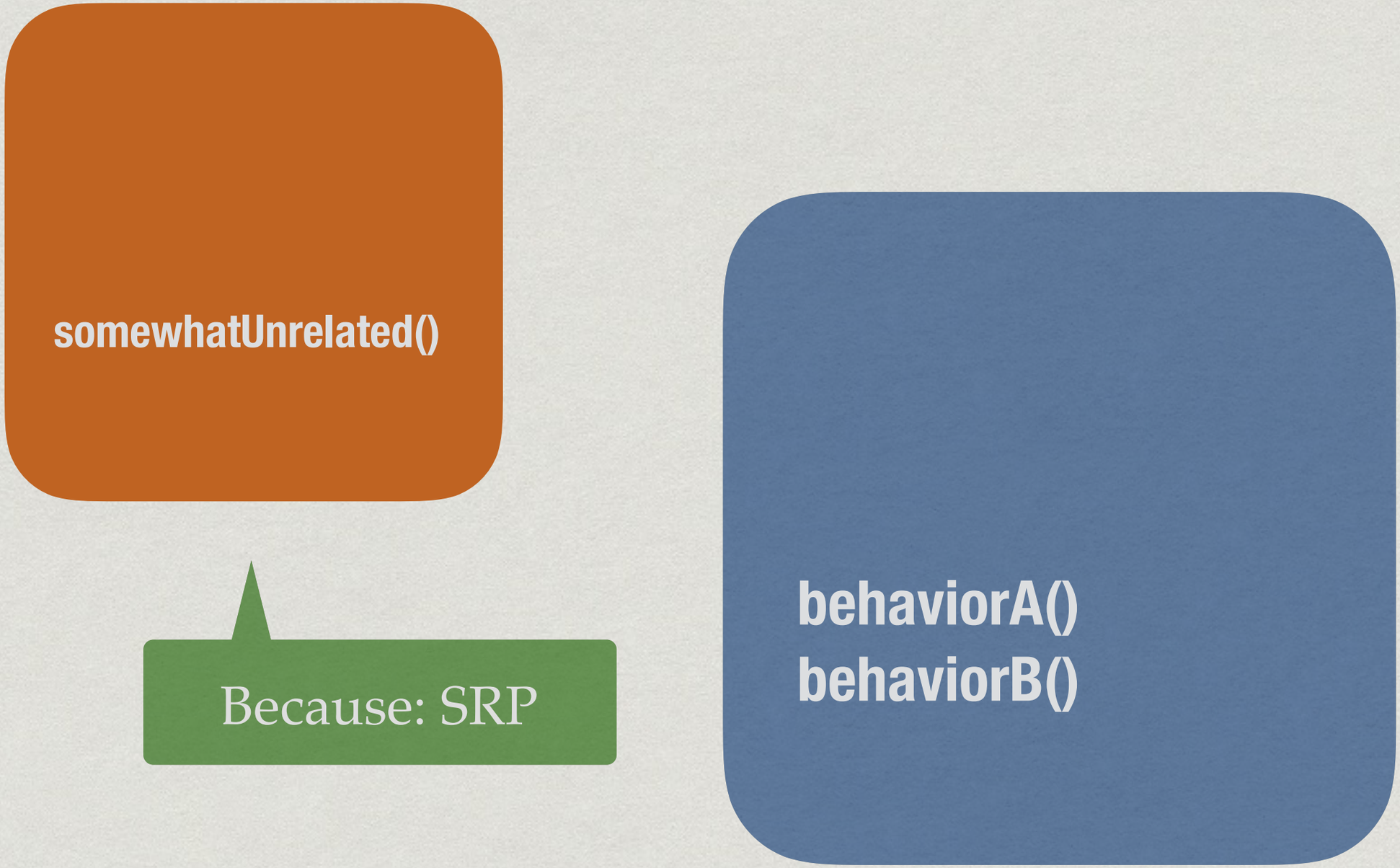
behaviorA()
behaviorB()

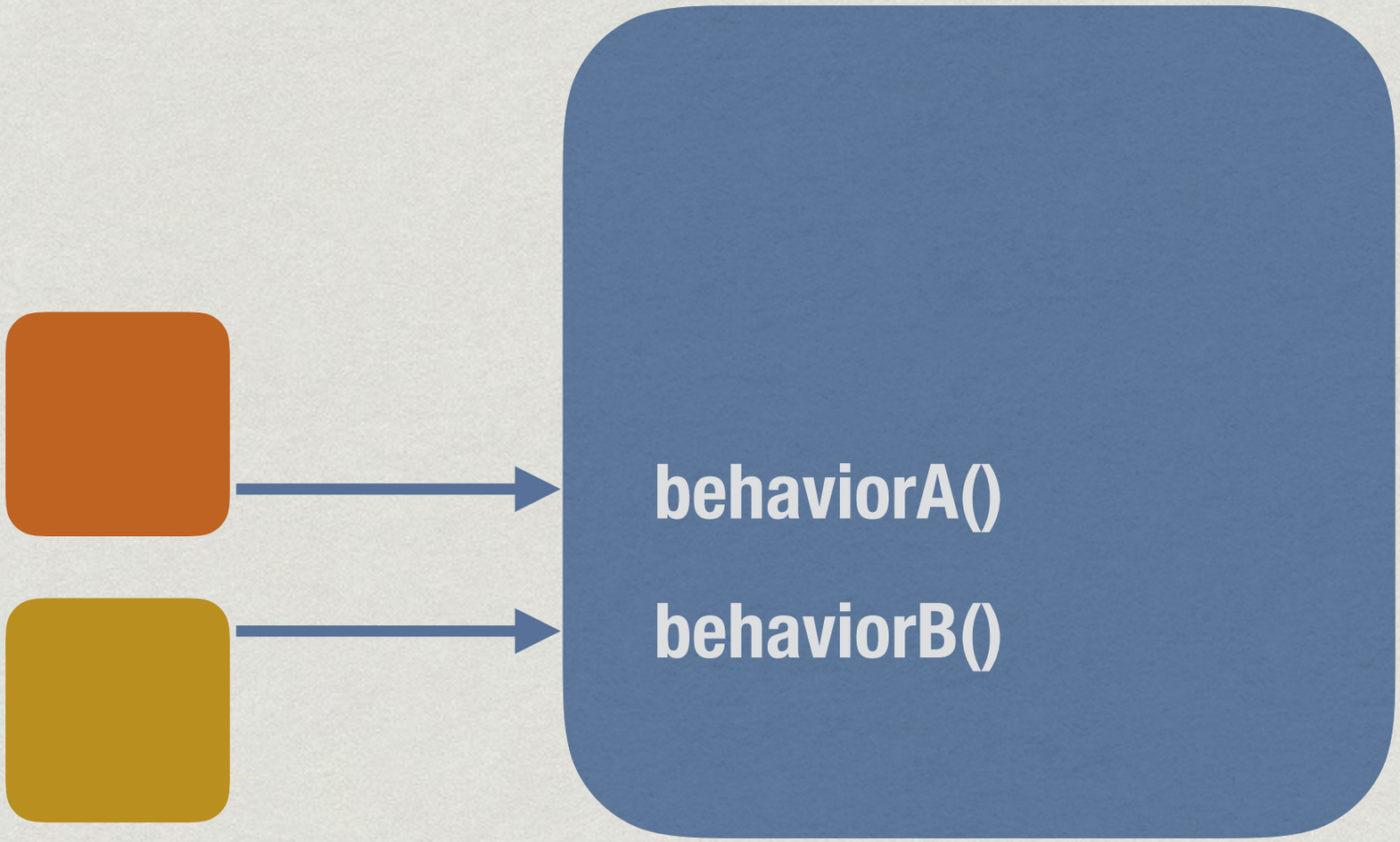

Primitive values
Anemic domain objects
Entities
Value objects
Services
State
Behavior

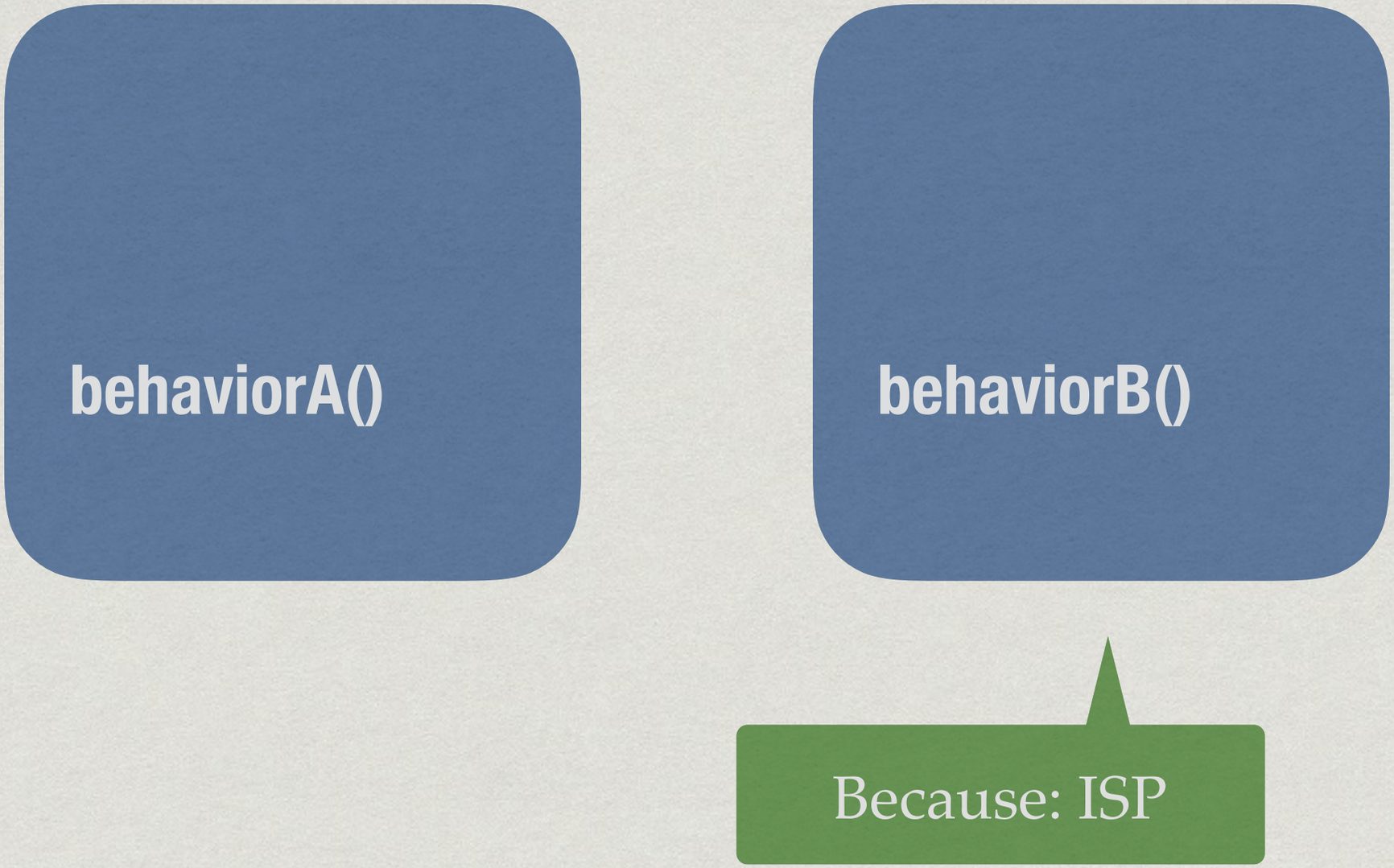
Find a balance

SERVICES SHOULD BE MODELLED AS FUNCTIONS

behaviorA()
behaviorB()
somewhatUnrelated()


somewhatUnrelated()
Because: SRP
behaviorA()
behaviorB()


behaviorA()
behaviorB()


behaviorA()
behaviorB()
Because: ISP

Single-method classes

~=

Functions

SOLID: the next step is Functional by Mark Seemann

If you take the SOLID principles to their extremes, you arrive at something that makes Functional Programming look quite attractive.

http://blog.ploeh.dk/...

Functions

Naturally stateless

```
class TheClass
{
    public doSomething(...)
    {
        check pre-conditions
        happy path
        handle failure
        check post-conditions
    }
}
```

Simpler design, easier to test

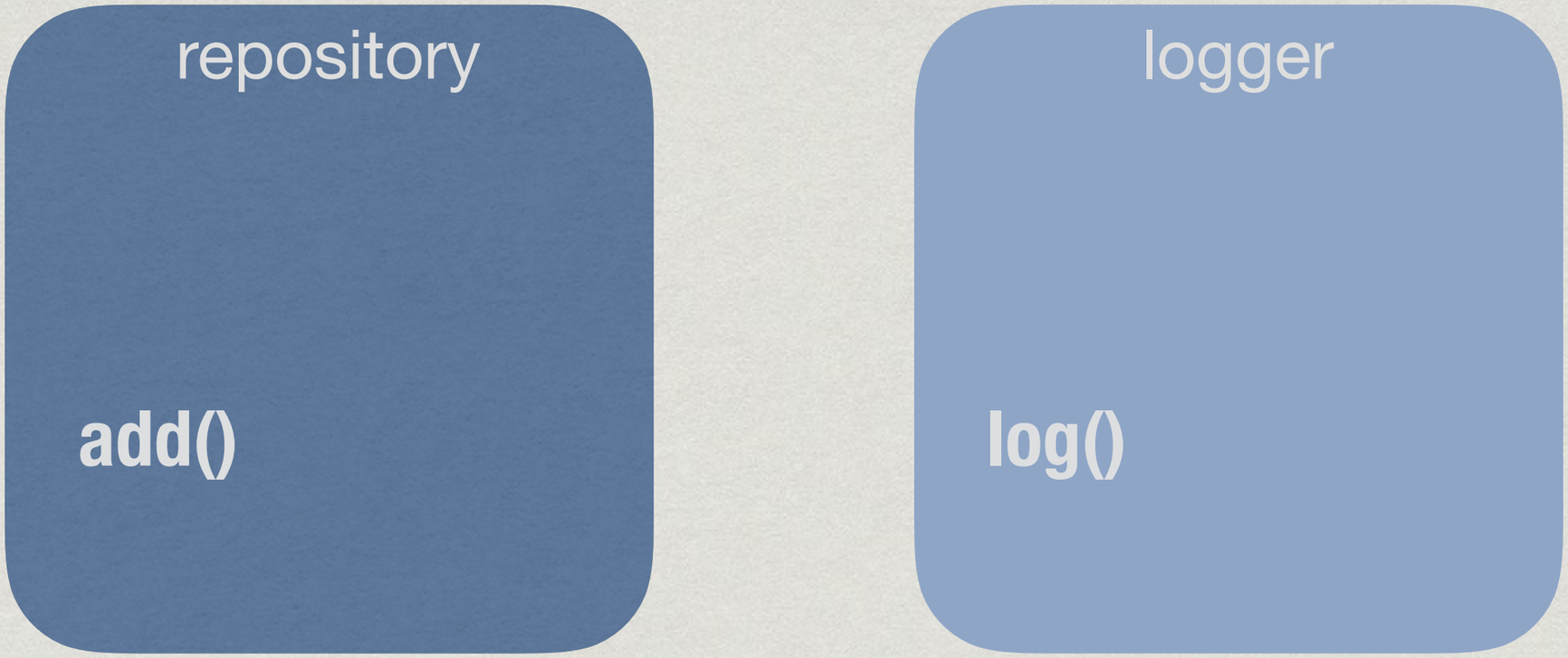
- **The unit is smaller**
- **Small number of possible execution paths**
- **No unexpected changes in behavior**

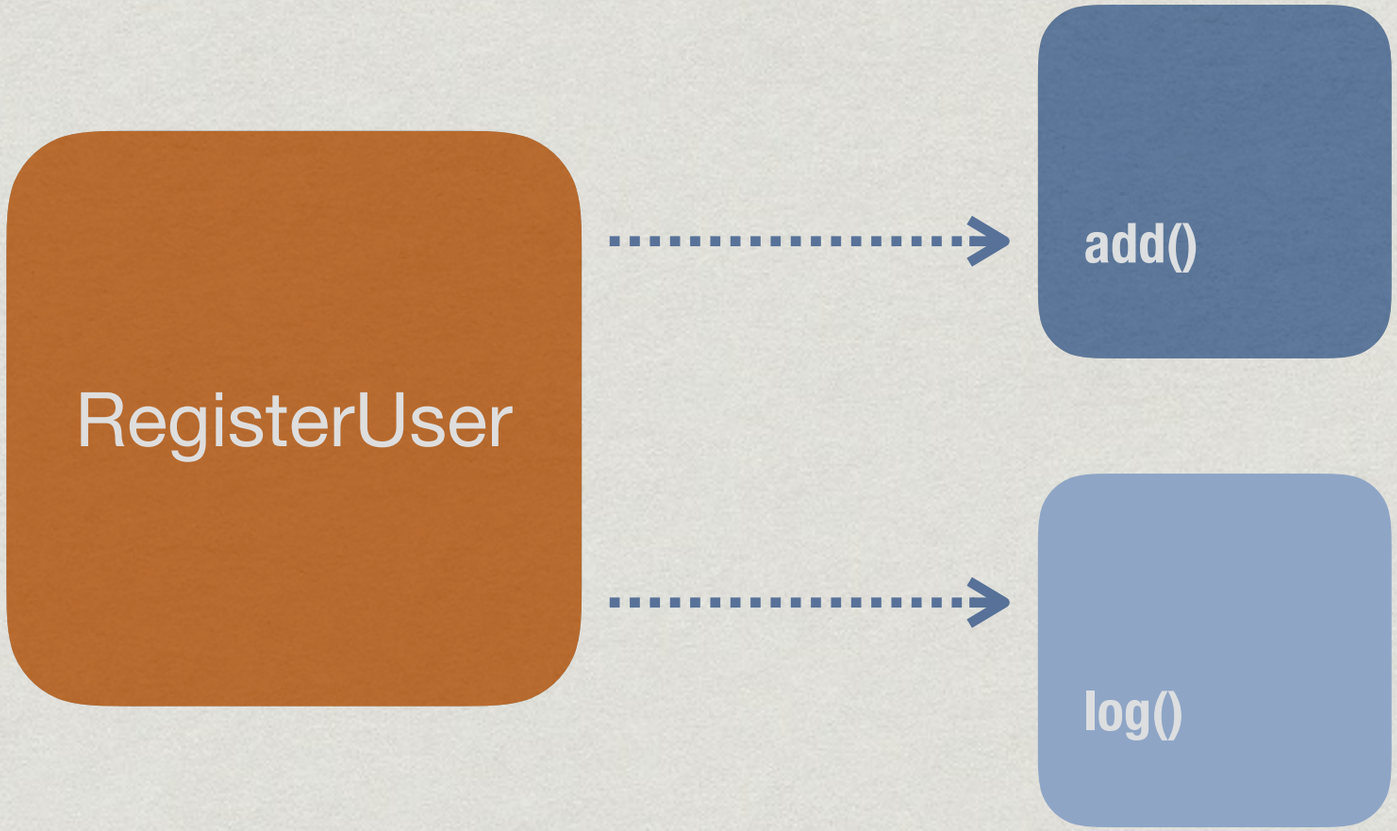
OBJECTS SHOULD BE EXPLICIT ABOUT SIDE-EFFECTS

```
class RegisterUser
{
    private repository
    private logger

    public void register(name) {
        repository.add(new User(name));
        logger.log("New user added");
    }
}
```

```
function registerUser(name, repository, logger)
{
    repository.add(new User(name));
    logger.log("New user added");
}
```


repository
add()
logger
log()


RegisterUser
add()
log()

INJECTED SERVICES SHOULD BE SINGLE-METHOD OBJECTS TOO

```
function registerUserFactory(repository, logger)
{
    return function(name) {
        repository.add(new User(name));
        logger.log("New user added");
    }
}

registerUser = registerUserFactory(
    repository,
    logger
);

registerUser("Matthias");
```

Inject this object instead!

```
serviceLocator.get('service id').method()
```

No service locators

Inject this object instead!

```
someService.getOtherService.doSomething()
```

No train wrecks

Inject only single-method services

Side-effects will be explicit

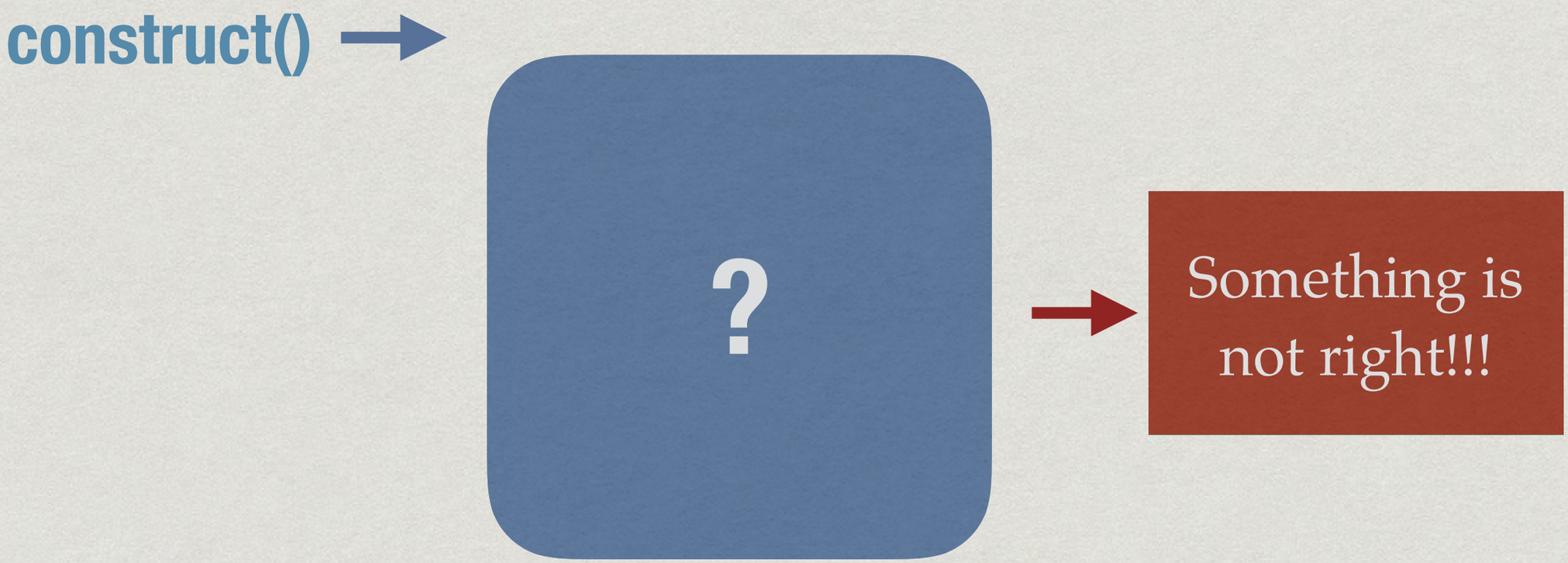
Responsibilities will be more explicit

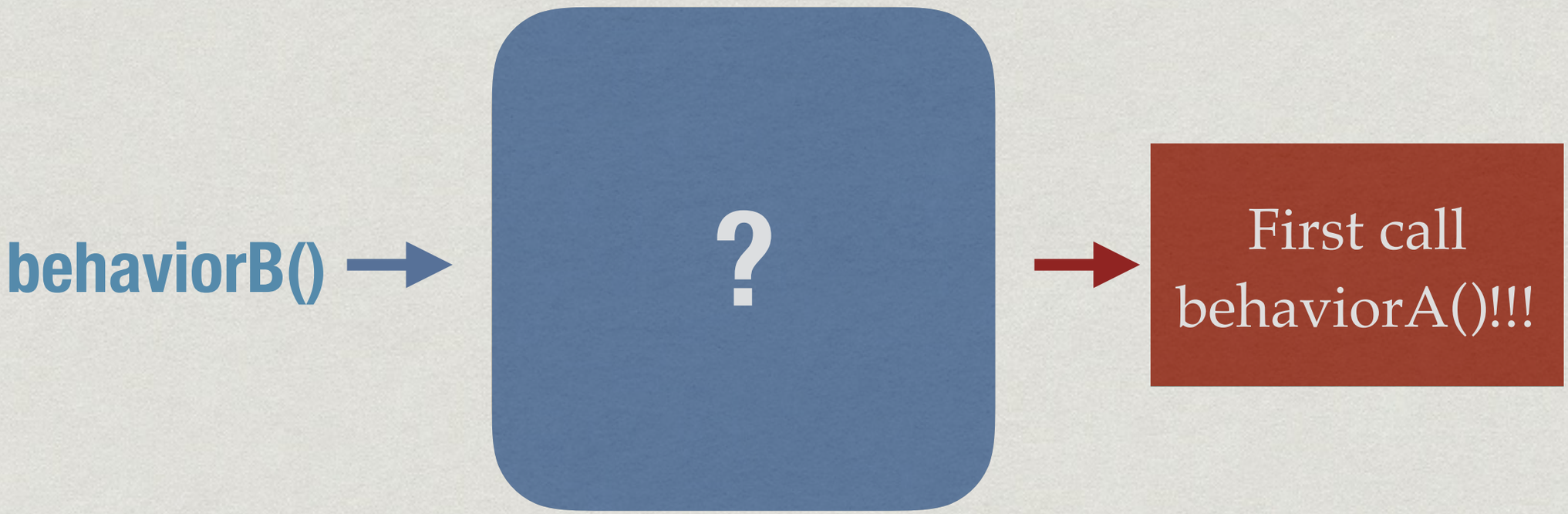
Dependencies will be easier to mock

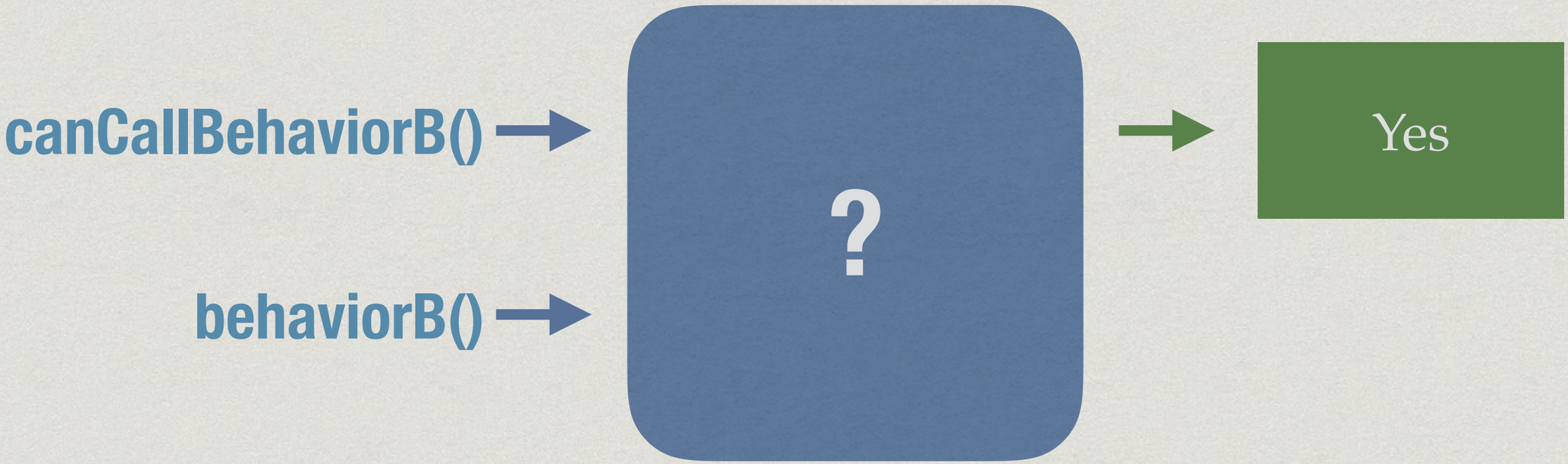
OBJECTS SHOULD ONLY EXIST IN A VALID STATE

condition

?

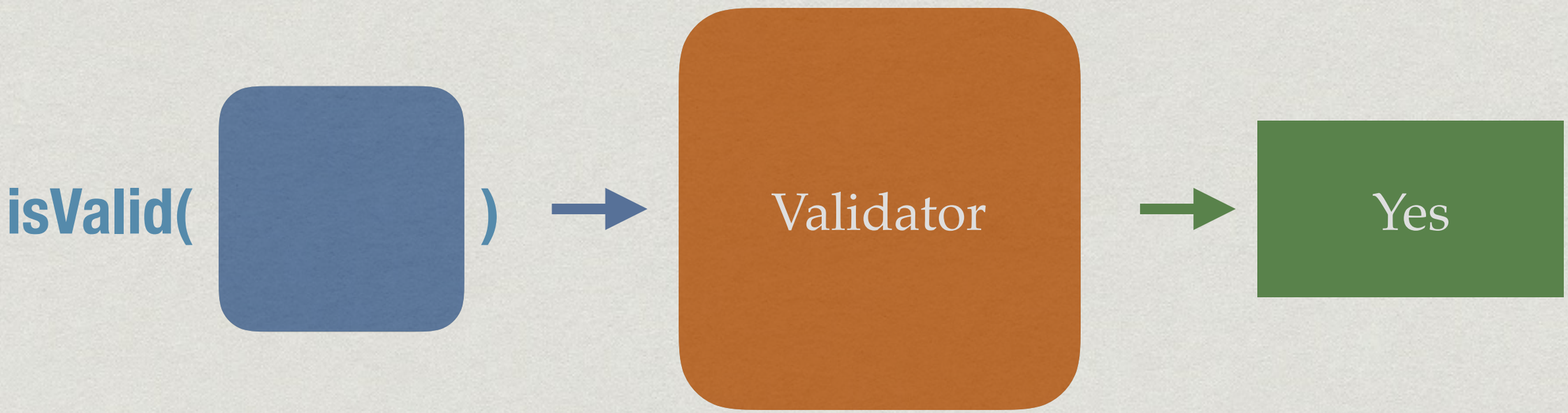

construct()
?
Something is not right!!!


behaviorB()
?
First call
behaviorA()!!!

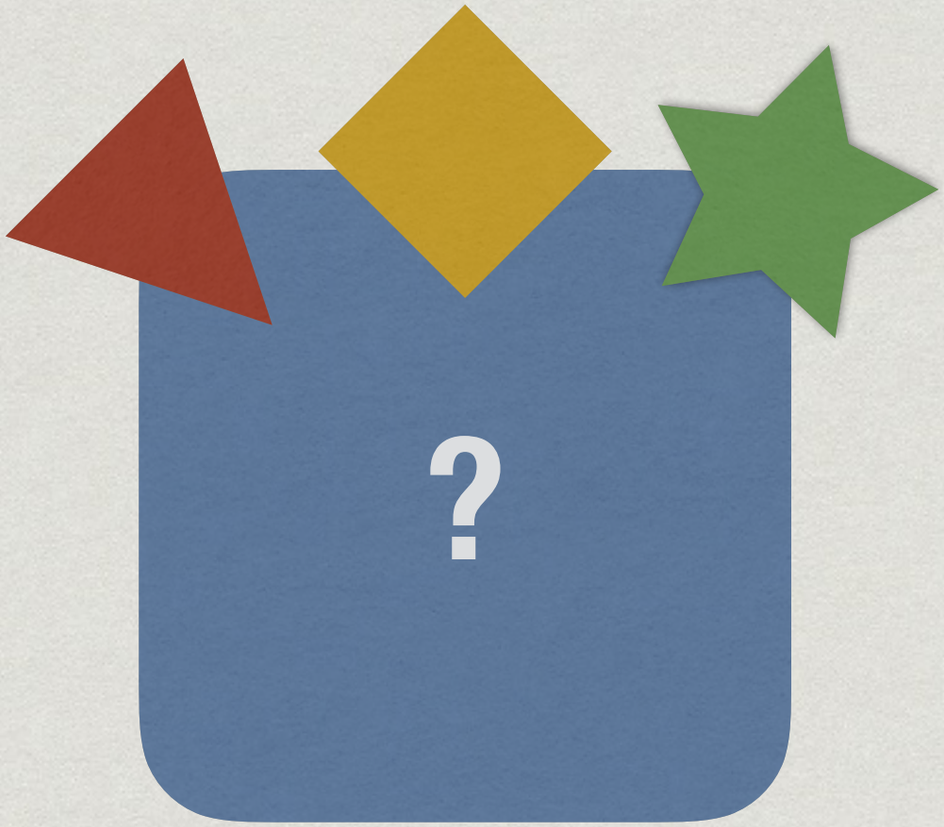


Anemic domain model

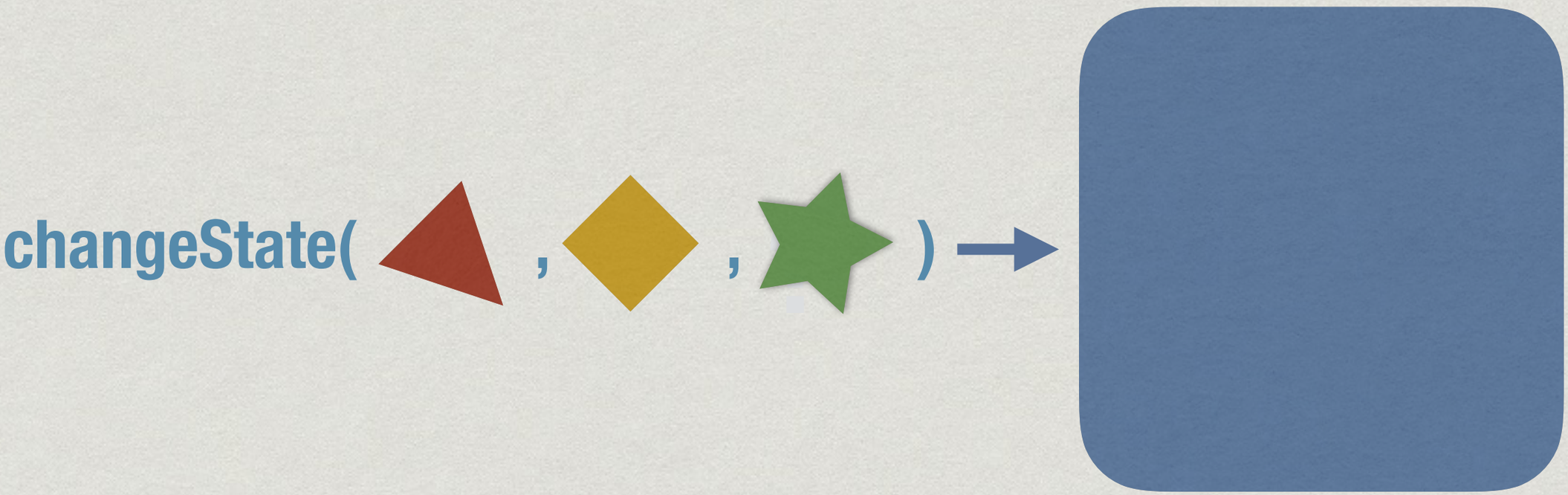
Tell, don't ask


isValid( )
Validator
Yes

How to prevent this?

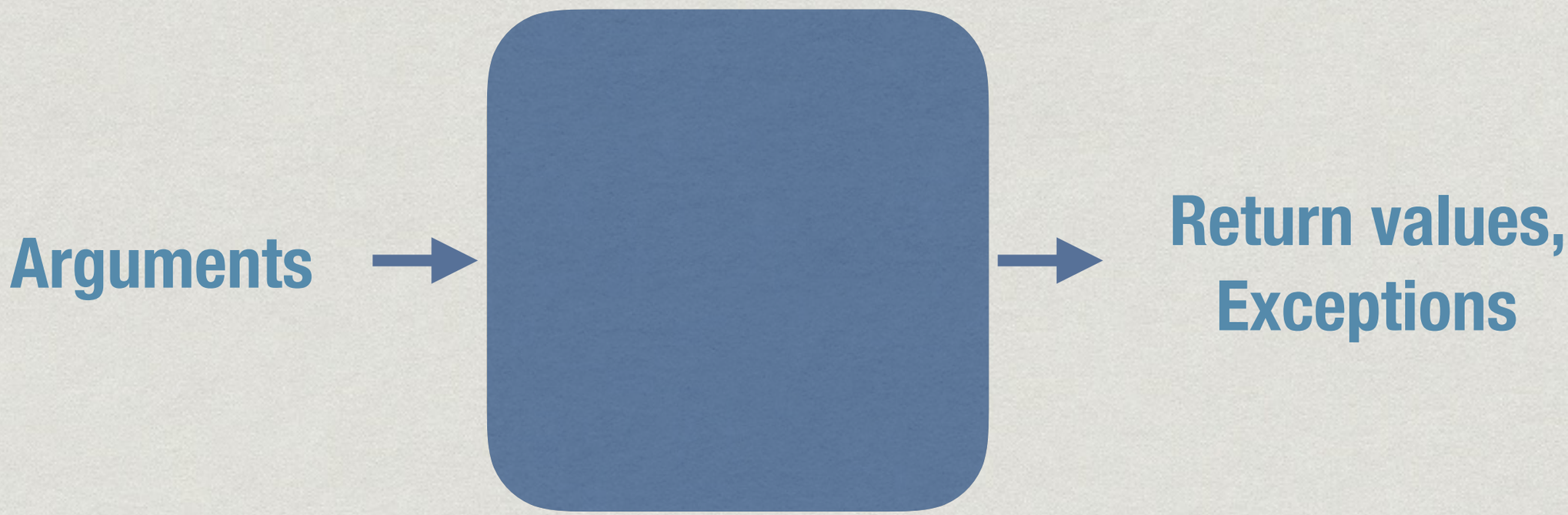


What about…

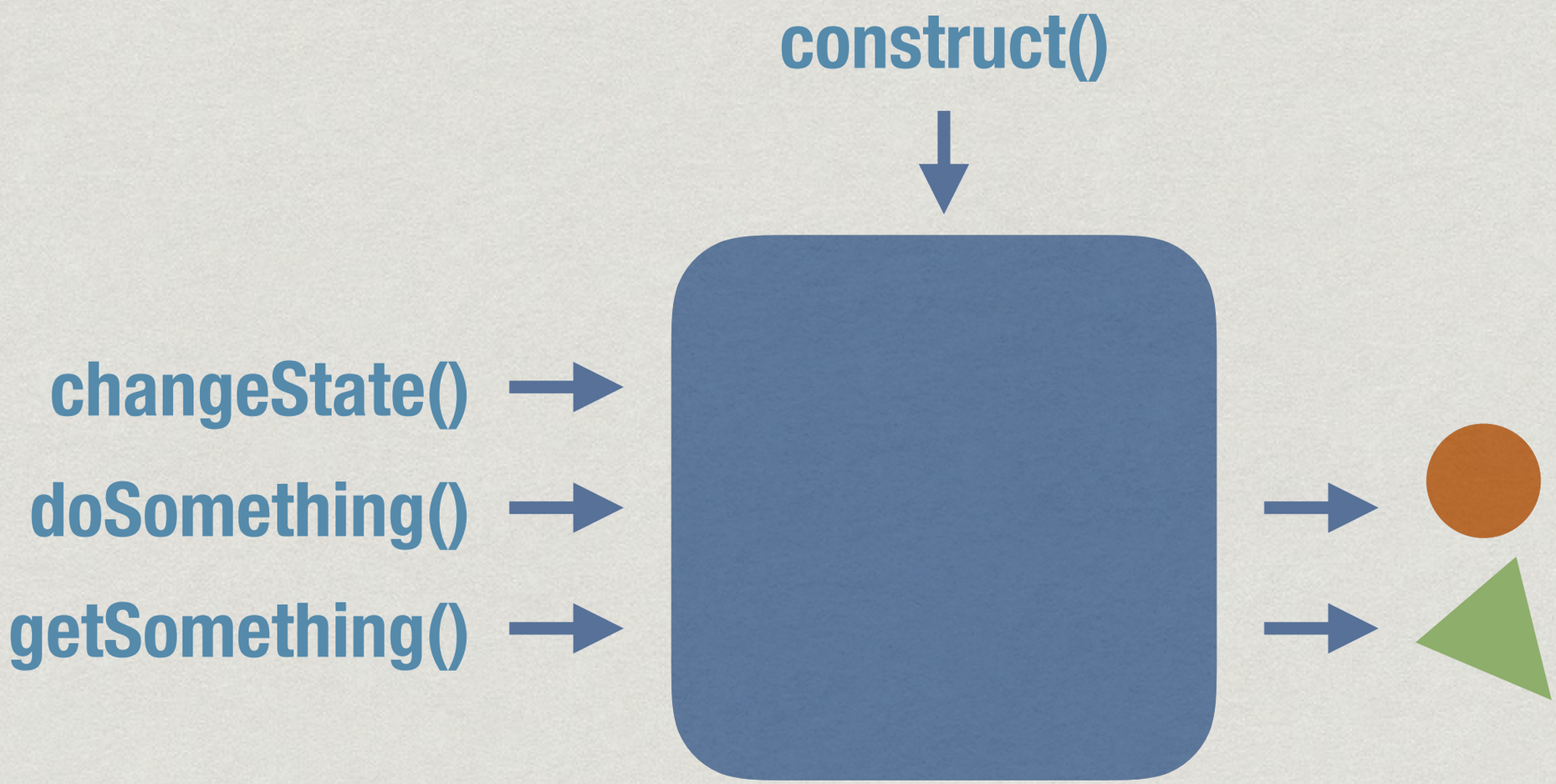


ONLY VALID VALUES SHOULD CROSS OBJECT BOUNDARIES

Crossing boundaries



```
changeState(...) {
    check pre-conditions
    ...
    check post-conditions
}
```


construct()
changeState()
doSomething()
getSomething()

Advantages of always-valid objects

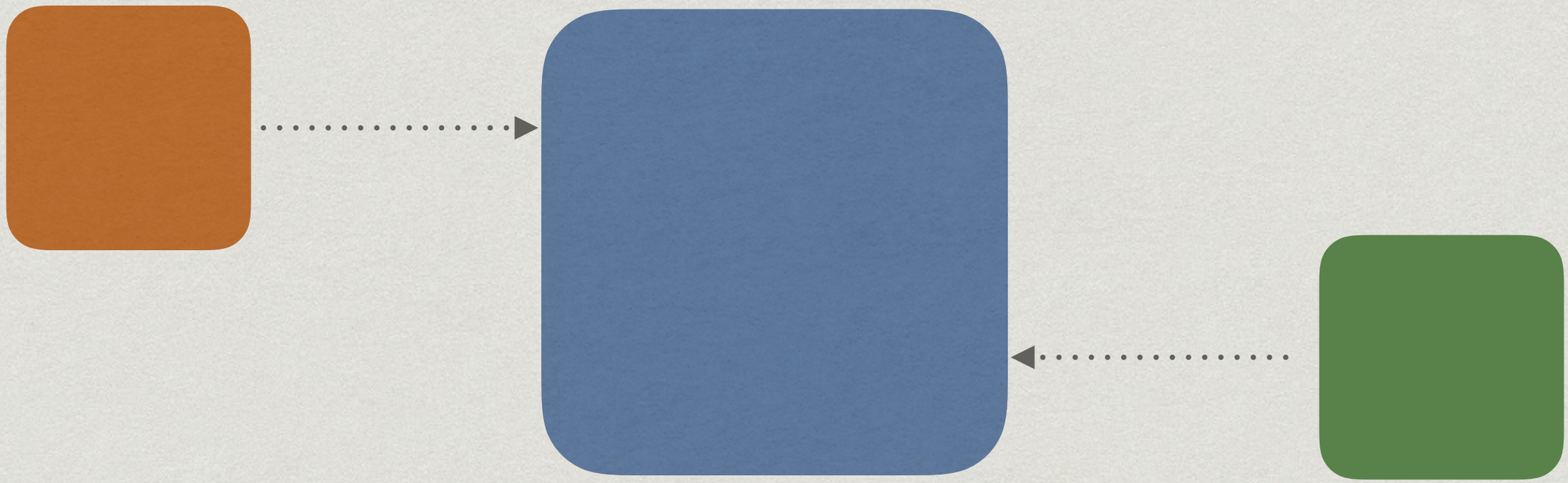
Easy to reason about

Easy to debug

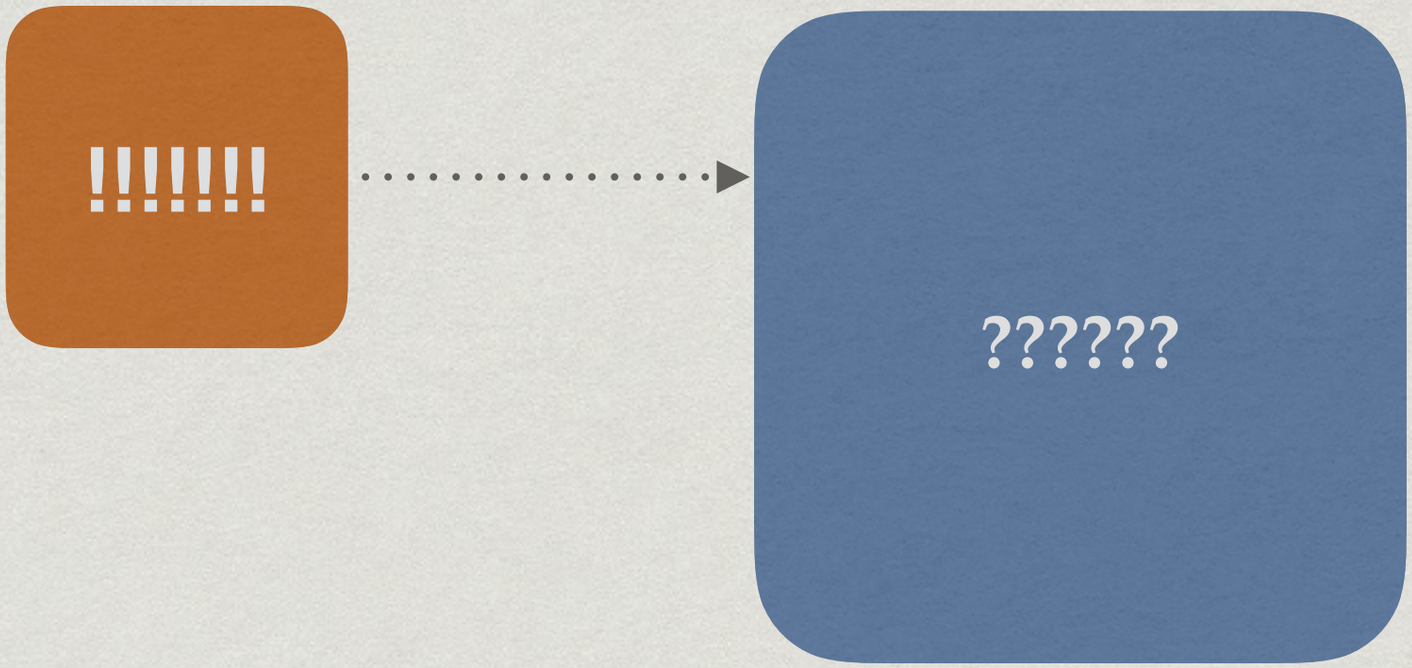
Easy to refactor

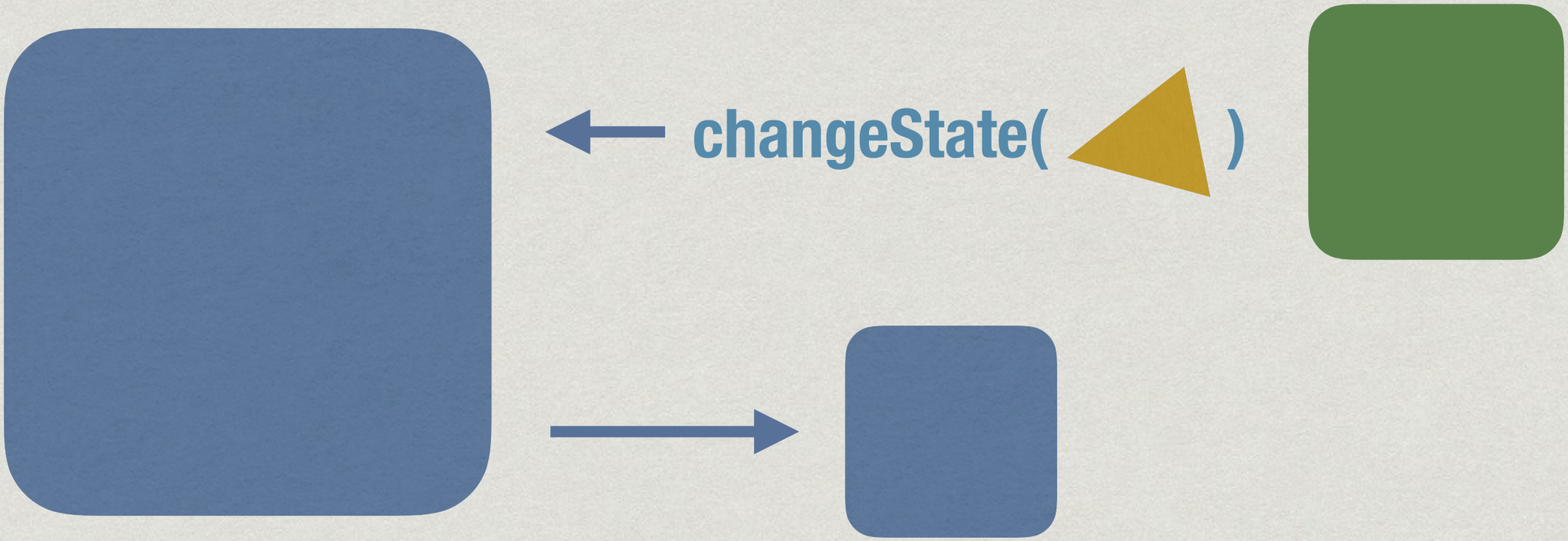
ALMOST ALL OBJECTS SHOULD BE IMMUTABLE

unable to be changed




changeState( )




changeState( )

Advantages of immutability

Clear distinction between *using* **state and** *changing* **state**

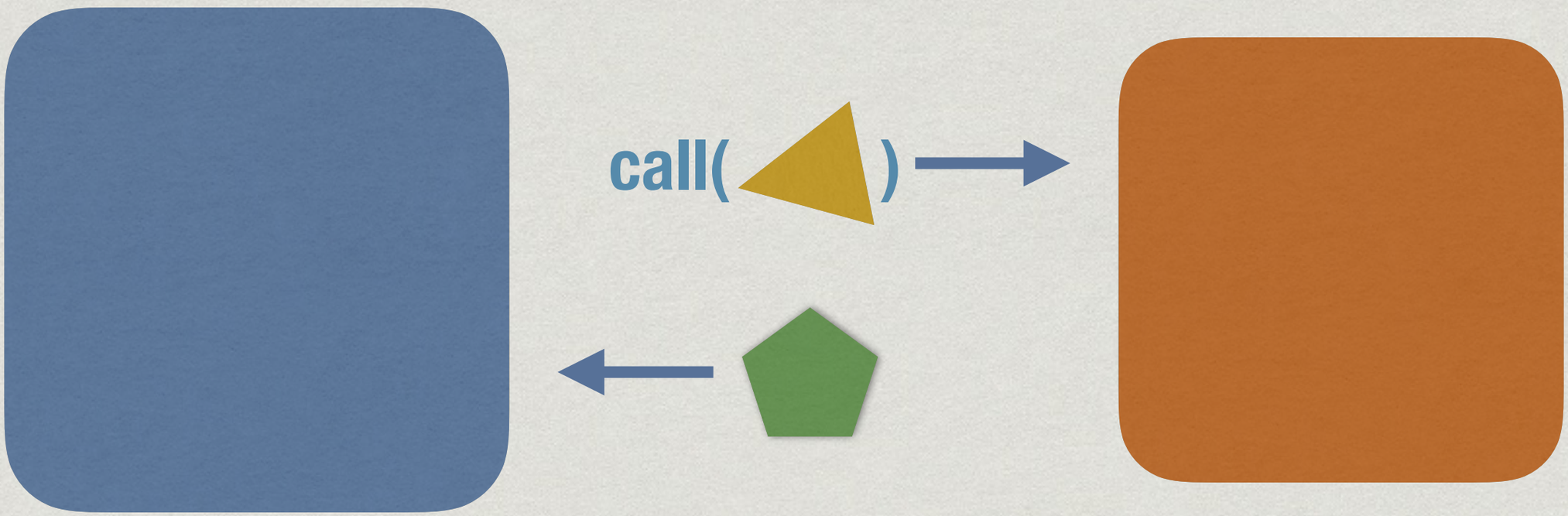
Pass on objects without worrying

Use objects without worrying

Easier to reason about

OBJECTS SHOULD COMMUNICATE USING WELL-DEFINED MESSAGES

"Alan Kay"


call( )

Calling methods

==

Sending messages

Receiving return values

==

Receiving messages

Catching exceptions

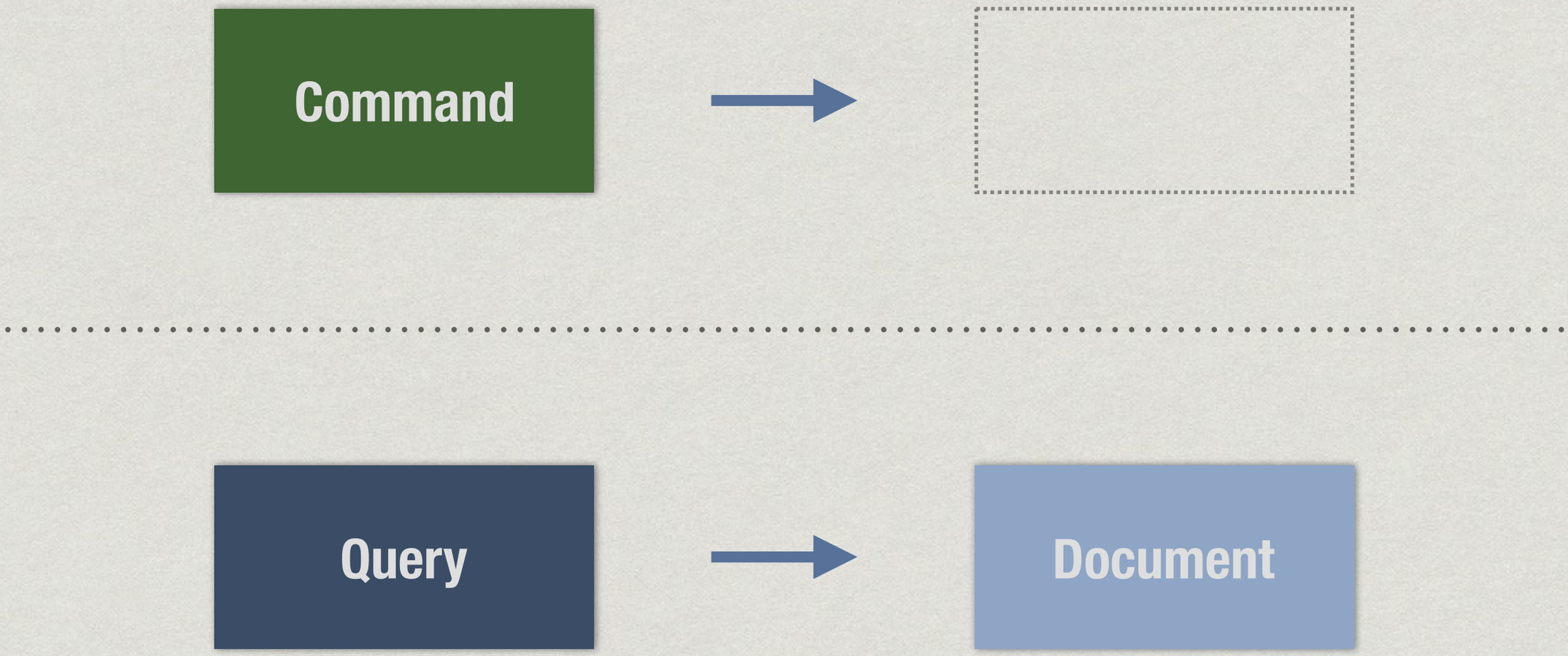
==

Receiving messages

Command

Query

Document


Command
Query
Document

```
public void doSomething(...) {
    ...
}

public [return-type] getSomething(...) {
    return ...
}
```

CQS: command-query separation

Asking for information doesn’t change observable state

EVERYTHING IS AN OBJECT

YOUR APPLICATION TOO!

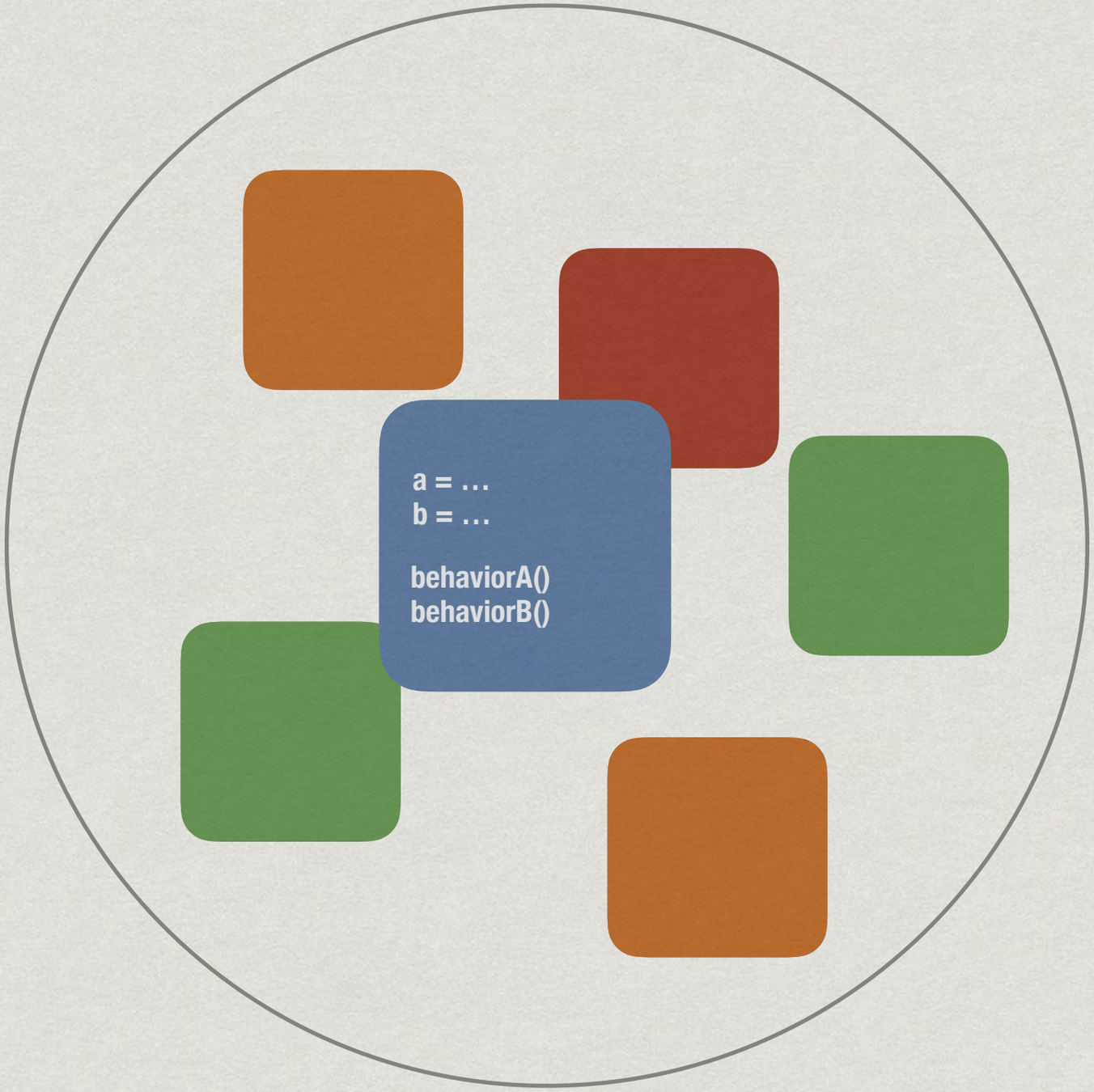
Sort of…

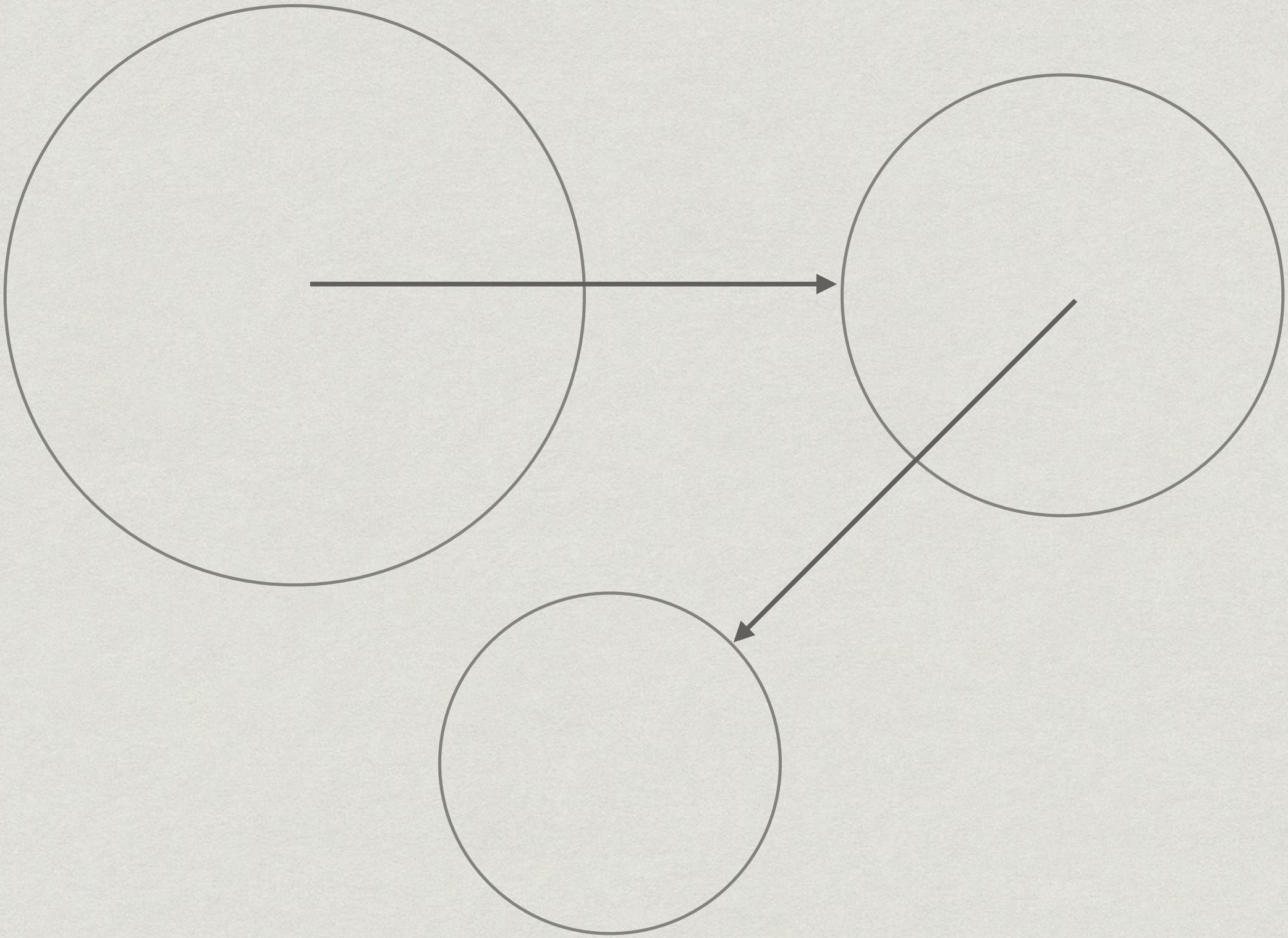
“I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages […]”

Alan Kay

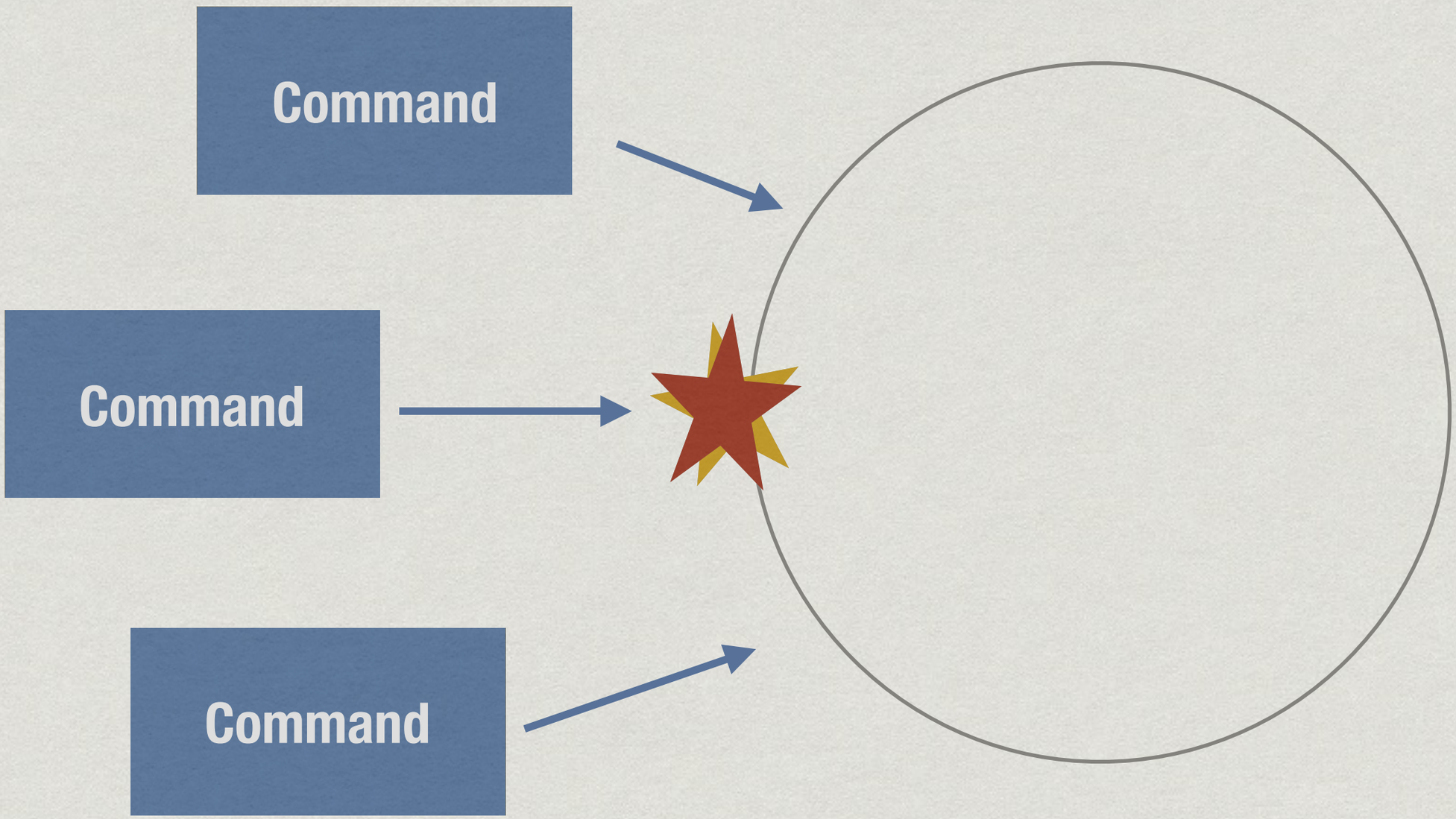
```
a = ...
b = ...

behaviorA()
behaviorB()
```

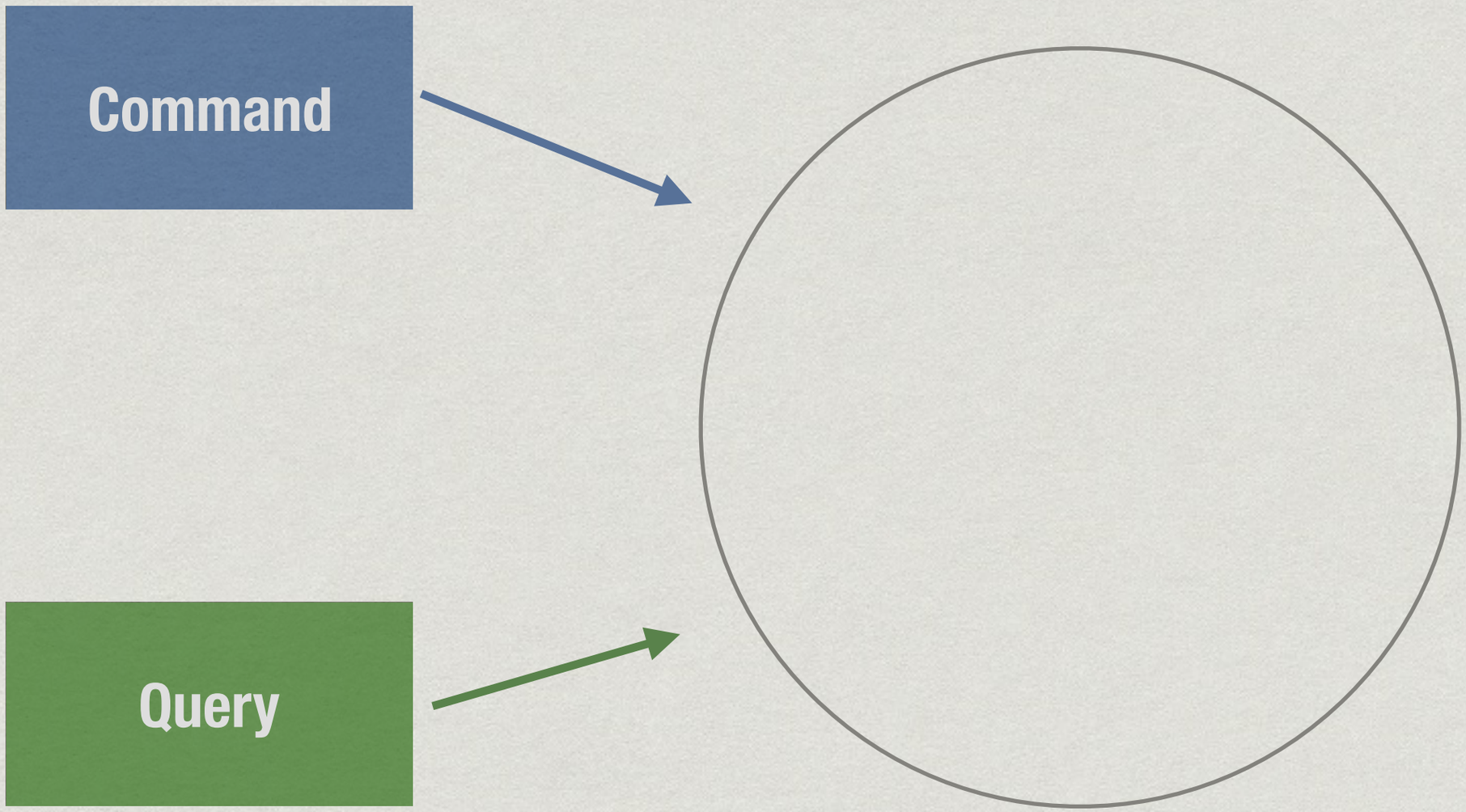

a = ...
b = ...
behaviorA()
behaviorB()



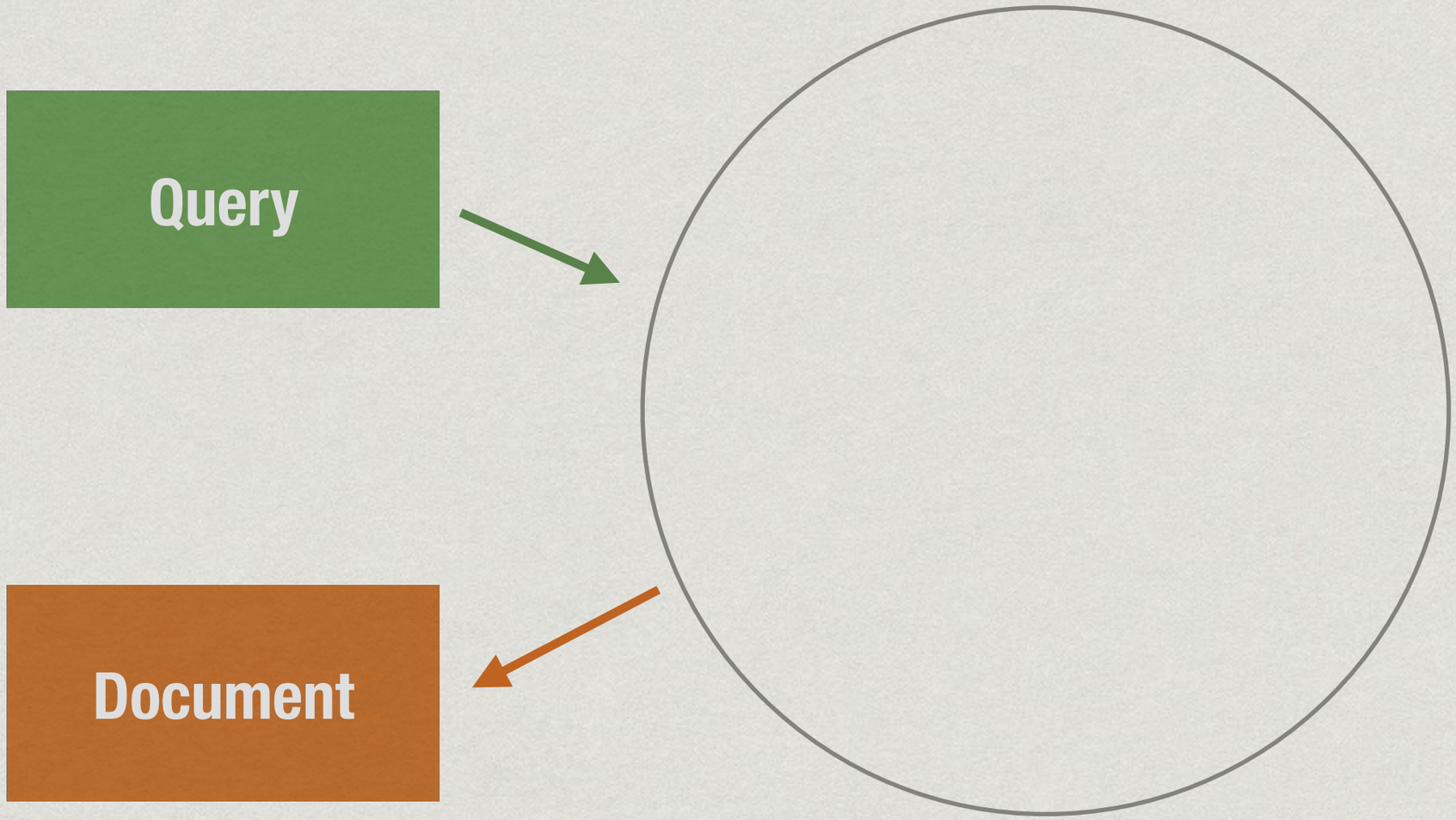
State should always be valid



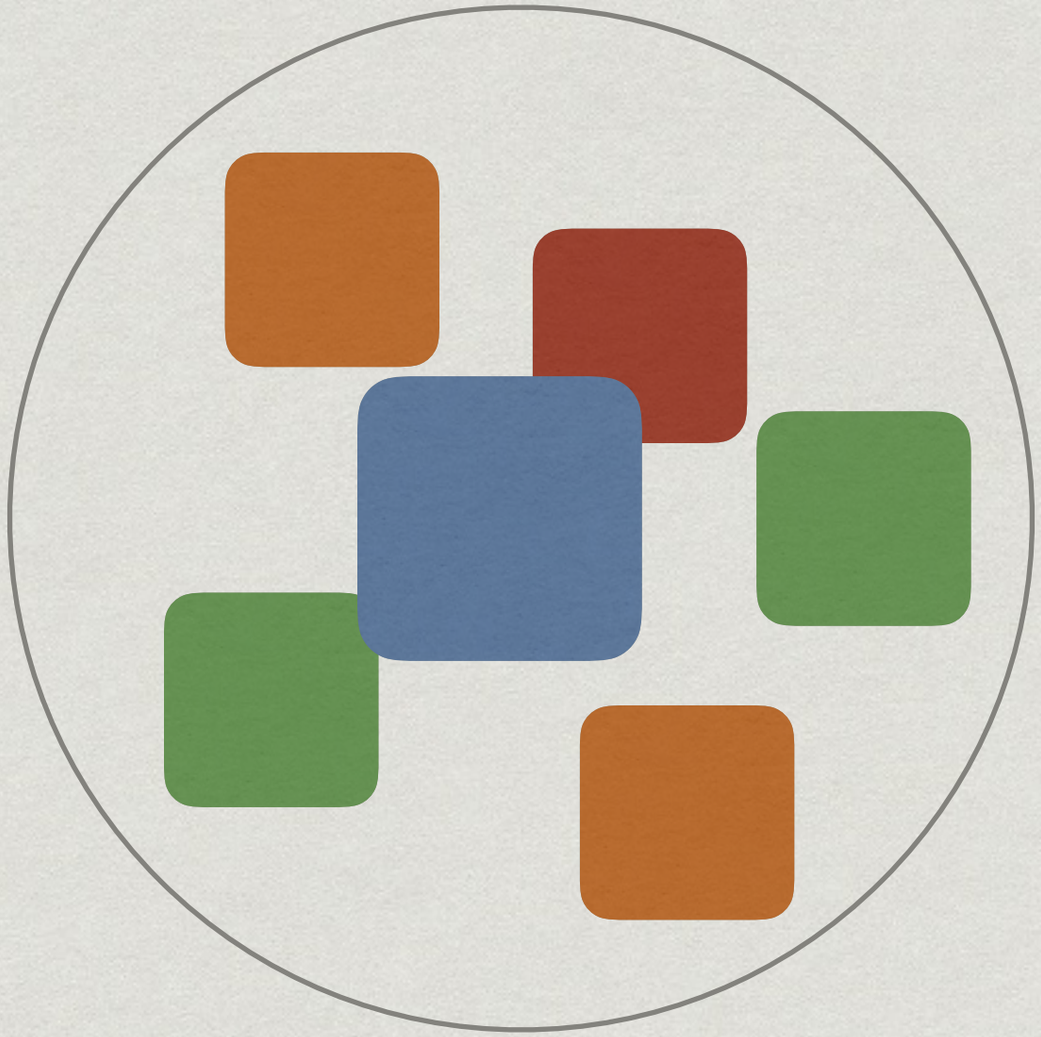
Apply CQS



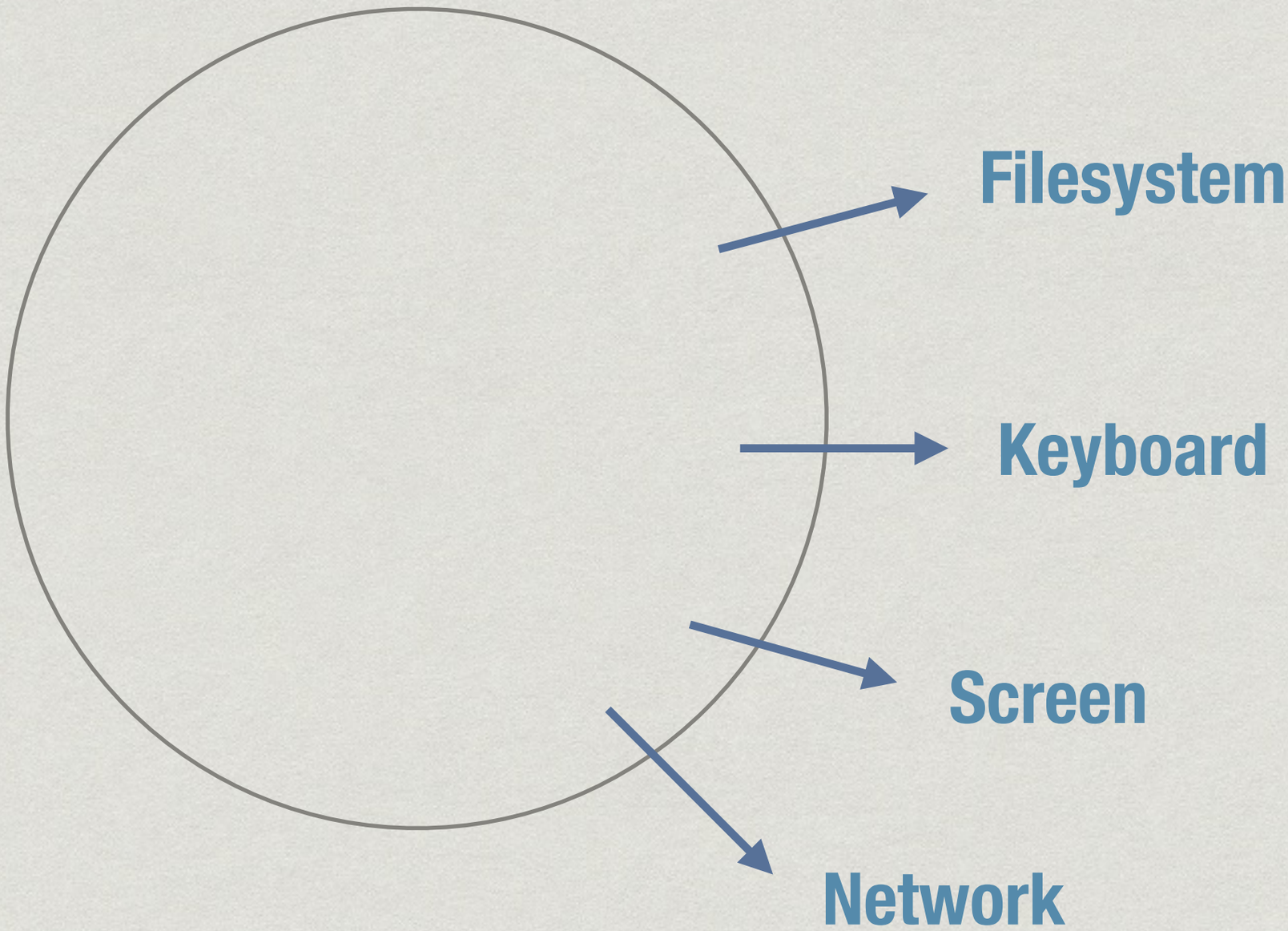
Prefer immutability



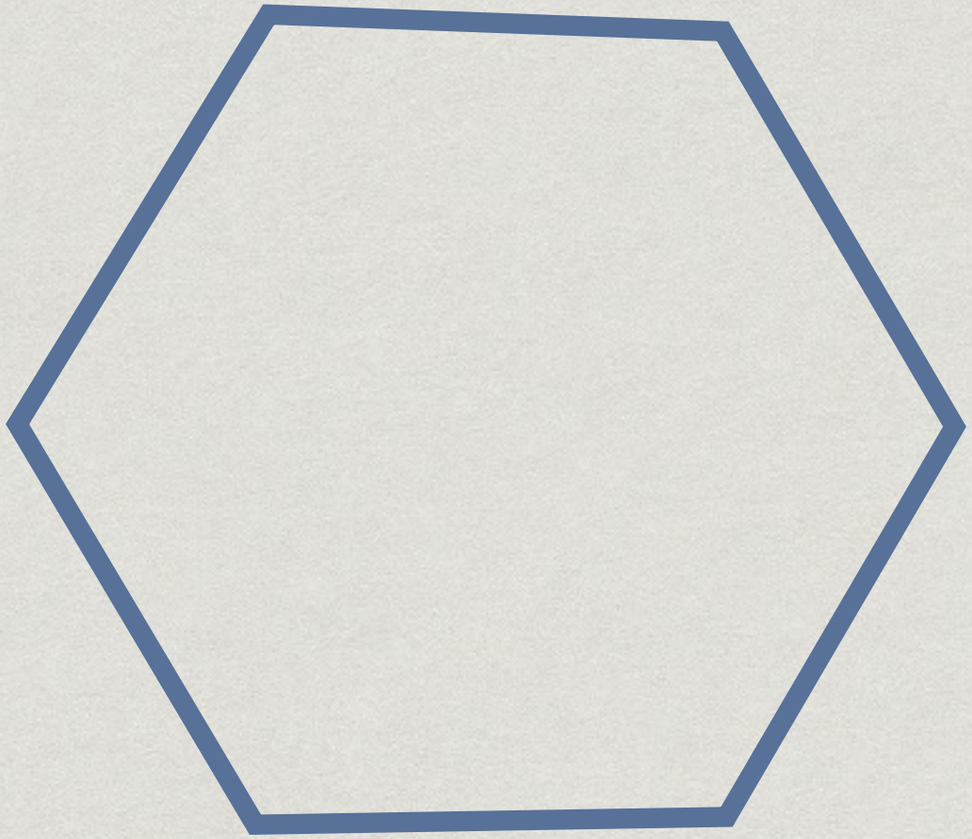
This design emerges



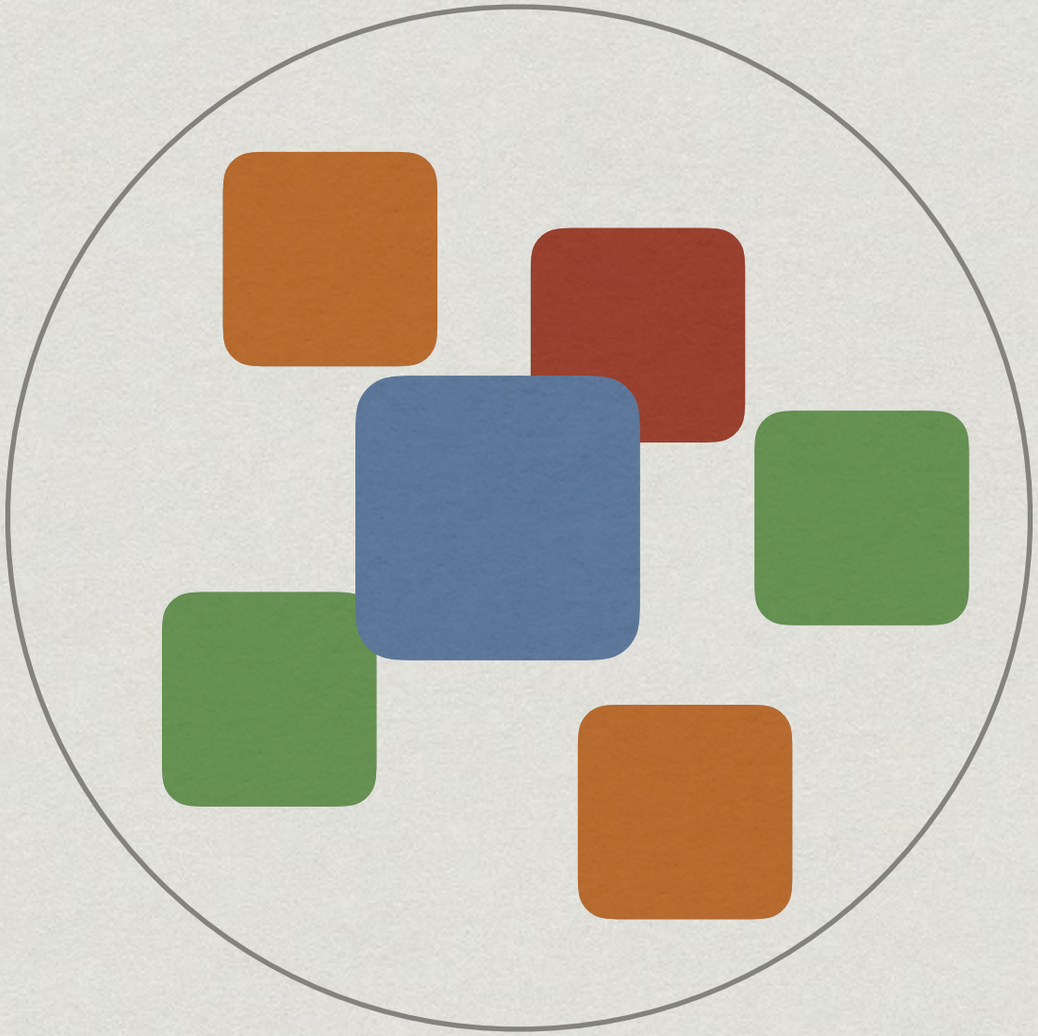
Your application has side-effects

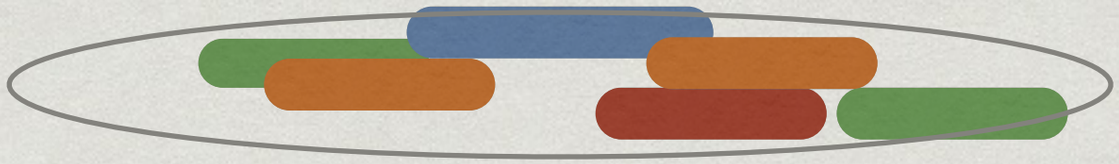


Apply hexagonal architecture



IN WHICH WAYS IS YOUR APPLICATION NOT AN OBJECT?





???

Some suggestions

- **Write the serialization code yourself**
- **The object's design always comes first**
- **Don't let tools mess with object encapsulation**

Also…

Try Event Sourcing

CONCLUSION

EVERYTHING IS AN OBJECT

OBJECTS ENCAPSULATE STATE AND BEHAVIOR

SERVICES SHOULD BE MODELLED AS FUNCTIONS

OBJECTS SHOULD BE EXPLICIT ABOUT SIDE-EFFECTS

OBJECTS SHOULD ONLY EXIST IN A VALID STATE

ONLY VALID VALUES SHOULD CROSS OBJECT BOUNDARIES

OBJECTS SHOULD COMMUNICATE USING WELL-DEFINED MESSAGES

ALMOST ALL OBJECTS SHOULD BE IMMUTABLE

TREAT YOUR APPLICATION AS AN OBJECT (AND VICE VERSA)