ЛЕКЦИЯ 7
Scheme и объектно-ориентированное программирование

Объектно-ориентированное программирование

- Абстракция данных, объединяющая состояние и функции
- Механизм передачи сообщений
- Объектные модели
  - диаграмма классов
  - диаграмма объектов
- Пример: «космические войны»

Абстракции

- Процедурные абстракции
- Абстракции данных

- Общее предназначение: выражать сложные понятия через более простые, скрывая детали.

- Вопросы:
  - Как лучше всего представить систему набором абстракций?
  - Как сделать простым расширение системы?
    - Добавлять новые типы данных?
    - Добавлять новые функции?

Подход «от данных»

- Структуры данных
  - Сборка сложных структур с помощью cons:
    point, line, 2dshape, 3dshape
  - Использование заглавного звена – тега, указывающего тип данных
    (define (make-point x y) (list 'point x y))
  - Реализация структуры данных требует набора функций: конструктора, селекторов, мутаторов, …

- Обобщённые операции

```
(define (scale x factor)
      (cond ((point? x)  (point-scale x factor))
            ((line? x)   (line-scale x factor))
            ((2dshape? x)(2dshape-scale x factor))
            ((3dshape? x)(3dshape-scale x factor))
            (else (display "unknown type"))))
```

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans

Обобщённые операции

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию

	Point	**Line**	**2-dShape**	**3-dShape**
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color
new-op	…	…	…	…

Обобщённые операции

- при добавлении новой функции
  - просто добавляем новую обобщённую операцию
- при добавлении нового типа данных
  - необходимо переписать каждую обобщённую операцию

	Point	**Line**	**2-dShape**	**3-dShape**	curve
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	c-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans	c-trans
color	point-color	line-color	2dshape-color	3dshape-color	c-color
new-op	…	…	…	…	…

Два взгляда на мир

Объект данных

	Point	**Line**	**2-dShape**	**3-dShape**	curve
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	c-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans	c-trans
color	point-color	line-color	2dshape-color	3dshape-color	c-color
new-op	…	…	…	…	…

Обобщённая операция

Что такое объект данных?

- Это структура данных
  - соединённая с набором собственных операций
  - для которой есть общее понятие (line), и экземпляры (line17)
  - экземпляр хранит свои свойства (состояние)
- Мы изобрели ООП!

ООП в Scheme: Функции и состояние

- У функции есть
  - **параметры** и **тело** описанные в lambda-выражении
  - **окружение,** где хранятся связывания её имён

- Можно использовать функцию для хранения (и сокрытия) данных и предоставления доступа к ним.
- При вызове функции создаётся новое окружение.
- Нужно иметь доступ к этому окружению.
  - для доступа нужны операции (конструктор, селекторы, мутаторы, …)
  - мутаторы изменяют состояние

Пример: мутируемые пары как объекты

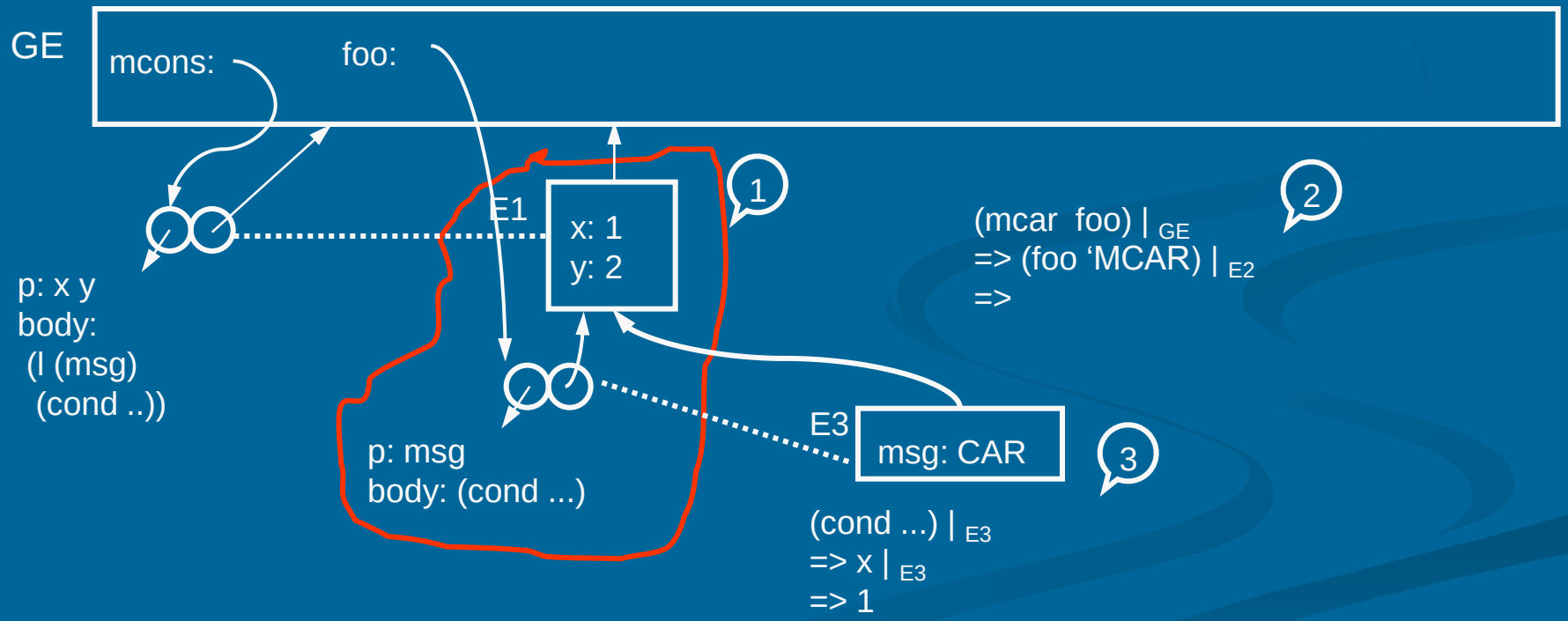
```
(define (mcons x y)
  (lambda (msg)
    (cond ((eq? msg 'MCAR) x)
          ((eq? msg 'MCDR) y)
          ((eq? msg 'MPAIR?) #t)
          (else (display "wrong msg")))))

(define (mcar p)   (p 'MCAR))
(define (mcdr p)   (p 'MCDR))
(define (mpair? p)
    (and (procedure? p) (p 'MPAIR?)))
```

Пара-объект с точки зрения MBO

(define foo (mcons 1 2)) 1

(mcar foo) 2 ====> (foo ‘MCAR) 3

```
(define (mcons x y)
  (lambda (msg)
    (cond ((eq? msg ’MCAR) x)
          ((eq? msg ’MCDR) y)
          ((eq? msg ’MPAIR?) #t)
          (else ...)
```



Мутация пары как изменение состояния объекта

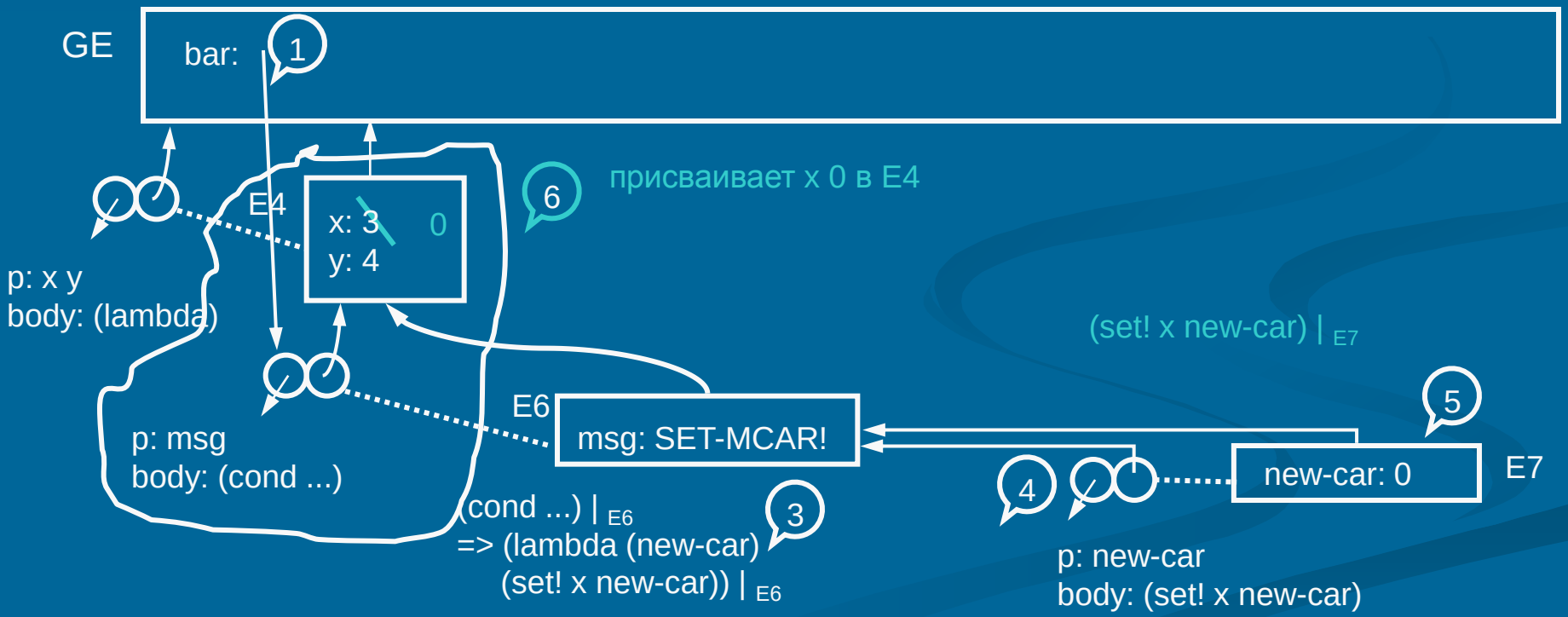
```
(define (mcons x y)
  (lambda (msg)
    (cond ((eq? msg ‘MCAR) x)
          ((eq? msg ‘MCDR) y)
          ((eq? msg ‘MPAIR?) #t)
          ((eq? msg 'SET-MCAR!)
           (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-MCDR!)
           (lambda (new-cdr) (set! y new-cdr)))
          (else ...)
(define (set-mcar! p new-car)
  ((p 'SET-MCAR!) new-car))
(define (set-mcdr! p new-cdr)
  ((p 'SET-MCDR!) new-cdr))
```

Пример мутации в MBO

(define bar (mcons 3 4)) 1

(set-mcar! bar 0)

2 (set-mcar! bar 0) | GE
=> ((bar 'SET-CAR!) 0) | E5



Стиль передачи сообщений

- дадим имена lambda и сделаем их приватными

```
(define (mcons x y)
  (define (change-car new-car) (set! x new-car))
  (define (change-cdr new-cdr) (set! y new-cdr))
  (lambda (msg . args)
    (cond ((eq? msg ‘MCAR) x)
          ((eq? msg ‘MCDR) y)
          ((eq? msg ‘MPAIR?) #t)
          ((eq? msg 'SET-MCAR!) (change-car (first args)))
          ((eq? msg 'SET-MCDR!) (change-cdr (first args)))
          (else …))))
```

- перепишем селектор и мутатор

```
(define (mcar p)
          (p ‘MCAR))
(define (set-mcar! p val)
          (p 'SET-MCAR! val))
```

Стили программирования: функциональный и объектно-ориентированный

- Функциональное программирование:
  - Система организуется набором функций, обрабатывающих данные

  (do-something <data> <arg> ...)

  (do-another-thing <data>)
- ООП:
  - Система организуется как набор объектов, обменивающихся сообщениями

  (<object> 'do-something <arg>)

  (<object> 'do-another-thing)
  - Объект инкапсулирует данные и операции

Основные термины ООП

- **Класс**
  - описывает общую структуру и поведение однотипных экземпляров
  - в Scheme класс – это функция – операция-конструктор.

- **Экземпляр**:
  - Отдельный объект некоторого класса
  - в Scheme, объект – это функция-обработчик сообщений, созданная операцией-конструктором некоторого класса

Пример. «Космические войны»

- В игре участвуют
  - корабли
  - планеты
  - объекты других типов
- Примеры экземпляров классов
  - Millenium Falcon
  - Enterprise
  - Earth

Корабль

```
(define (make-ship position velocity num-torps)
    (define (move)
      (set! position (add-vect position ...)))
    (define (fire-torp)
      (cond ((> num-torps 0) ...)
            (else ‘fail)))
    (lambda (msg)
      (cond ((eq? msg ‘position) position)
            ((eq? msg ‘velocity) velocity)
            ((eq? msg ‘move) (move))
            ((eq? msg ‘attak) (fire-torp))
            (else (display “wrong message to ship”)))))
```

Класс корабль

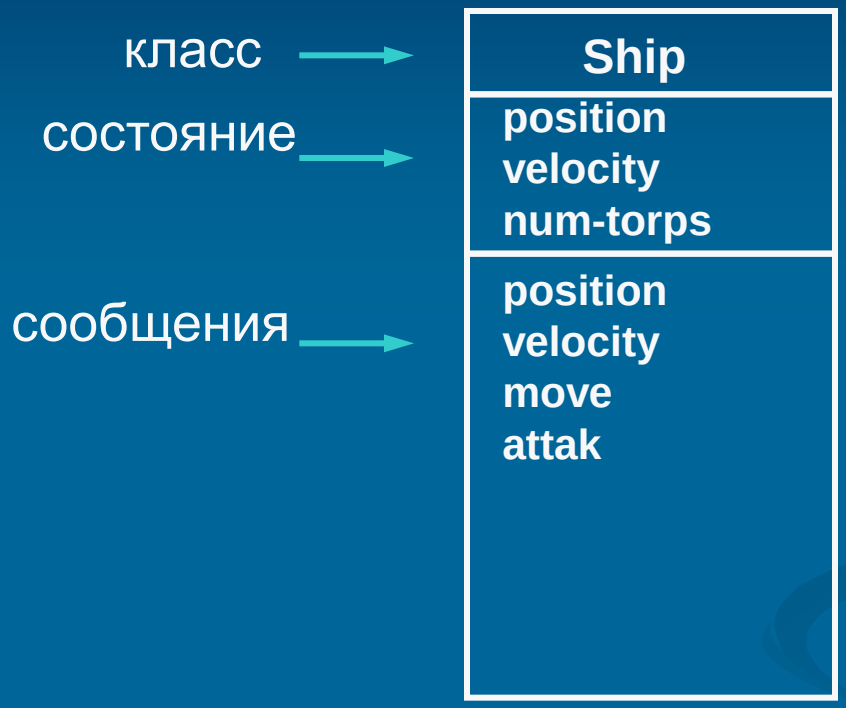
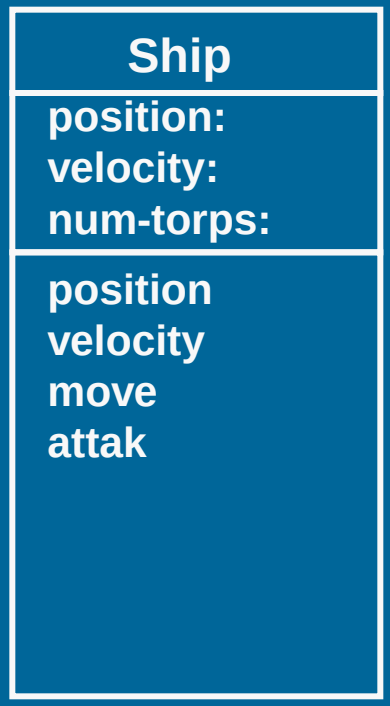
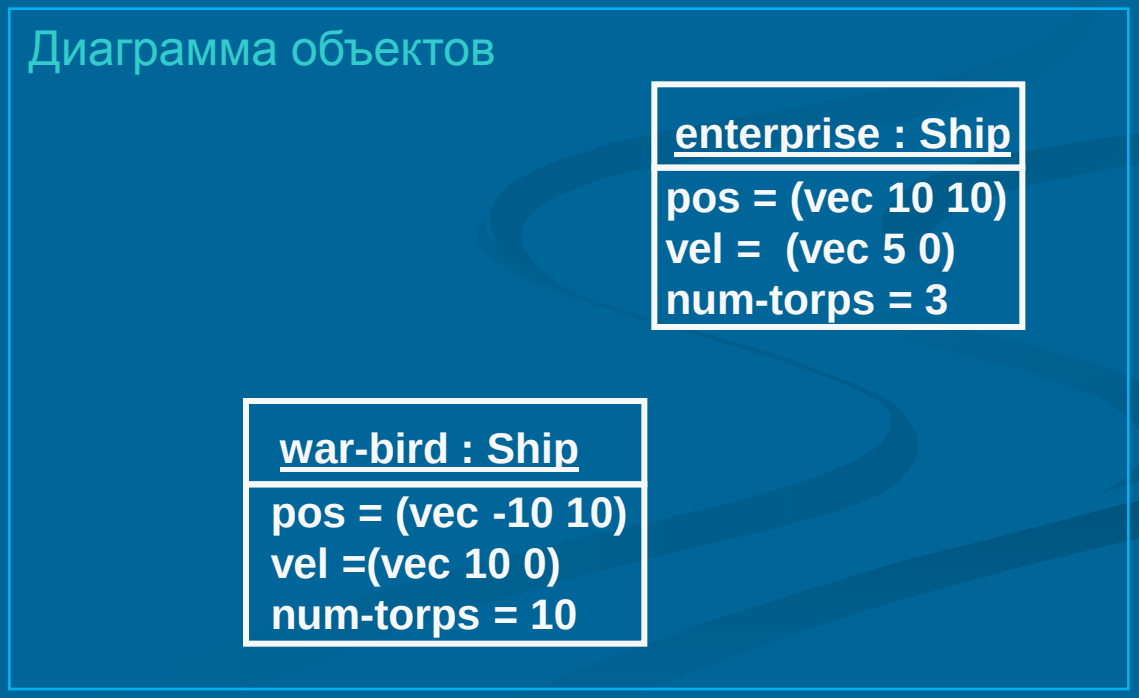


Диаграмма объектов

```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(define war-bird
  (create-ship (make-vect -10 10) (make-vect 10 0) 10))
```



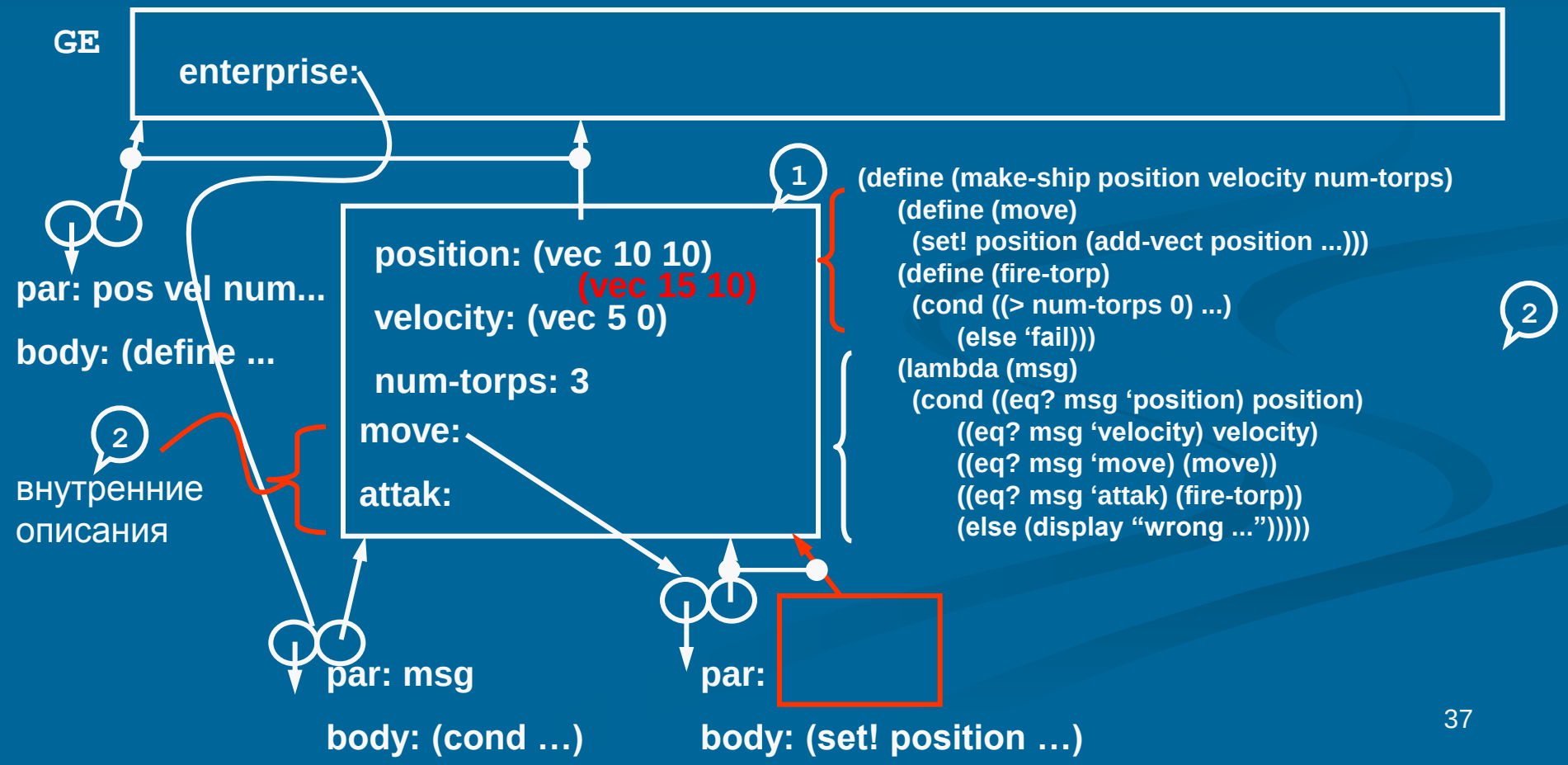


С точки зрения модели вычислений с окружениями

(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))

(enterprise ‘move) ==>

(enterprise ‘position) ==> ? (vec 15 10)

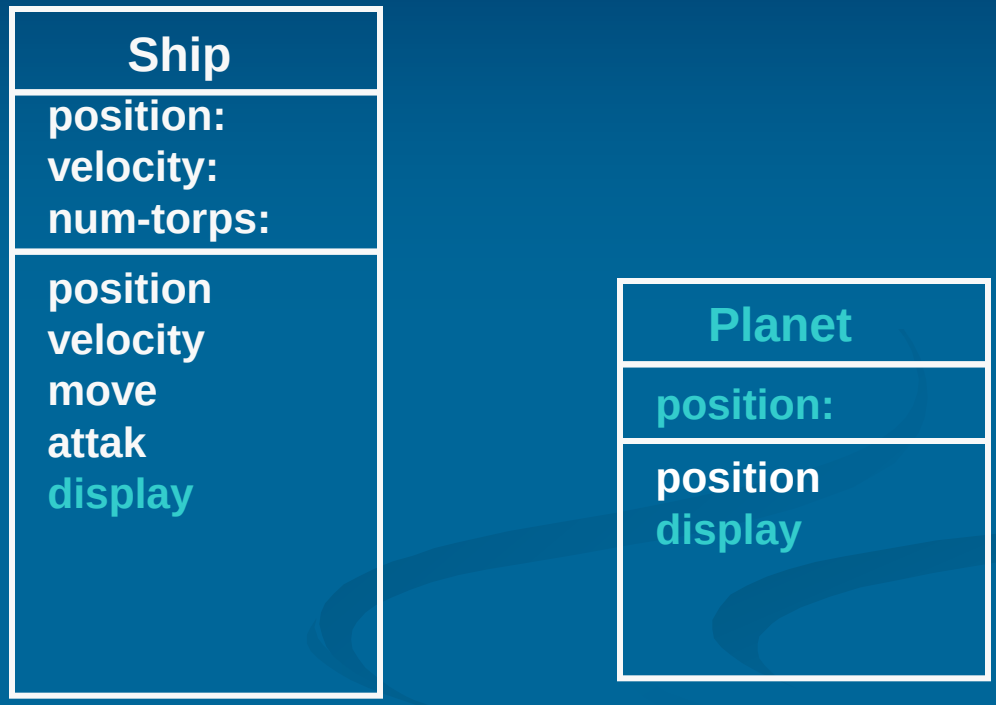


Продолжим наполнять мир «космических войн»

- Добавим планеты – класс Planet

- Добавим вывод
  - отображение объектов на экране
  - допустим, что вывод умеет делать функция draw
  - добавим сообщение ‘display, обработка которого заключается в вызове draw с нужными параметрами.

Обновленная диаграмма классов



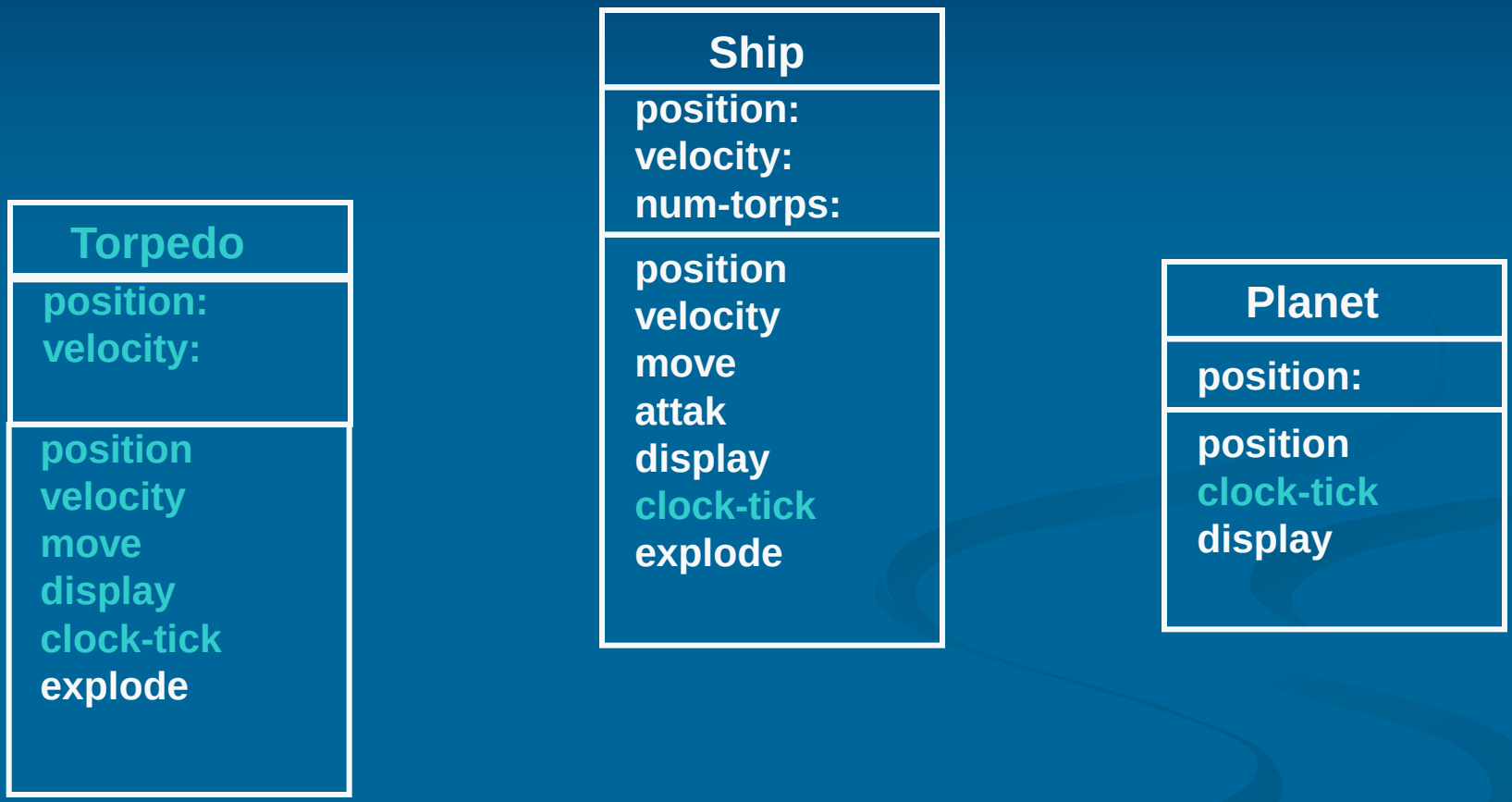
Класс Планета

```
(define (make-planet position)
  (lambda (msg)
    (cond
        ((eq? msg ‘position) position)
        ((eq? msg ‘display) (draw ...))
        (else (display “wrong message to planet”)))))

(define earth (make-planet (make-vect 0 0)))
(earth ‘position) ==> ...
```

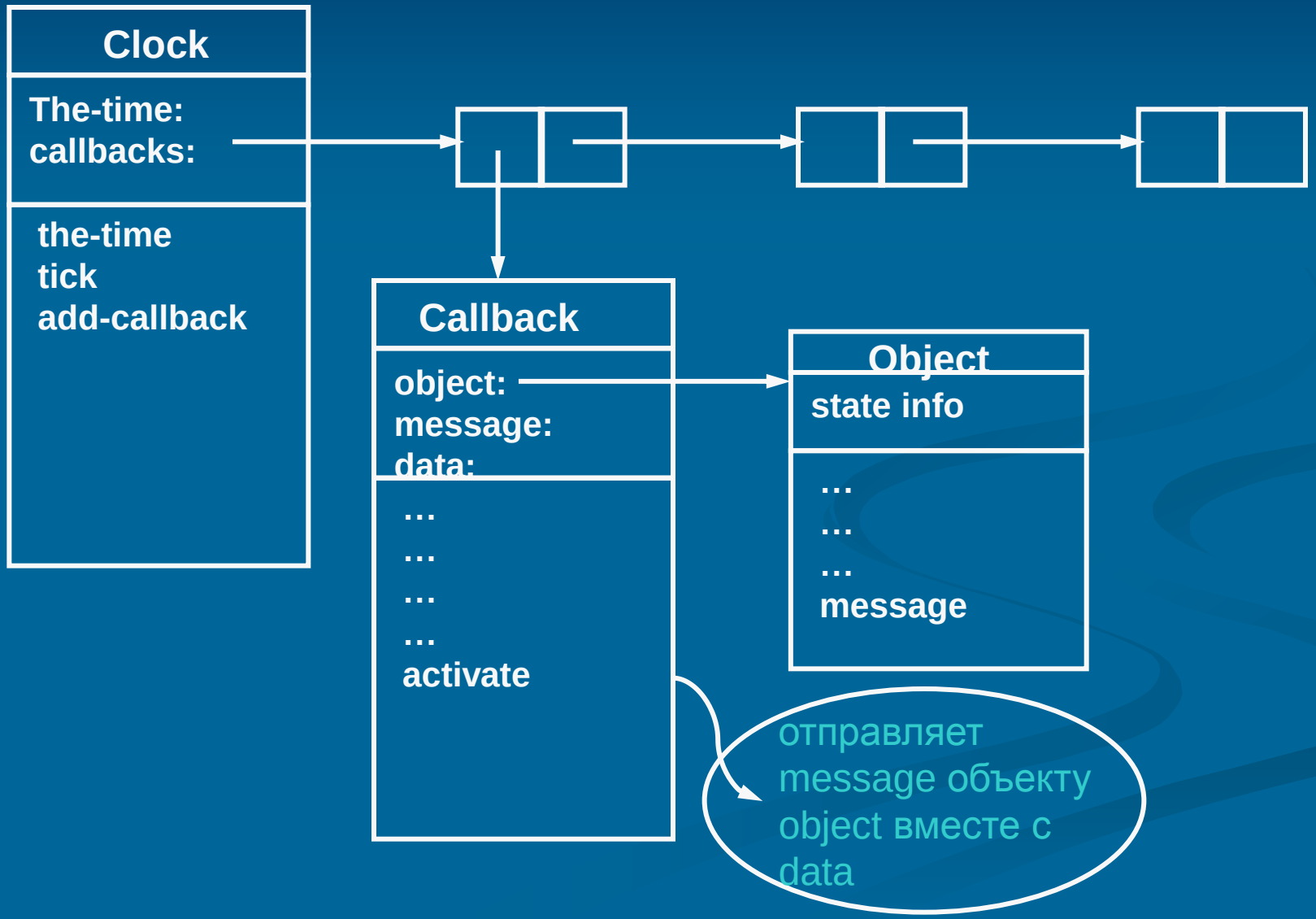
Запустим время

- Движение в космосе
  - добавим часы, отсчитывающие время
  - будем отслеживать положение движущихся объектов
  - часы будут посылать сообщение ‘clock-tick движущимся объектам, чтобы те обновили своё состояние
- Добавим класс Torpedo

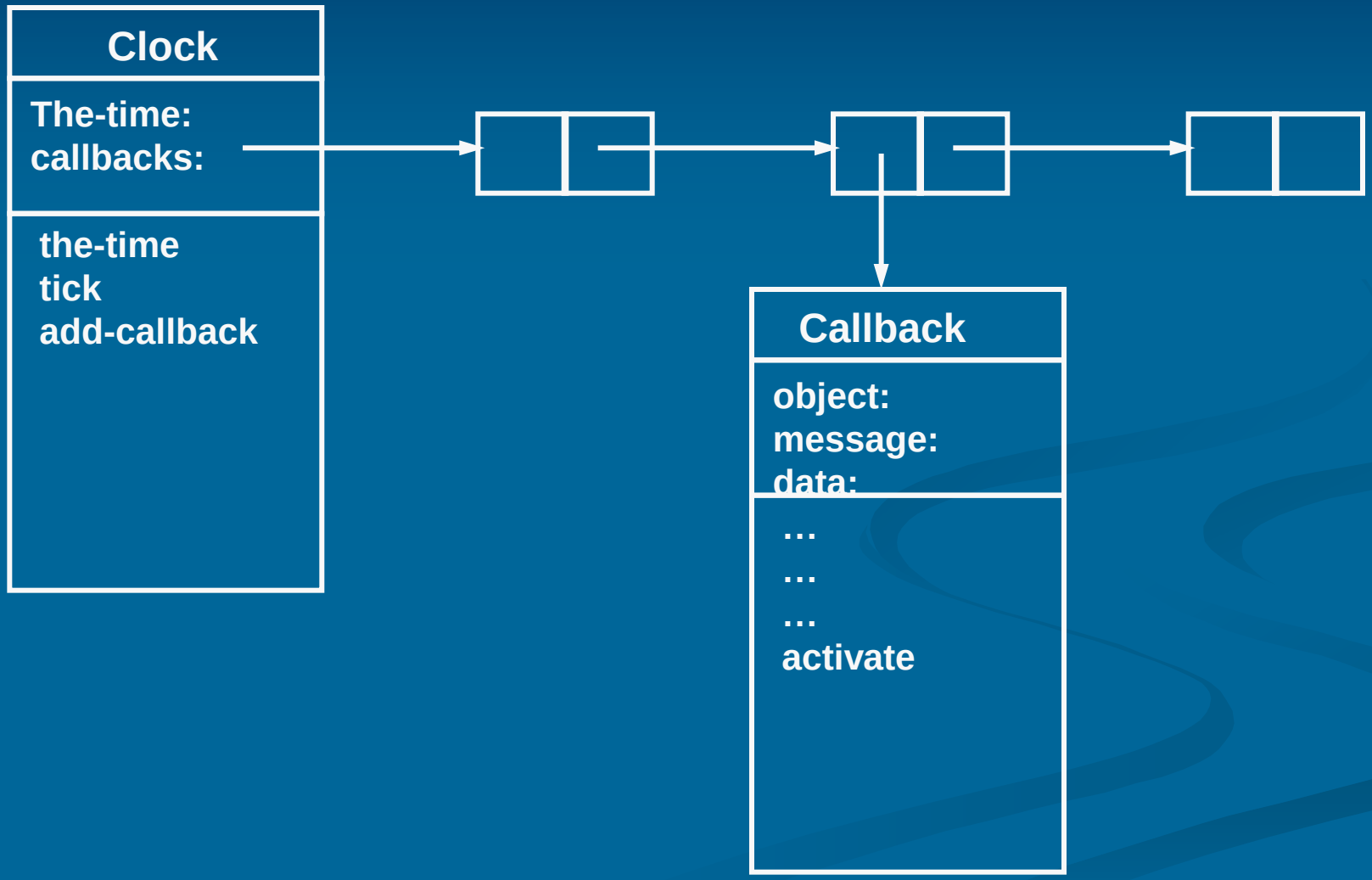
Изменения на диаграмме классов



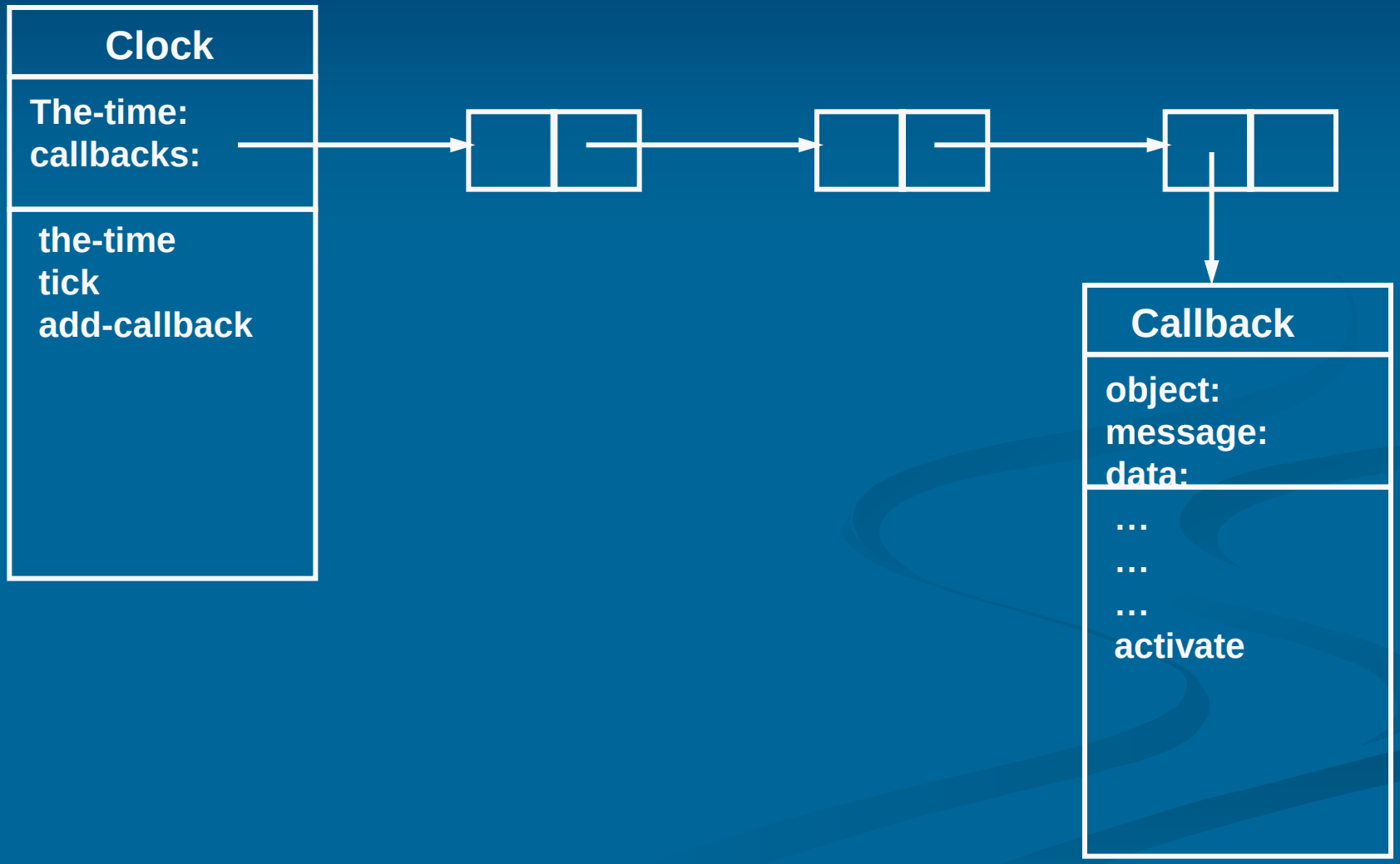
Часы управляют всем



Часы управляют всем



Часы управляют всем



Реализация часов

```
(define (make-clock . args)
  (let ((the-time 0)
        (callbacks '()))
    (lambda (msg)
      (cond
          ((eq? msg ‘the-time) (lambda (self) the-time))
          ((eq? msg ‘tick)  (lambda (self)
                      (map (lambda (x) (x 'activate)) callbacks)
                      (set! the-time (+ the-time 1))))
          ((eq? msg ‘add-callback)
           (lambda (self cb)
            (set! callbacks (cons cb callbacks))
            'added)))
           (else (display “wrong message to clock”))))))
```

Реализация callback

```
(define (make-clock-callback object message . data)
  (lambda (msg)
    (cond
      ((eq? msg ‘object)  (lambda (self) object))
      ((eq? msg ‘message)  (lambda (self) message))
      ((eq? msg ‘activate) (lambda (self)  (object message data)))
       (else (display “wrong message to clock-callbak”)))))
```

Торпеда

```
(define (make-torpedo position velocity)
  (define (explode torp)
    (display “torpedo goes off!”)
    (remove-from-universe torp))
  (define (move)
    (set! position ...))
  (define (me msg . args)
    (cond
          ((eq? msg ‘position) position)
          ((eq? msg ‘velocity) velocity)
          ((eq? msg ‘move) (move))
          ((eq? msg ‘explode) (explode (car args)))
          ((eq? msg ‘display) (draw ...))
          (else (display “wrong message to torp”))))
  (clock ‘add-callback
     (make-clock-callback me   ‘move))
me)
```

объект сообщение

Промежуточные итоги

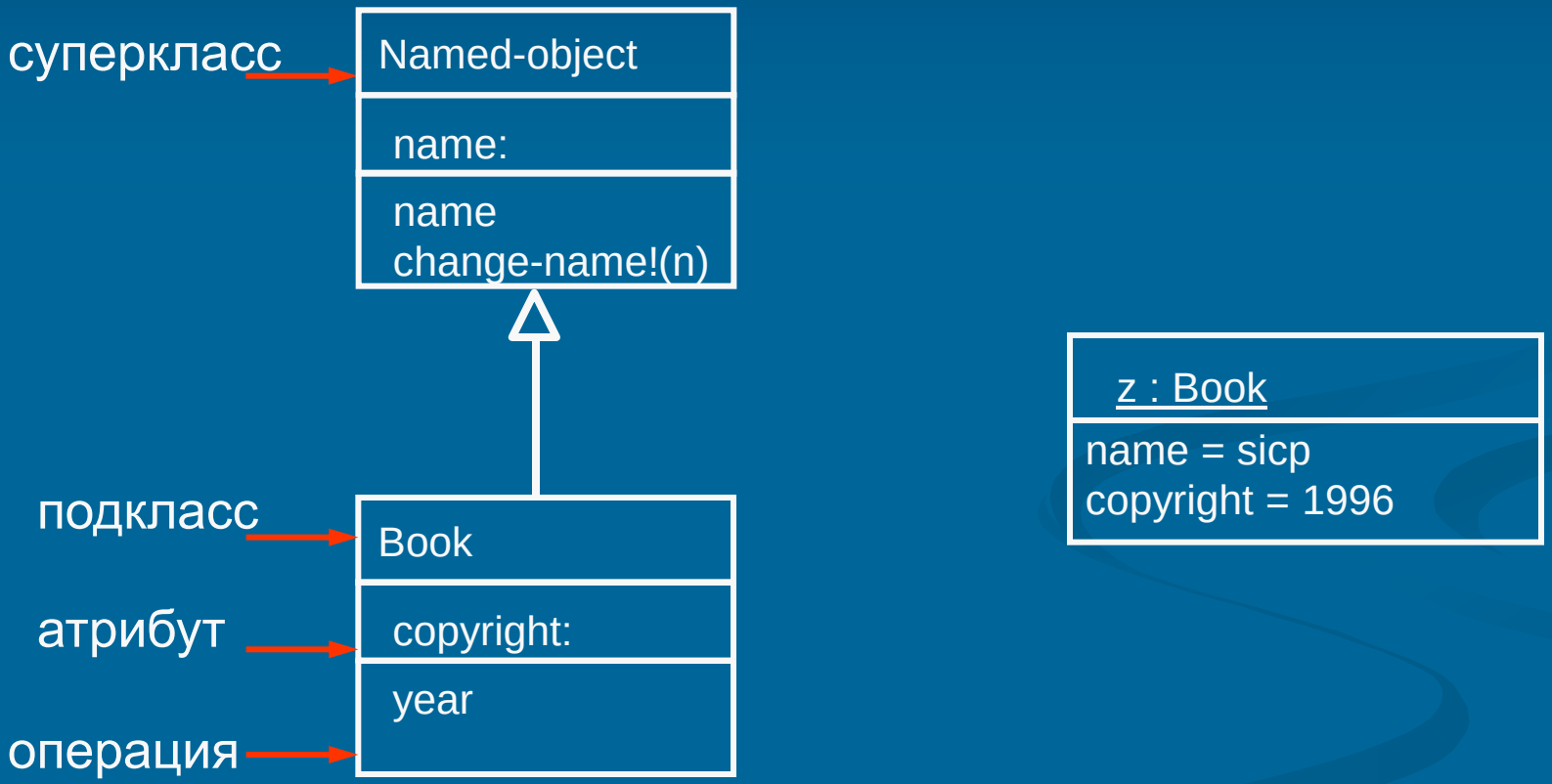
- Рассмотрели ОО-стиль программирования:
  - отличный от функционального
  - подходящий для симуляторов, сложных систем, ...
- Рассмотрели объектные модели
  - н/з от языка реализации

    класс – шаблон структуры и поведения объектов

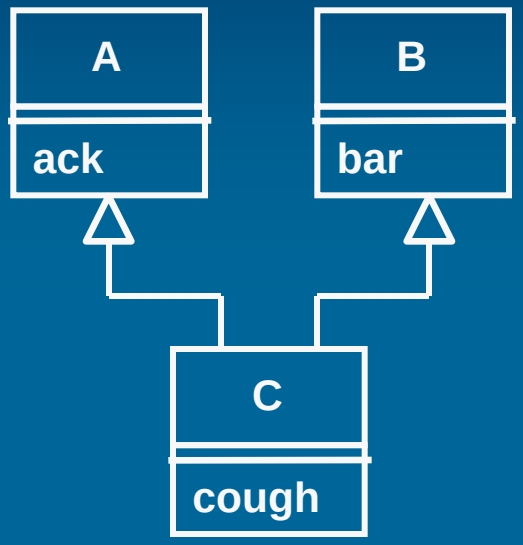
    экземпляр – конкретный объект, созданный по шаблону

    диаграммы классов и диаграммы объектов

Диаграмма с обобщением (наследованием)



Множественное наследование



- Суперкласс и подкласс
  - A – суперкласс C
  - C – подкласс A и B
- Подкласс наследует атрибуты и операции своих суперклассов
  - У C три операции: ack, bar, cough

Класс именованных объектов

Диаграмма классов

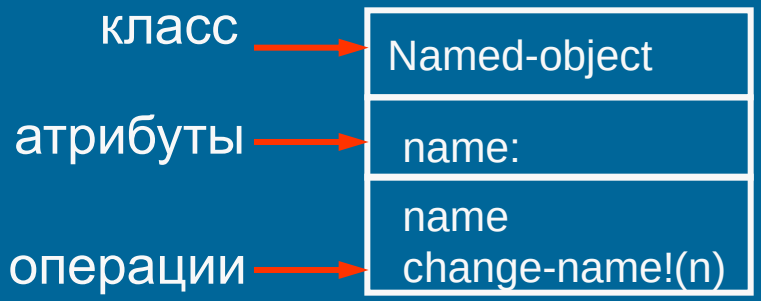
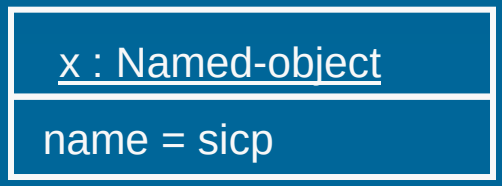
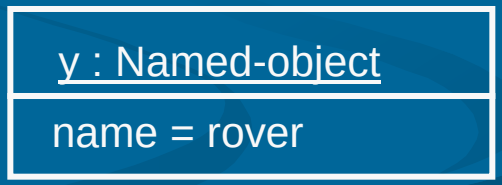


Диаграмма объектов





Учёт наследования

- Класс: описывается функцией, возвращающей конструктор
  - определяет общую структуру и поведение своих экземпляров
    - набор атрибутов
    - обработчик сообщений
    - определяет суперклассы, наследуемую структуру и поведение.
  - Корневой класс: Root
    - Корень иерархии наследования. Все классы – напрямую или косвенно его наследники.
  - Типизация:
    - Каждый класс должен реализовывать операцию type, возвращающую путь к Root в иерархии

Учёт наследования

- Экземпляр: порождается конструктором (например create-named-object)
  - все экземпляры различны в смысле eq?
  - новый формат отправки сообщений:

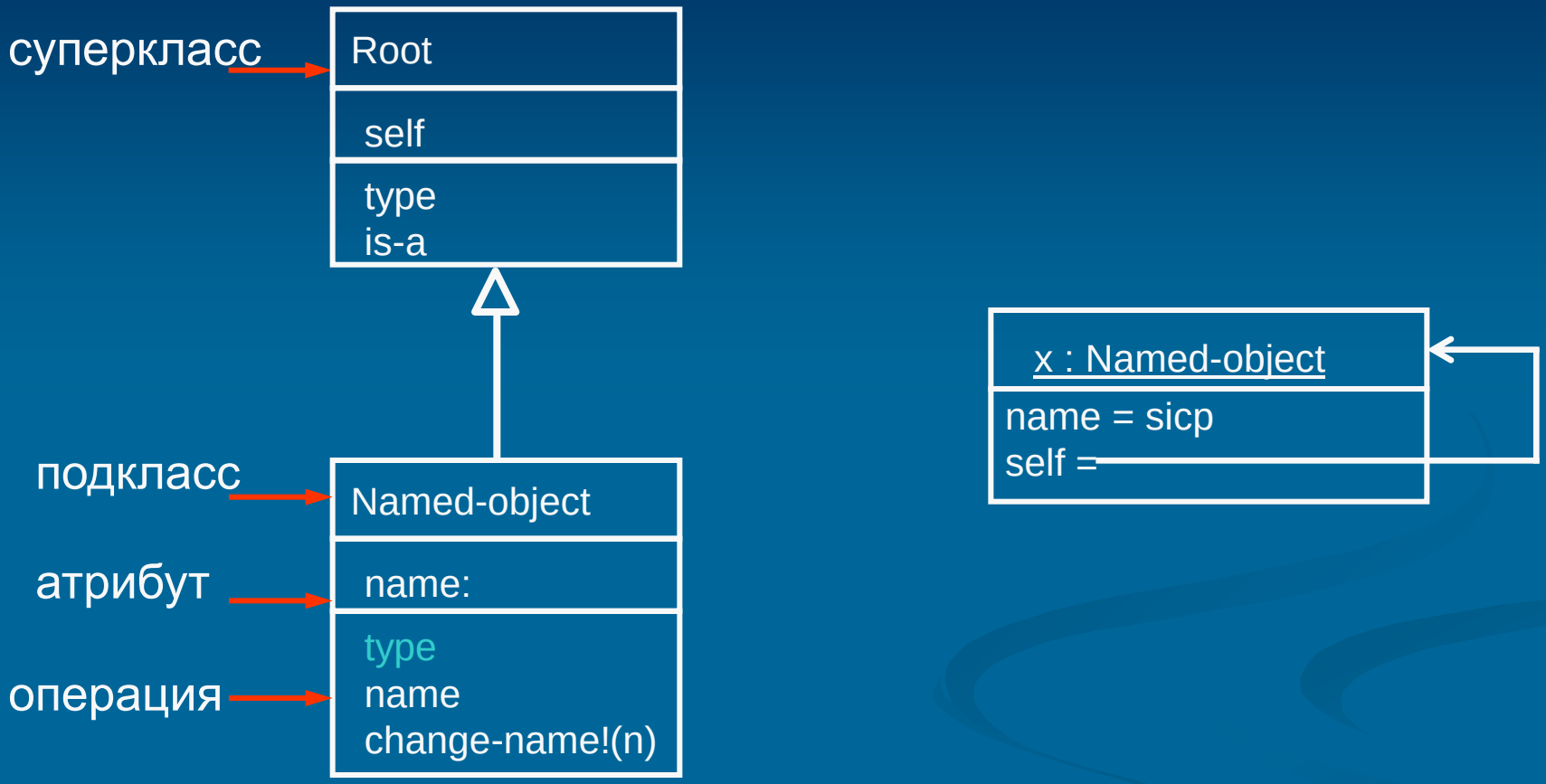
(ask <instance> ‘<message> <arg1> … <argn>)

  - у всех экземпляров есть операции:

(ask <instance> ‘type) ==> (<type> <supertype> …)

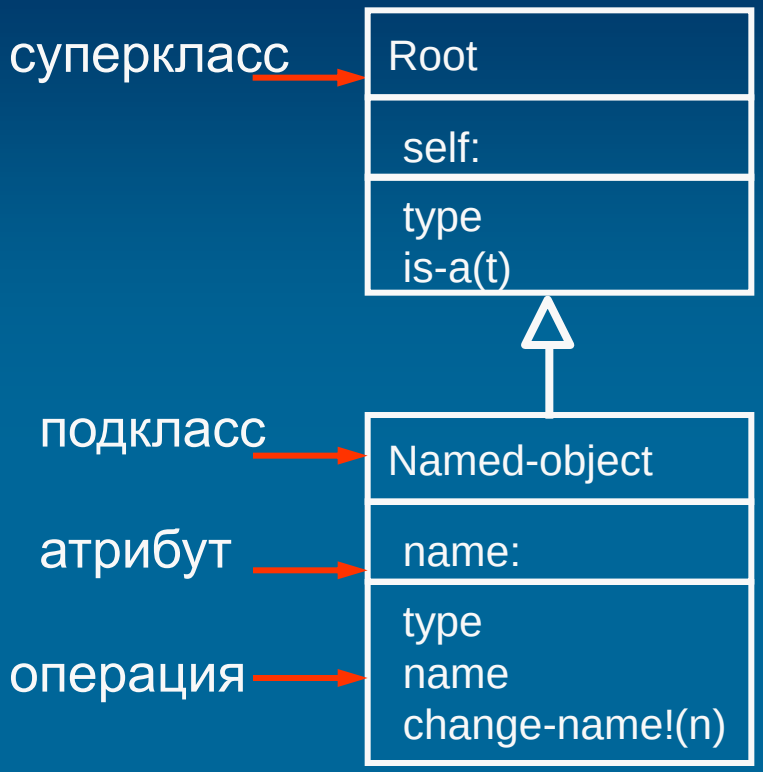
(ask <instance> ‘is-a <some-type>) ==> <boolean>

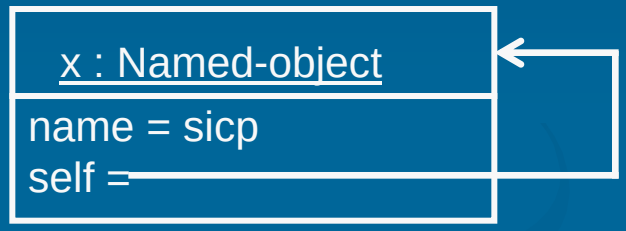
Обновлённая диаграмма



- **Named-object** наследует от **Root:**
  - атрибут self : у каждого объекта есть ссылка на себя
  - операцию **is-a**
  - переопределяет операцию **type**

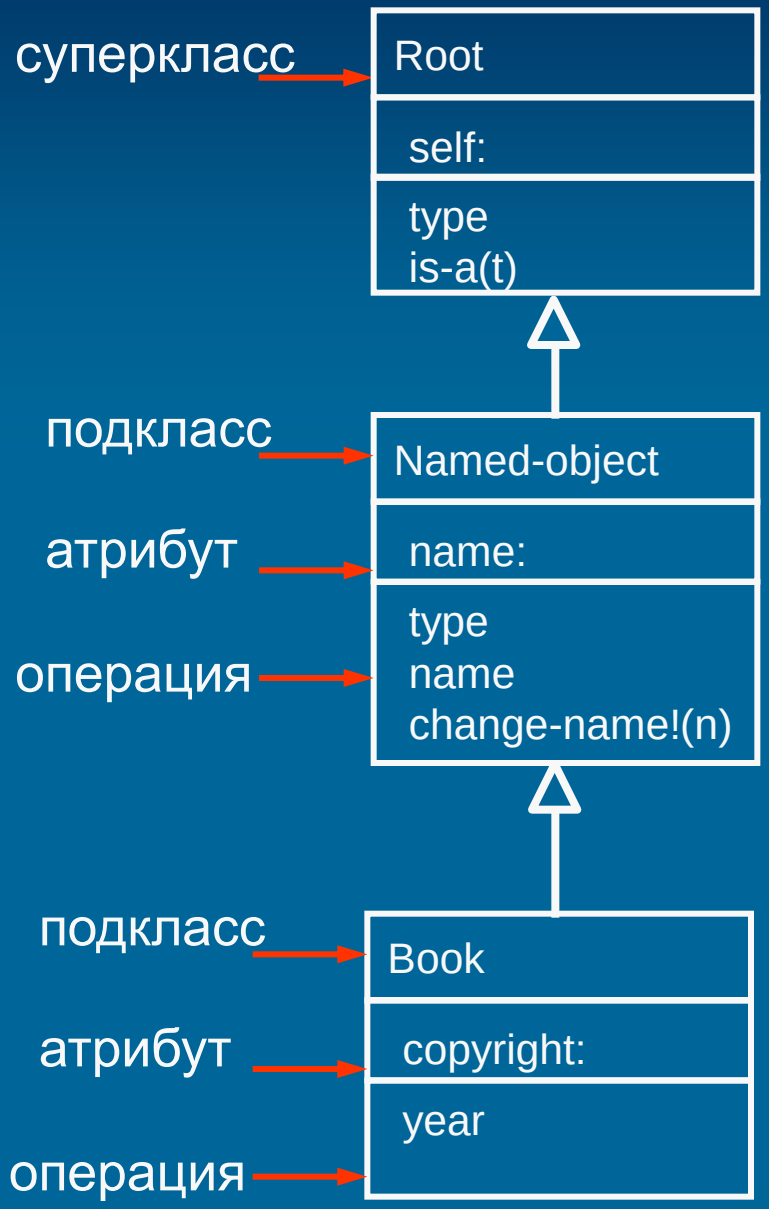
Обновлённая диаграмма

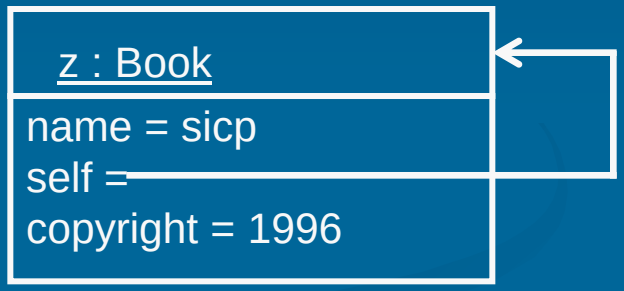





```
(define x (create-named-object 'sicp))
(ask x 'name) => sicp
(ask x 'change-name! 'sicp-2nd-ed)
(ask x 'name) => sicp-2nd-ed)
(ask x 'type) => (named-object root)
(ask x 'is-a 'named-object) => #t
(ask x 'is-a 'clock) => #f
(ask x 'is-a 'root) => #t
```

Обновлённая диаграмма





(define z (create-book 'sicp 1996))

(ask z 'year) ==> 1996
(ask z 'name) ==> sicp
(ask z 'is-a 'book) ==> #t
(ask z 'is-a 'named-object) ==> #t

Типовое описание класса

- Описание класса по имени <type> содержит
  - указание суперклассов
  - состояние (включающее self)
  - обработчик сообщений, определяющий сообщения и операции
    - обязательно обрабатывающий type как указано ниже
    - имеющий ветвь (else (get-method …)) для сообщений суперкласса

```
(define (<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (<super1> self <args>)
        (<super2>-part (<super2> self <args>)
        <other superclasses>
        <other local state> )
    (lambda (msg)
      (cond
        ((eq? msg ‘type) (lambda () (type-extend ‘<type> <super1>-part
                                    <super2>-part …)) )
        <другие сообщения и операции>
        (else (get-method msg <super1>-part <super2>-part …))))))
```

Порождение экземпляров. Объекты Instance

- Для класса с именем <type> описываем функцию create-<type>
  - Она использует create-instance – функцию высшего порядка
    - порождающую объект Instance
    - добавляющую объекту Instance обработчик сообщений класса <type>
    - возвращающую ссылку на объект Instance

```
(define (create-<type> <arg1> <arg2> … <argn>)
  (create-instance <type> <arg1> <arg2> … <argn>) )
```

- Объект создаётся вызовом create-<type>

```
(define <instance> (create-<type> <arg1> <arg2> … <argn>))
```

Пример. Класс Book

конструктор для Book

```
(define (create-book name copyright)
  (create-instance book name copyright))
```

обработчик сообщений

атрибуты Book

```
(define (book self name copyright)
  (let ((named-object-part    (named-object  self  name)))
    (lambda (msg)
      (cond
        ((eq? msg 'type) (lambda () (type-extend 'book 'named-object)))
        ((eq? msg 'year) (lambda () copyright))
        (else (get-method msg named-object-part)))))))
```

суперкласс

создание обработчика суперкласса

новое сообщение

унаследованные методы

Ещё пример: класс Named-object

```
(define (create-named-object name)
   (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
        ((eq? msg ‘type)
             (lambda () (type-extend 'named-object root-part)))
        ((eq? msg ‘name) (lambda () name))
        ((eq? msg ‘change-name!) (lambda (newname) (set! name newname)))
        (else (get-method msg root-part))))))
```

Использование объектов Instance

- поиск метода: get-method для <сообщения> в <instance>
- вызов метода
- оба шага сразу: **ask**

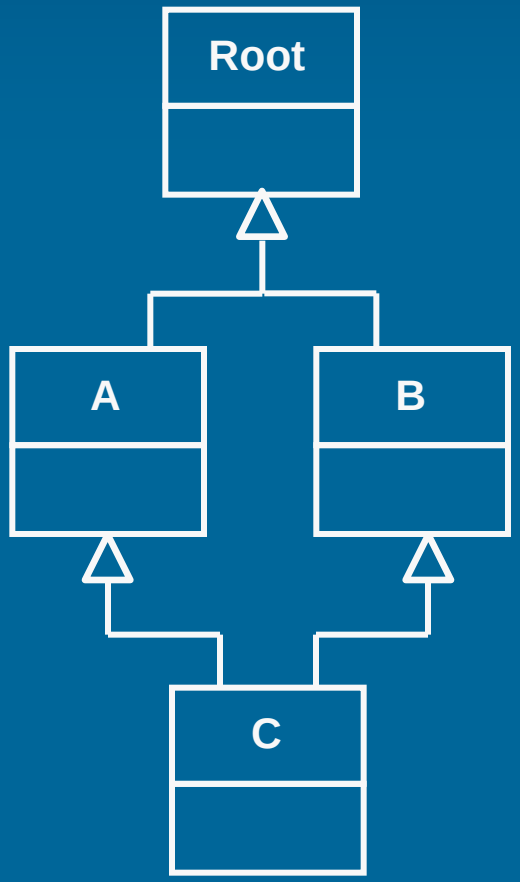
(define <instance> (create-<type> <arg1> <arg2> … <argn>))

(define sm-method (get-method ‘<сообщение> <instance>))
(sm-method <m-arg1> <m-arg2> … <m-argm>)

(ask <instance> ‘<сообщение> <m-arg1> … <m-argm>)

Иерархия типов

- При наследовании у экземпляра несколько типов




```
(define a-instance (create-a))
(define c-instance (create-c))

(ask a-instance ‘type) => (a root)
(ask c-instance ‘type) => (c a b root)

(ask c-instance ‘is-a ‘c)=> #t
(ask c-instance ‘is-a ‘b)=> #t
(ask c-instance ‘is-a ‘a)=> #t
(ask c-instance ‘is-a ‘root)=> #t

(ask a-instance ‘is-a ‘c)=> #f
(ask a-instance ‘is-a ‘b)=> #f
(ask a-instance ‘is-a ‘a)=> #t
```

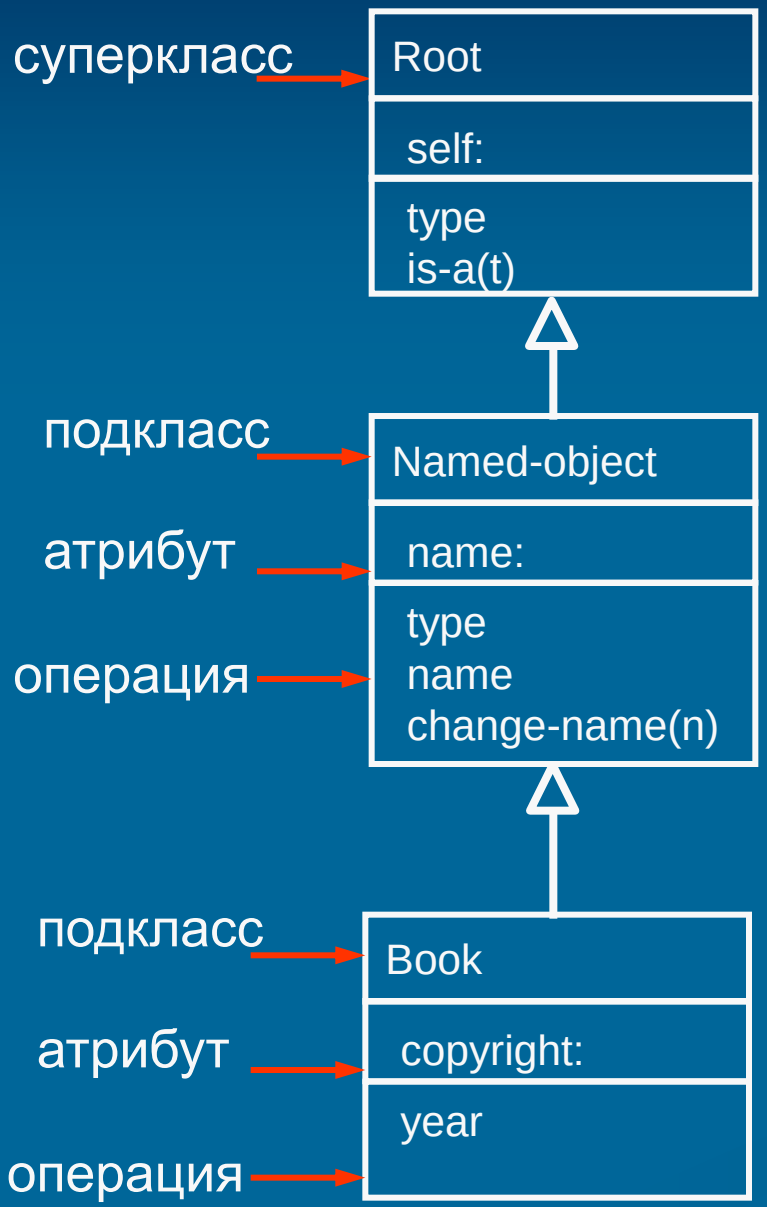
Точки зрения на ООП

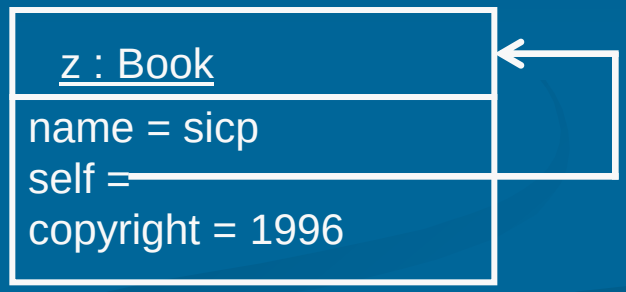
- Точка зрения модели
  - диаграмма классов и диаграмма объектов
  - терминология: сообщения, операции, обобщение, суперкласс, подкласс, …
- Точка зрения использования
  - Соглашения о том как писать ОО-код на Scheme:
    - описывать класс
      - указывать его суперклассы
    - порождать экземпляры
    - использовать экземпляры (вызывать операции)

Далее:

- ➔ Точка зрения реализации
  - Как реализованы экземпляры, классы, наследование

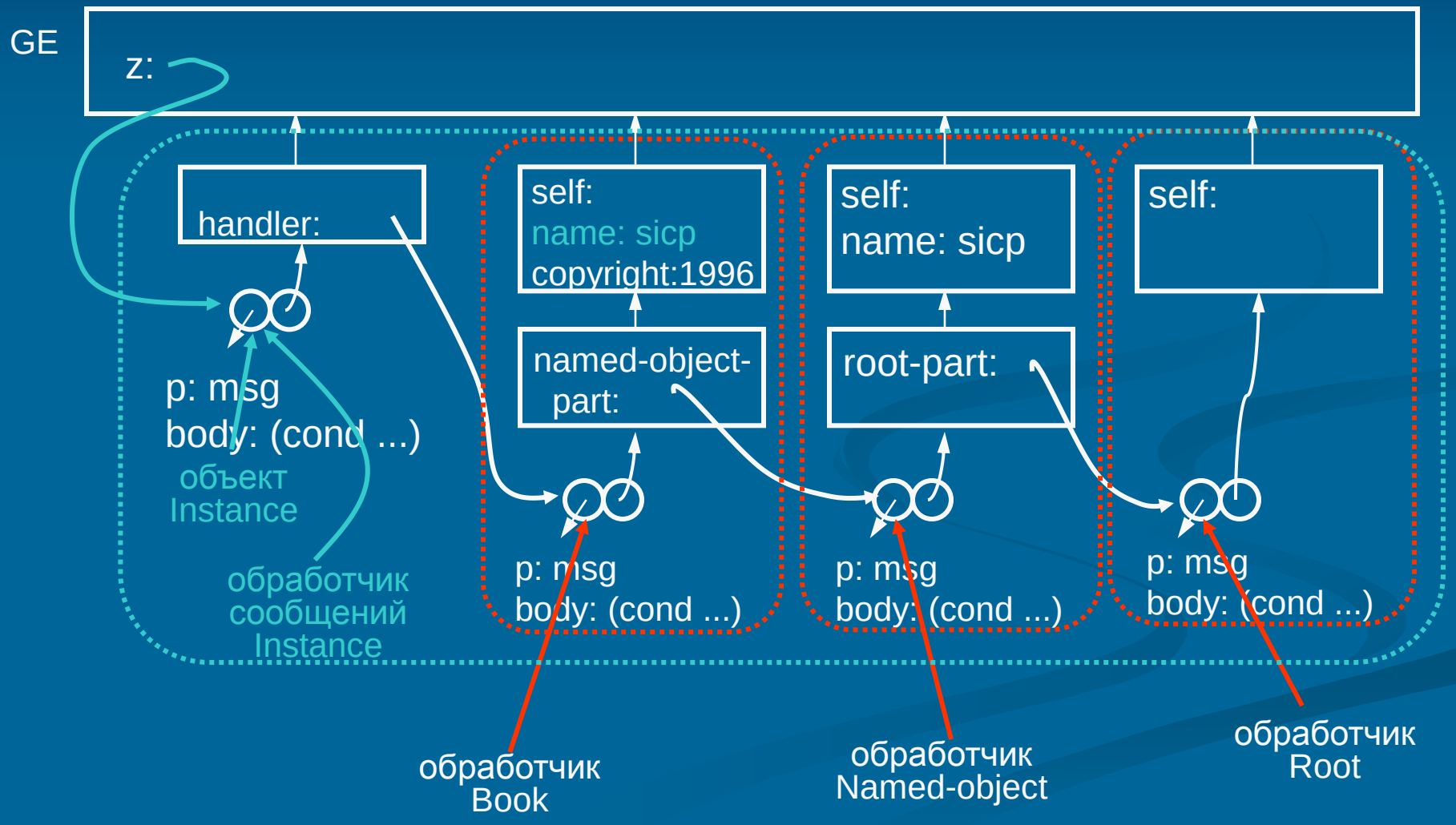
Вернёмся к примеру





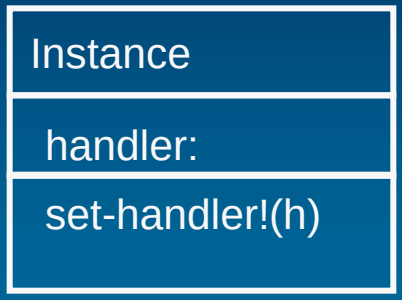
С точки зрения модели окружений

(define z (create-book ’sicp 1996))



Точка зрения реализации: Instance

```
(define (make-instance)
  (let ((handler #f))
    (lambda (msg)
      (cond
        ((eq? msg 'set-handler!)
         (lambda (handler-proc)
           (set! handler handler-proc)))
        (else (get-method msg handler))))))
```




```
(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (ask instance 'set-handler! handler)
    instance))
```

Точка зрения реализации: get-method и ask

- получить метод:

```
(define (get-method message object)
  (object message))
```

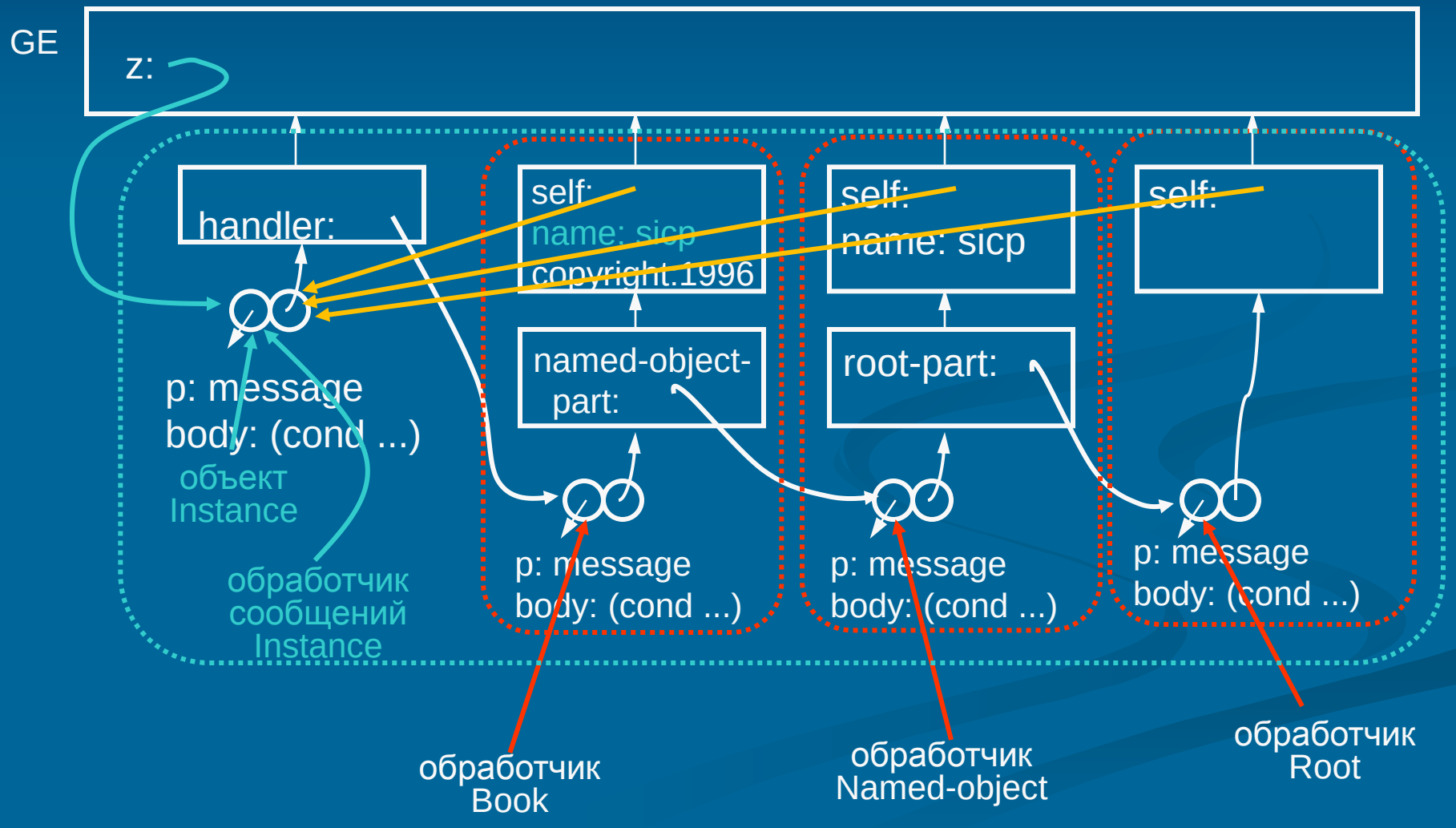
"ask" – комбинация поиска метода и его вызова

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (display “wrong method”))))
```

(apply op args) ==> (op arg1 arg2 … argn)

В модели окружений

(define z (create-book ’sicp 1996))



Точка зрения использования. Self

- В каждом классе есть атрибут self
  - **self** – это ссылка на весь экземпляр

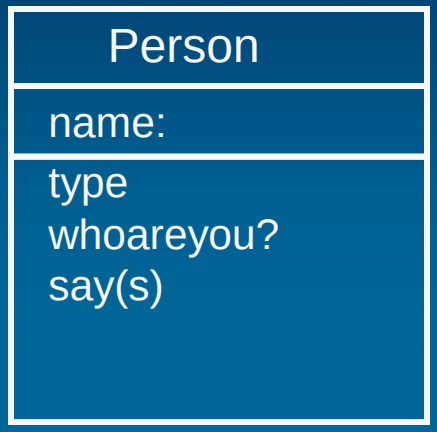
- Зачем? Как и когда используется **self** ?
  - При реализации операции объект может послать сообщение своей части: например, внутри операции класса Book, можно

    (ask named-object-part ‘change-name ‘mit-sicp)
  - Иногда нужно послать сообщение экземпляру целиком: например

    (ask self ‘year)
    - В ситуации, когда операция подкласса переопределяет операцию суперкласса нужна возможность указать, чью операцию мы хотим вызвать: объекта или унаследованной части объекта

- Рассмотрим на примере

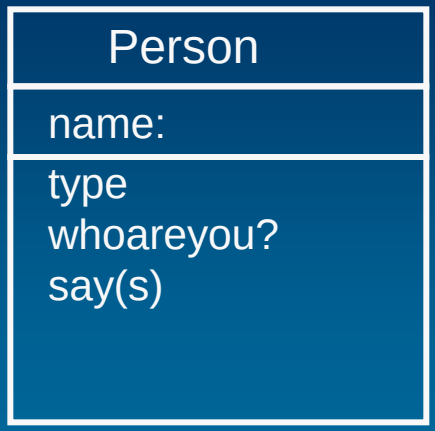
Класс Person



(define p1 (create-person 'joe))
(ask p1 'whoareyou?) ==> joe

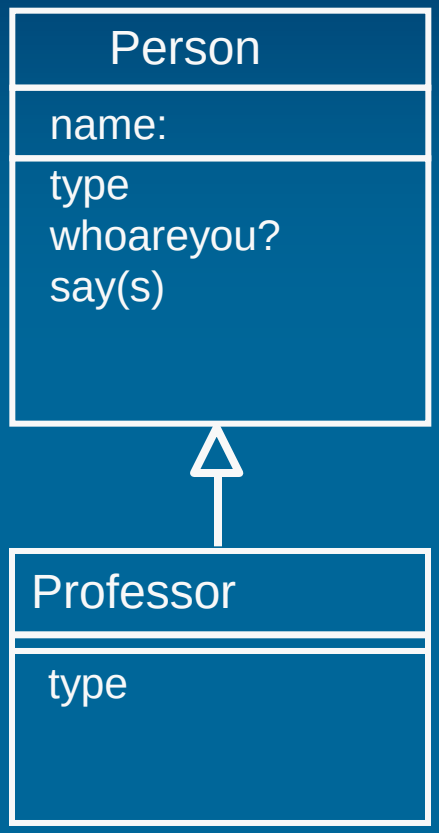
(ask p1 'say '(the sky is blue)) ==> (the sky is blue)

Реализация Person



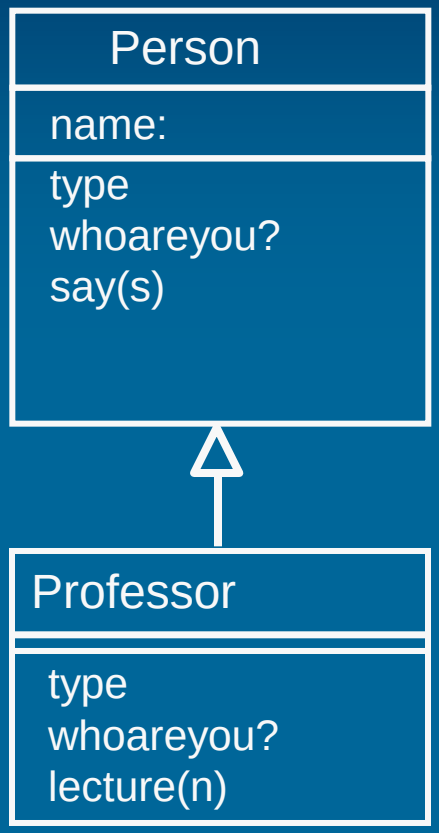

```
(define (create-person name)
  (create-instance person name))

(define (person self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
          ((eq? msg ‘type)
                              (lambda () (type-extend 'person root-part)))
          ((eq? msg ‘whoareyou?)    (lambda () name))
          ((eq? msg ‘say)                (lambda (stuff) stuff))
          (else (get-method msg root-part))))))
```

Класс Professor




```
(define prof1 (create-professor 'fred))

(ask prof1 'say '(the sky is blue))
==> (the sky is blue)
```

Класс Professor



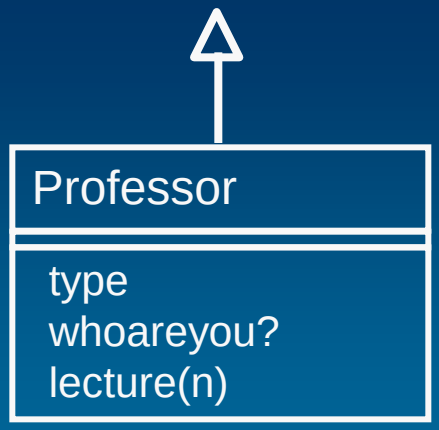
(define prof1 (create-professor 'fred))

(ask prof1 'whoareyou?)
==> (prof fred)
операция профессора ‘whoareyou? использует операцию персоны ‘whoareyou?

(ask prof1 'lecture '(the sky is blue))
==> (therefore the sky is blue)

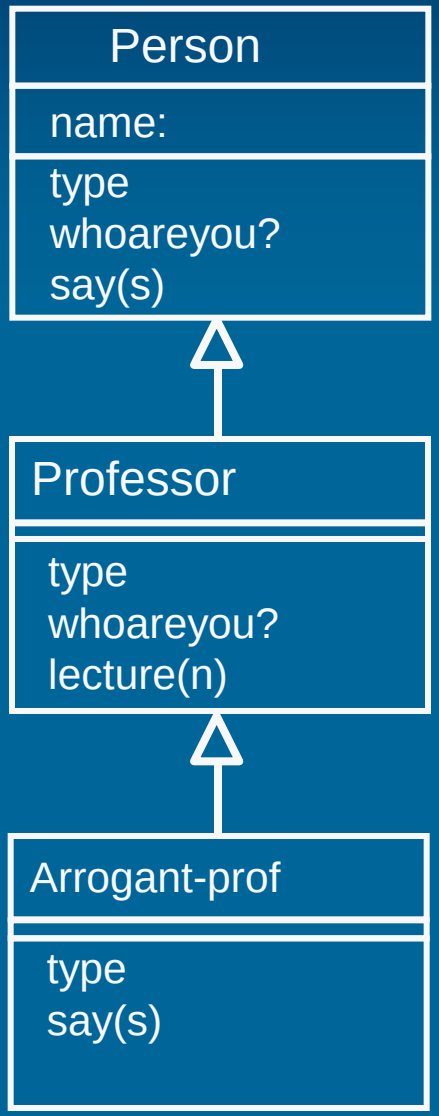
операция lecture использует операцию say

Реализация Professor



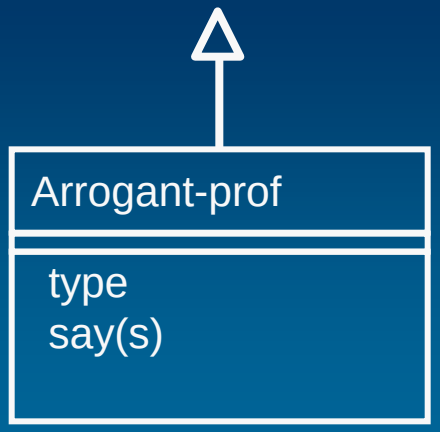

```
(define (create-professor name)
  (create-instance professor name))
(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
        ((eq? msg ‘type)
           (lambda () (type-extend 'professor person-part)))
        ((eq? msg ‘whoareyou?)
           (lambda () (list 'prof (ask person-part 'whoareyou?))))
        ((eq? msg ‘lecture)
           (lambda (notes)
             (cons 'therefore (ask person-part 'say notes))))
        (else (get-method msg person-part))))))
```

Класс Arrogant-prof



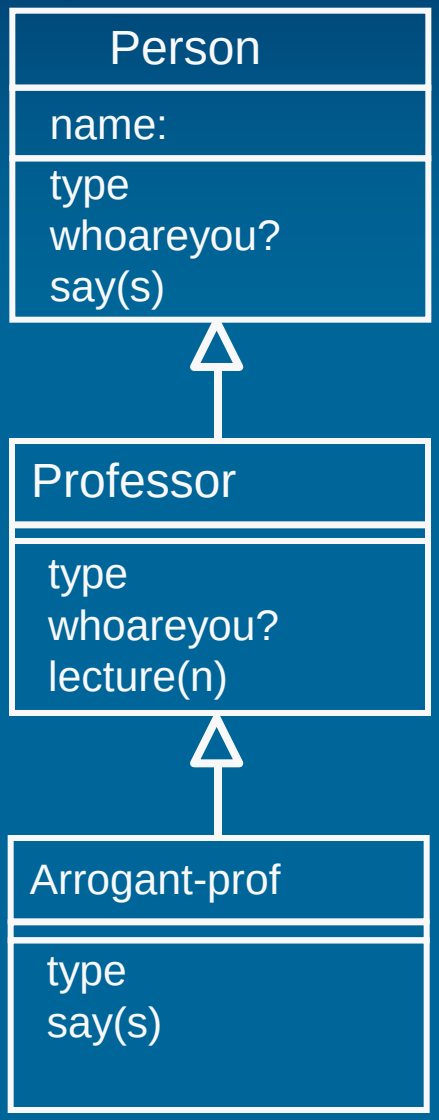

```
(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'whoareyou?)
==> (prof perfect)

(ask ap1 'say '(the sky is blue))
==> (the sky is blue obviously)
```

Реализация Arrogant-prof




```
(define (create-arrogant-prof name)
  (create-instance arrogant-prof name))

(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (msg)
      (cond
          ((eq? msg ‘type)
        (lambda () (type-extend 'arrogant-prof prof-part)))
          ((eq? msg ‘say) (lambda (stuff)
                    (append (ask prof-part 'say stuff)
                                (list 'obviously))))
         (else (get-method msg prof-part))))))
```

Чудаковатость Arrogant-prof

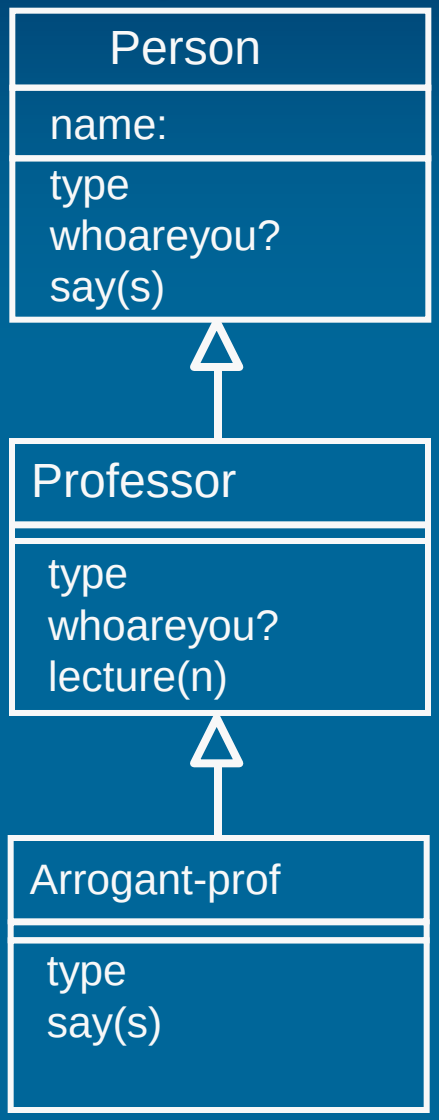


(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'lecture '(the sky is blue))
==> (therefore the sky is blue)

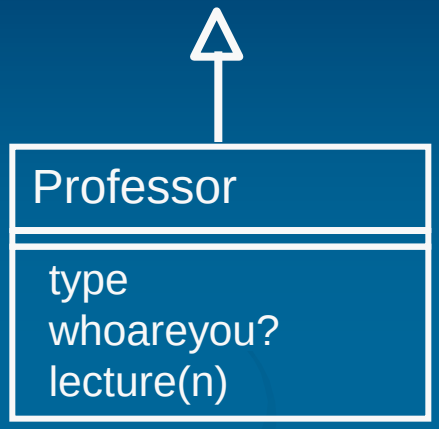
- Почему в конце фразы нет “obviously”?
  - lecture использует say класса Professor, а не say класса Arrogant-professor

Исправленный Arrogant-prof



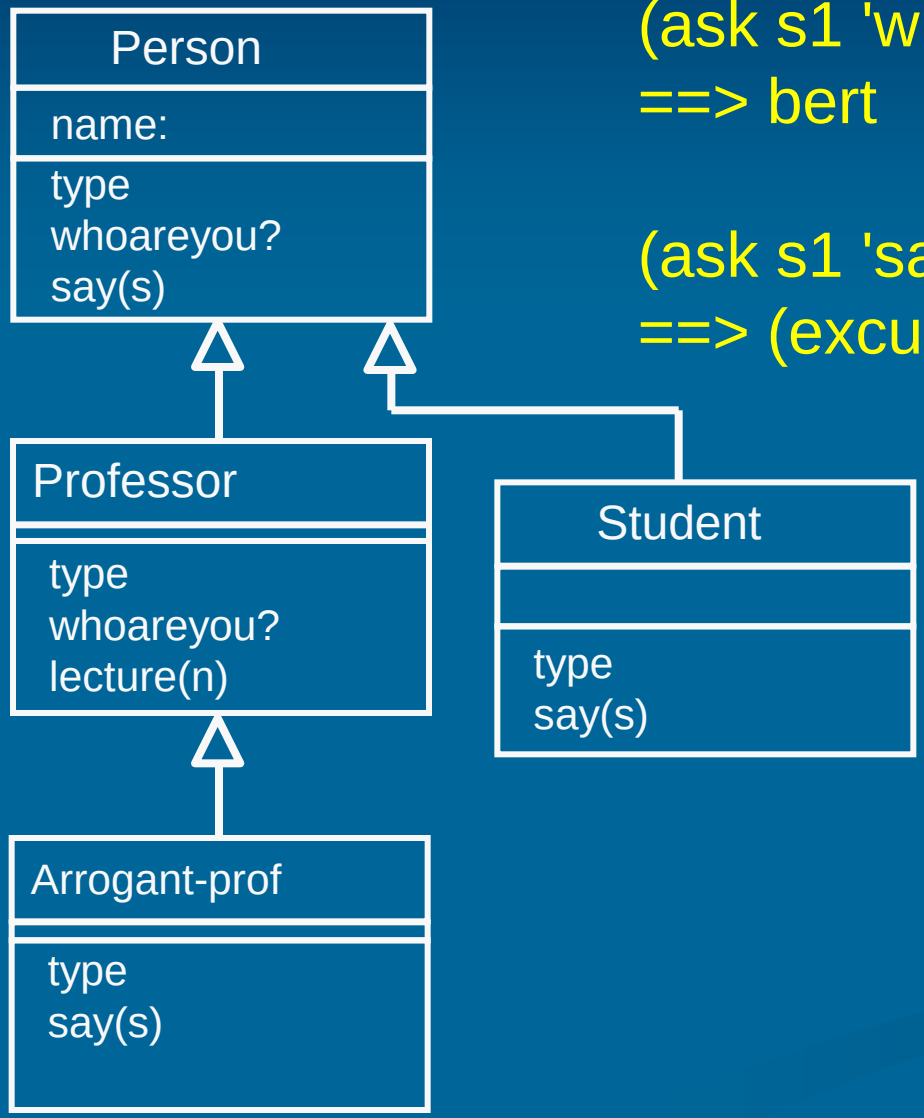

```
(define ap1 (create-arrogant-prof 'perfect))

(ask ap1 'lecture '(the sky is blue))
==> (therefore the sky is blue obviously)
```

Пересмотренная реализация Professor



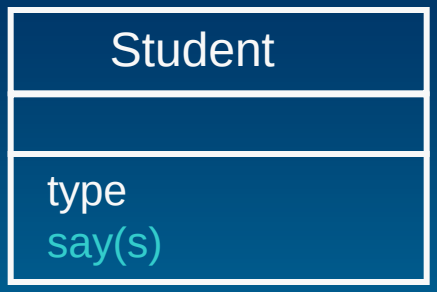

```
(define (create-professor name)
  (create-instance professor name))

(define (professor self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
            ((eq? msg ‘type)
        (lambda () (type-extend 'professor person-part)))
            ((eq? msg ‘whoareyou?) (lambda () (list 'prof name)))
            ((eq? msg ‘lecture) (lambda (notes)
                        (cons 'therefore
                                    (ask self 'say notes))))
           (else (get-method msg person-part))))))
```

Класс Student



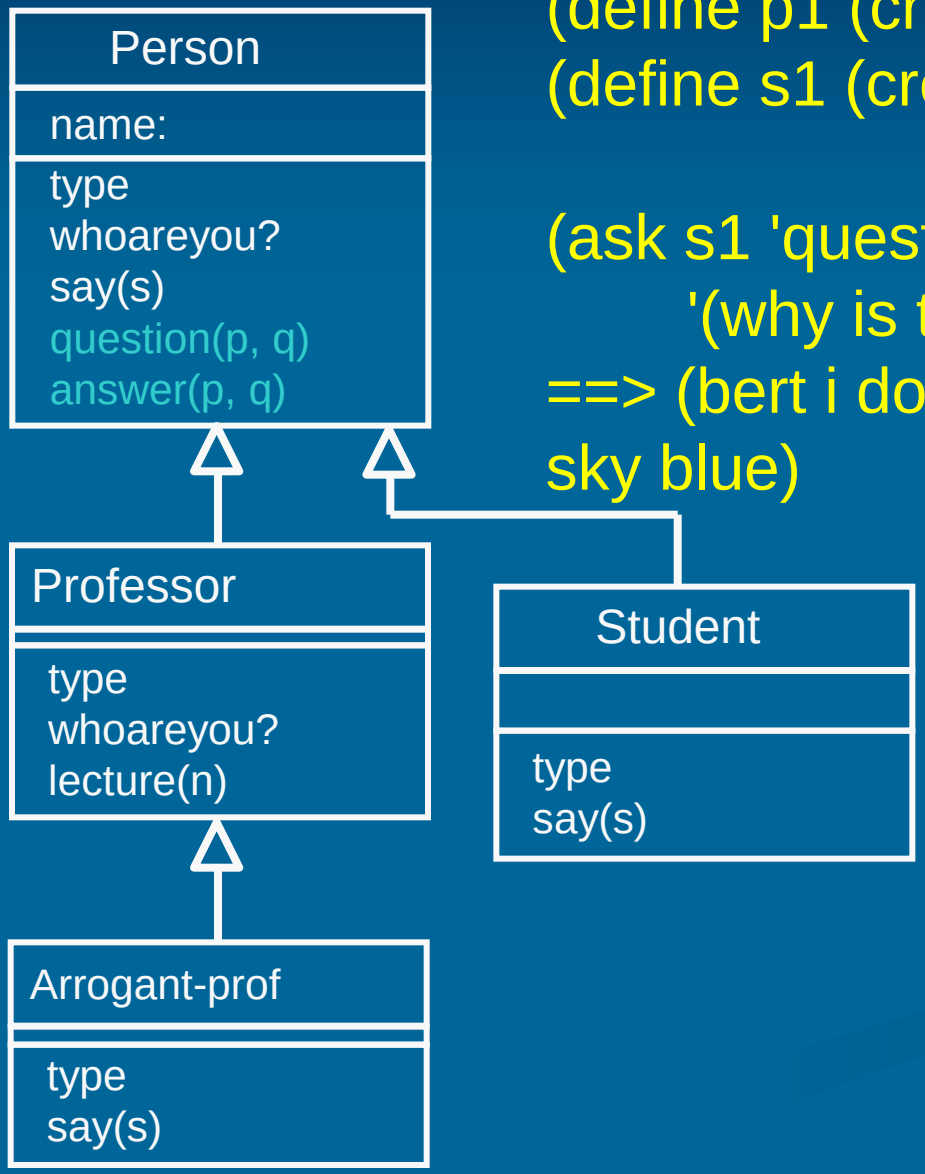

```
(define s1 (create-student 'bert))

(ask s1 'whoareyou?)
==> bert

(ask s1 'say '(i do not understand))
==> (excuse me but i do not understand)
```

Реализация Student




```
(define (create-student name)
  (create-instance student name))

(define (student self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
          ((eq? msg ‘type)
       (lambda () (type-extend 'student person-part)))
           ((eq? msg ‘say) (lambda (stuff)
                     (append '(excuse me but)
                             (ask person-part 'say stuff))))
          (else (get-method msg person-part))))))
```

Операции question и answer



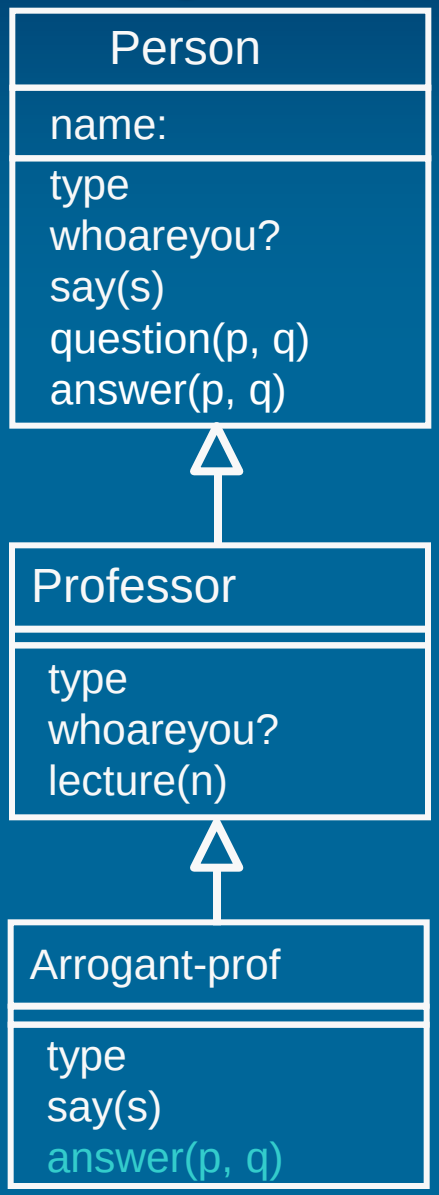

```
(define p1 (create-person 'joe))
(define s1 (create-student 'bert))

(ask s1 'question p1
        '(why is the sky blue))
==> (bert i do not know about why is the
sky blue)
```

Новая реализация Person

Person
name:
type whoareyou? say(s) question(p, q) answer(p, q)

```
(define (person self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
            ((eq? msg ‘type)
            (lambda () (type-extend 'person root-part)))
            ((eq? msg ‘whoareyou?)    (lambda () name))
            ((eq? msg ‘say)                (lambda (stuff) stuff))
            ((eq? msg ‘question)
            (lambda (of-whom query)
              (ask of-whom 'answer self query)))
            ((eq? msg ‘answer)
            (lambda (whom query)
              (ask self 'say
                (cons (ask whom 'whoareyou?)
                  (append '(i do not know about)
                          query)))))
           (else (get-method msg root-part))))))
```

Arrogant-prof – специальный ответ




```
(define s1 (create-student 'bert))
(define prof1 (create-professor 'fred))
(define ap1 (create-arrogant-prof 'perfect))

(ask s1 'question ap1
       '(why is the sky blue))
==> (this should be obvious to you obviously)

(ask prof1 'question ap1
          '(why is the sky blue))
==> (but you wrote a paper about why
    is the sky blue obviously)
```

Новая реализация Arrogant-prof

```
(define (arrogant-prof self name)
  (let ((prof-part (professor self name)))
    (lambda (msg)
      (cond
            ((eq? msg ‘type)
            (lambda () (type-extend 'arrogant-prof prof-part)))
            ((eq? msg ‘say) (lambda (stuff)
                        (append (ask prof-part 'say stuff)
                                    (list 'obviously))))
            ((eq? msg ‘answer)
            (lambda (whom query)
              (cond ((ask whom 'is-a 'student)
                          (ask self 'say
                              '(this should be obvious to you)))
                         ((ask whom 'is-a 'professor)
                          (ask self 'say
                              (append '(but you wrote a paper about)
                                          query)))
                         (else (ask prof-part 'answer whom query)))))
          (else (get-method msg prof-part))))))
```

Итоги

- Мы описываем иерархии классов
  - используя соглашение по структуре описания класса
    - для наследования структуры и поведения из суперкласса
- Управление поведением
  - “ask” к своей части
  - “ask” к self
- Дополнительное управление на основе типов объектов

Дальше больше

- Просто объекты
  - соглашение о сообщениях и операциях
  - self
- Наследование (одиночное)
  - внутренние части от суперклассов
  - в операции можно обратиться к внутренней части
  - get-method для суперкласса находит нужный метод
- **Множественное наследование**

Singer и Singing-arrogant-prof

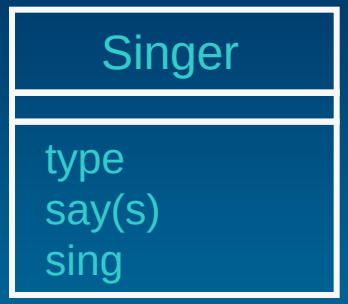
Певец -- не персона.

Он говорит, прибавляя в конце фразы "tra la la".

Умеет петь.

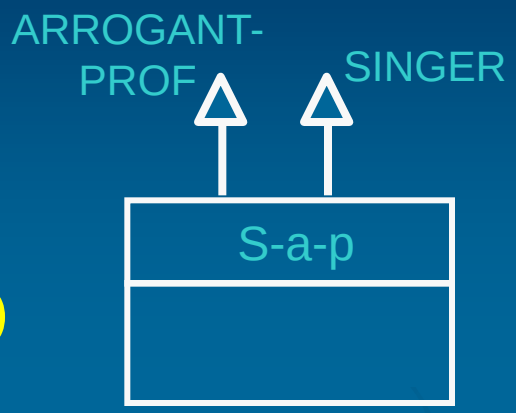
Реализация Singer

```
(define (create-singer)
  (create-instance singer))

(define (singer self)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
            ((eq? msg ‘type)
             (lambda () (type-extend 'singer root-part)))
             ((eq? msg ‘say)
             (lambda (stuff) (append stuff '(tra la la))))
             ((eq? msg ‘sing)
             (lambda () (ask self 'say '(the hills are alive))))
            (else (get-method msg root-part))))))
```



Singer – базовый класс (у него один суперкласс -- Root)

Реализация Singing-arrogant-prof




```
(define (create-singing-arrogant-prof name)
  (create-instance singing-arrogant-prof name))

(define (singing-arrogant-prof self name)
  (let ((singer-part (singer self))
        (arr-prof-part (arrogant-prof self name)))
    (lambda (msg)
      (cond
        ((eq? msg ‘type)
         (lambda () (type-extend 'singing-arrogant-prof
                                 singer-part
                                 arr-prof-part)))
        (else (get-method msg singer-part arr-prof-part))))))
```

Пример с Singing-arrogant-professor

```
(define sap1 (create-singing-arrogant-prof  'zoe))
(ask sap1 'whoareyou?)
==> (prof zoe)

(ask sap1 'sing)
==> (the hills are alive tra la la)

(ask sap1 'say '(the sky is blue))
==> (the sky is blue tra la la)

(ask sap1 'lecture '(the sky is blue))
==> (therefore the sky is blue tra la la)
```

- операция say класса arrogant-prof не работает в sap1 (нет “obviously” в конце фраз)
  - get-method находит say в классе singer, указанном первым
- если нам нужен точный контроль,
  - то следует реализовать say в классе Singing-arrogant-professor

Точка зрения реализации: Множественное наследование

- Как реализовать новый get-method?
  - просматривать экземпляры по порядку, пока не будет найден тот, который может обработать сообщение.

```
(define (get-method message object)
  (object message))
```

превращается в

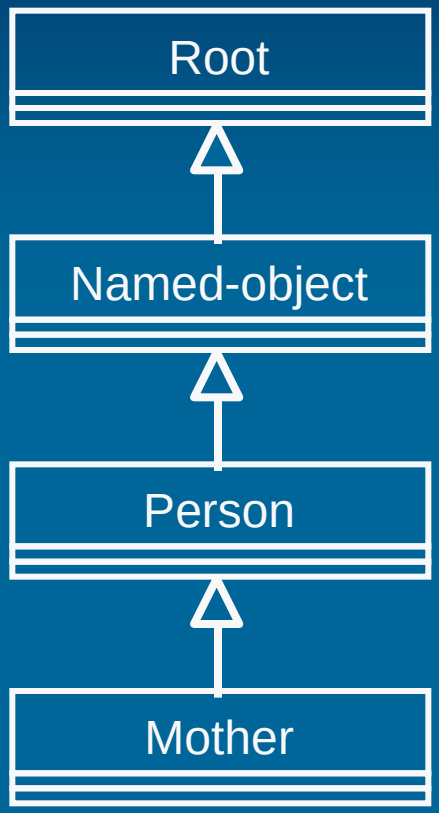
```
(define (get-method message . objects)
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method ((car objects) message)))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
  (try objects))
```

Ещё пример

- Цель примера: продемонстрировать разницу между
  - “is-a” (связью обобщения)
  - “has-a” (ассоциацией)

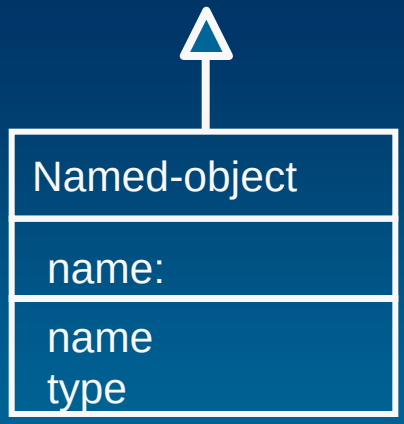
- Добавим родственные связи к person!

Родственные связи



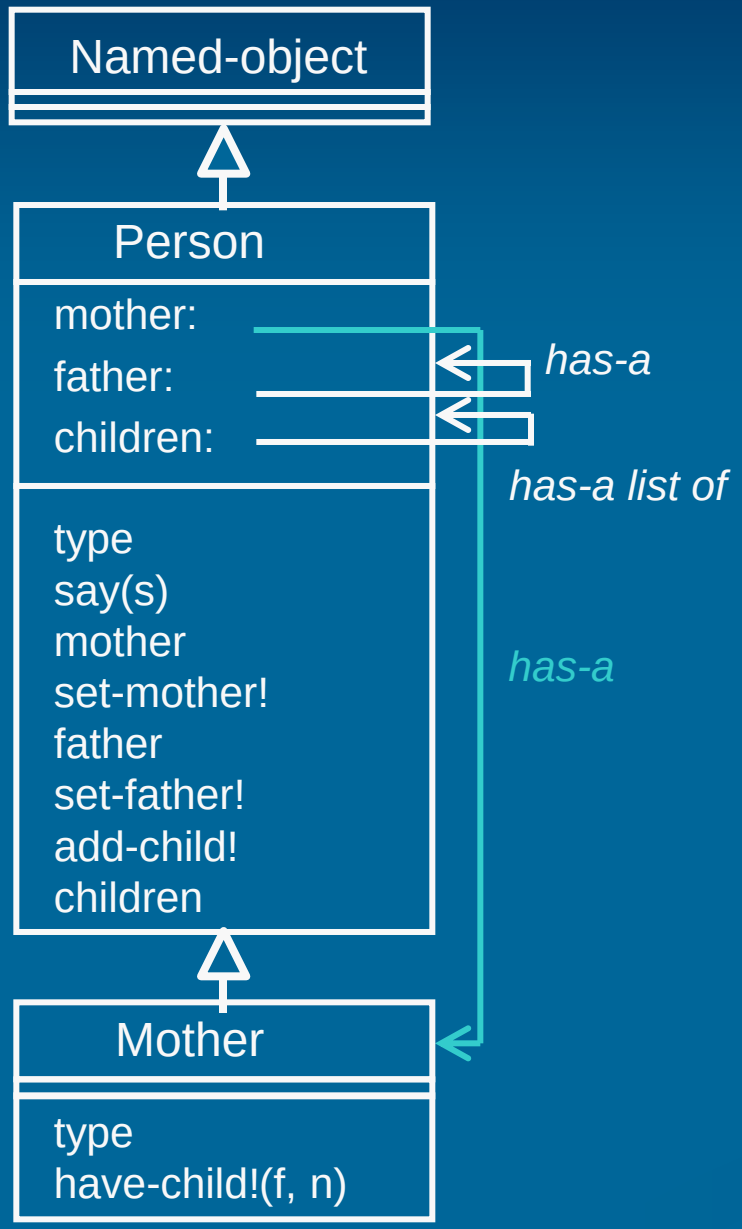
- Добавим класс Mother и рассмотрим, что получится с точки зрения
  - диаграммы классов
  - поведения
  - диаграммы объектов
  - описания классов
  - модели вычислений с окружениями

Описание Named-object



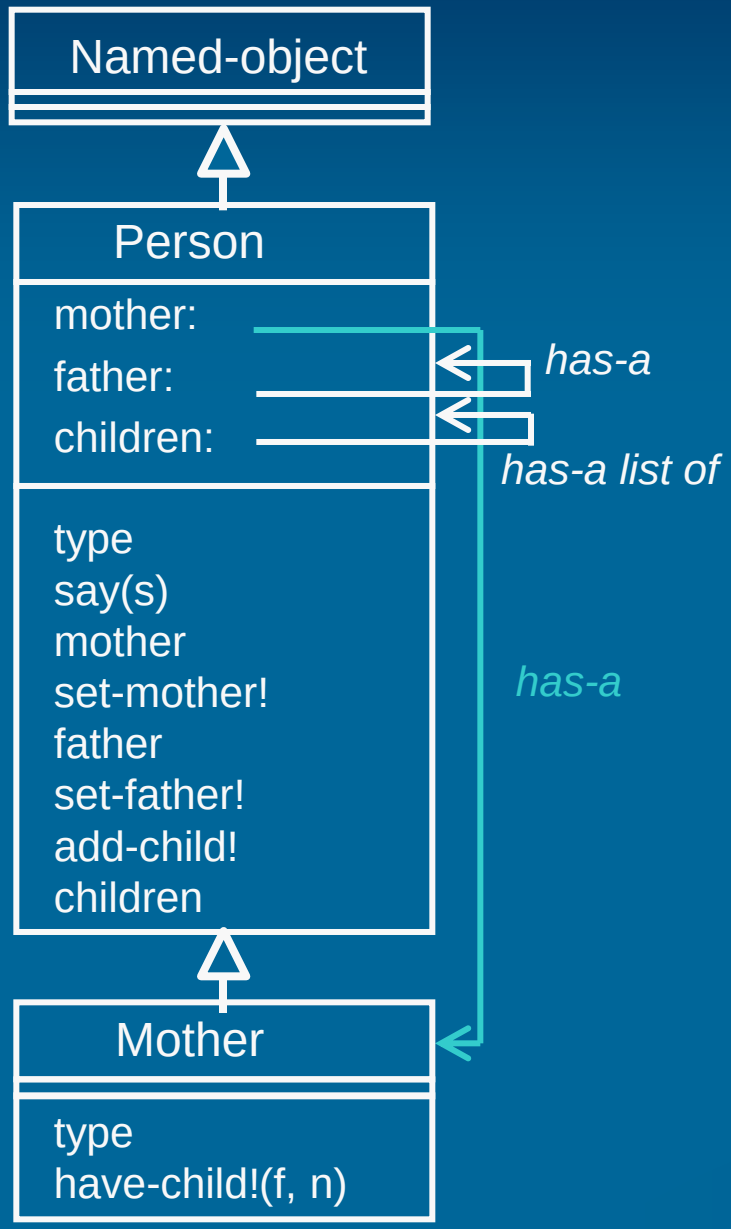

```
(define (create-named-object name)
  (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root self)))
    (lambda (msg)
      (cond
        ((eq? msg ‘type)
         (lambda () (type-extend 'named-object root-part)))
        ((eq? msg ‘name) (lambda () name))
        (else (get-method msg root-part))))))

(define (names-of objects)
    (map (lambda (x) (ask x ‘name)) objects))
```

Класс Person



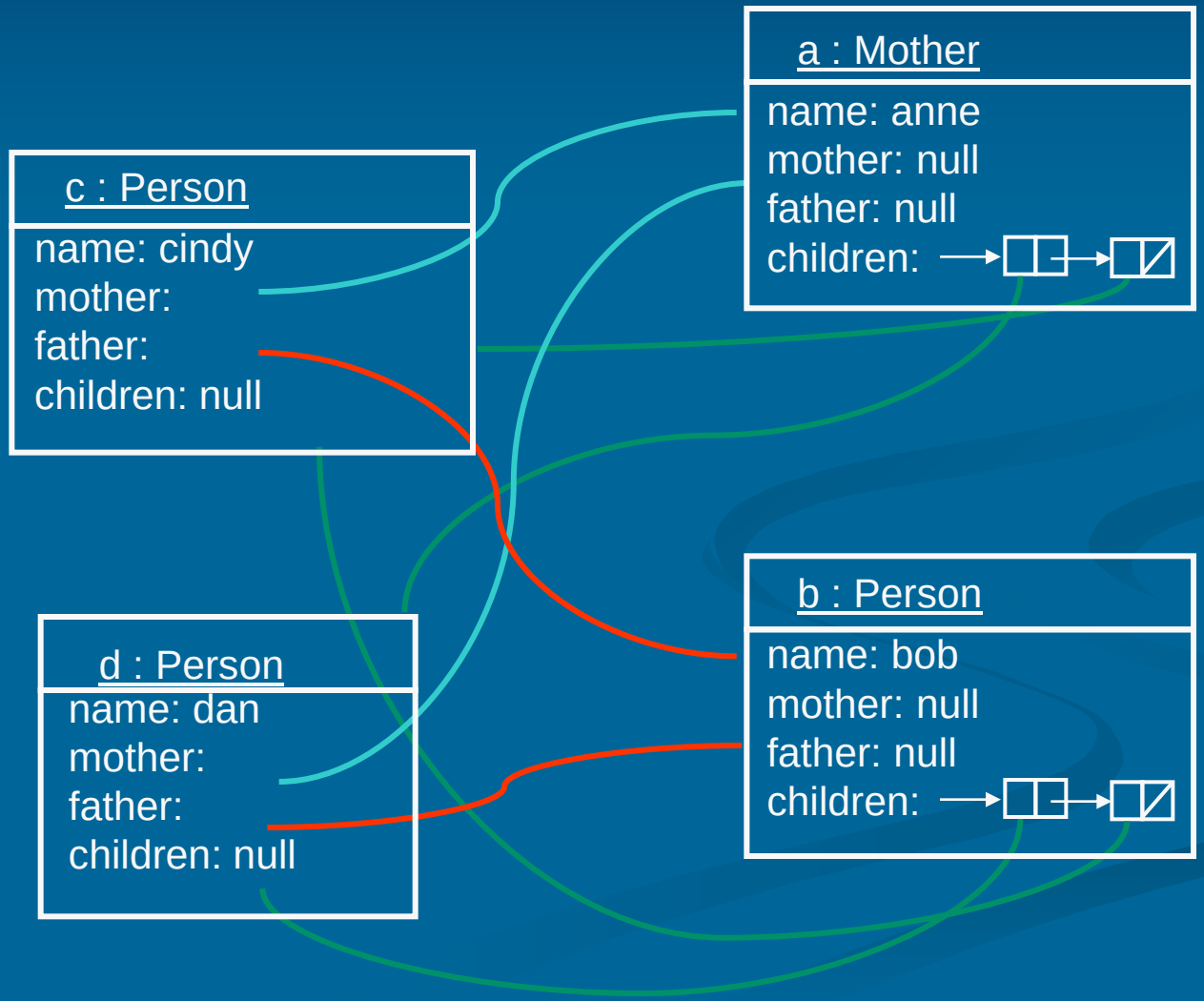
- Person подкласс Named-object
- состояние: теперь у персоны…
  - *есть* мать (экземпляр Mother)
  - *есть* отец (экземпляр Person)
  - *есть* список детей (экземпляров person)
- поведение: добавлены операции для управления состоянием
- Mother подкласс Person
  - добавлена операция have-child!(father, name)

Поведение




```
(define a (create-mother 'anne))
(define b (create-person 'bob))
(ask a 'name)  ==> anne
(ask b 'name) ==> bob
(ask a 'type) ==>
            (mother person named-object root)

(ask b 'type) ==>
            (person named-object root)

(define c (ask a 'have-child! b 'cindy))
(define d (ask a 'have-child! b 'dan))
(names-of (ask a 'children)) ==>
            (dan cindy)

(names-of (ask b 'children)) ==>
            (dan cindy)

(ask d 'name) ==> dan
(ask (ask d 'mother) 'name) ==> anne
```

Диаграмма объектов

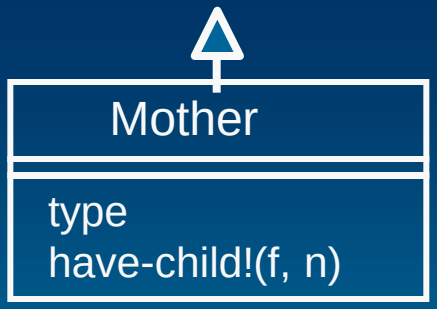


Описание класса Person

```
(define (create-person name)
  (create-instance person name))

(define (person self name)
  (let ((named-part (named-object self name))
        (mother null)
        (father null)
        (children null))
    (lambda (msg)
      (cond
        ((eq? msg ‘type) (lambda () (type-extend 'person named-part)))
        ((eq? msg ‘say) (lambda (stuff) (display stuff)))
        ((eq? msg ‘mother) (lambda () mother))
        ((eq? msg ‘father) (lambda () father))
        ((eq? msg ‘children) (lambda () children))
        ((eq? msg ‘set-mother!) (lambda (mom) (set! mother mom)))
        ((eq? msg ‘set-father!) (lambda (dad) (set! father dad)))
        ((eq? msg ‘add-child!)   (lambda (child)
                                   (set! children (cons child children))
                                   child))
        (else (get-method msg named-part))))))
```

Person
mother: father: children:
type say(s) mother set-mother! father set-father! add-child! children

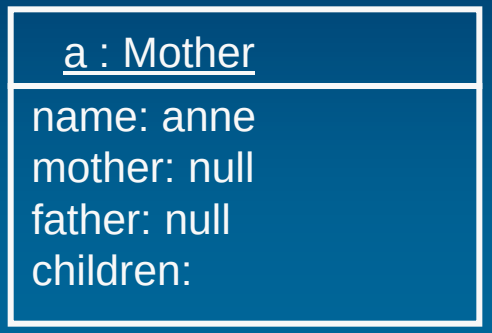
Описание класса Mother

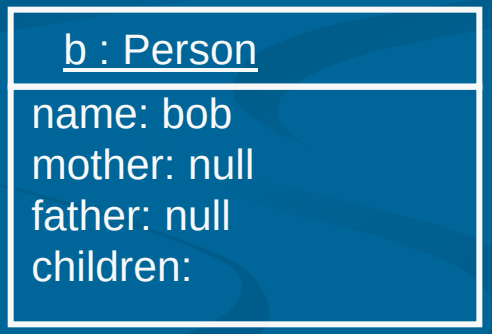



```
(define (create-mother name)
  (create-instance mother name))

(define (mother self name)
  (let ((person-part (person self name)))
    (lambda (msg)
      (cond
        ((eq? msg ‘type) (lambda () (type-extend 'mother person-part)))
        ((eq? msg ‘have-child!)
         (lambda (dad child-name)
           (let ((child (create-person child-name)))
             (ask child 'set-mother! self)
             (ask child 'set-father! dad)
             (ask self 'add-child! child)
             (ask dad 'add-child! child))))
        (else (get-method msg person-part))))))
```

Добавление объекта-ребёнка

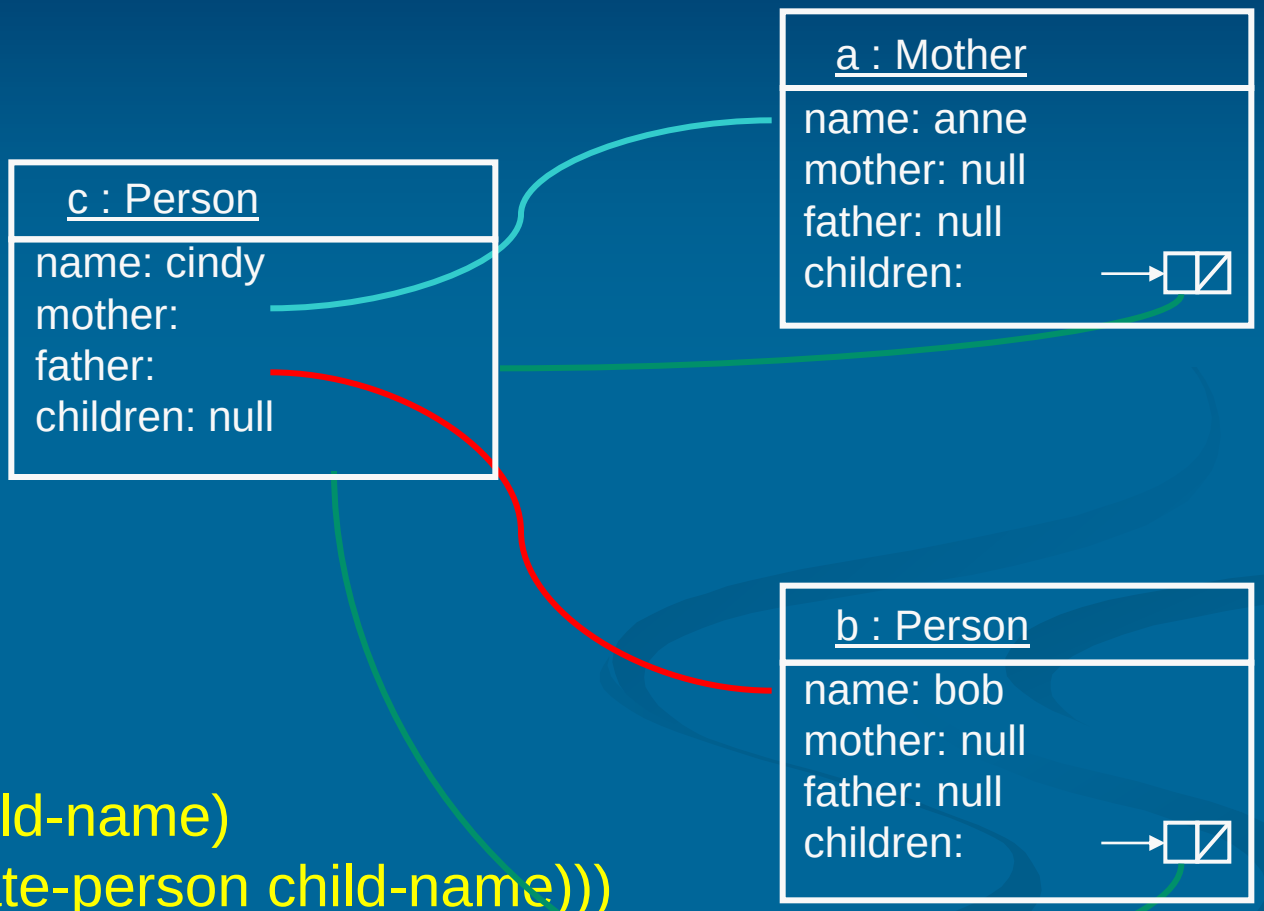
```
(define a (create-mother 'anne))
(define b (create-person 'bob))
```






```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
    (ask child 'set-father! dad)
    (ask self 'add-child child)
    (ask dad 'add-child child)))
```

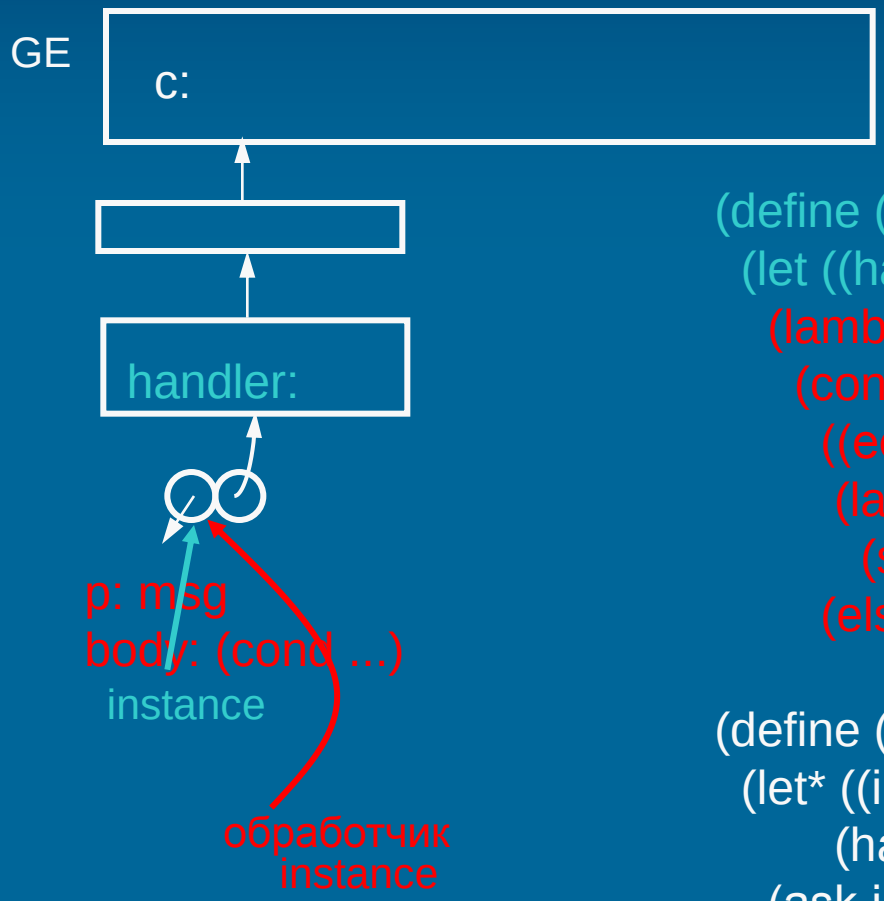
Добавление объекта-ребёнка




```
(lambda (dad child-name)
  (let ((child (create-person child-name)))
    (ask child 'set-mother! self)
    (ask child 'set-father! dad)
    (ask self 'add-child child)
    (ask dad 'add-child child) ))
```

В модели окружений

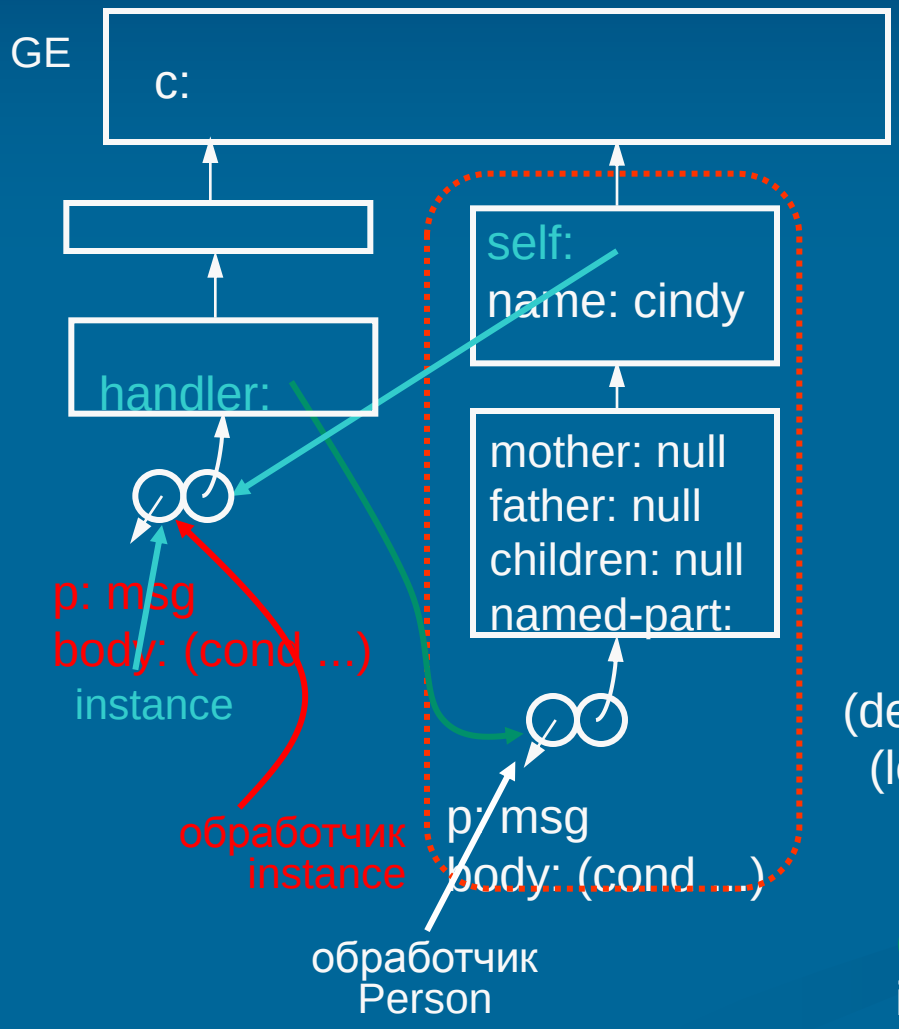
(create-person ‘cindy) =>
(create-instance person ‘cindy)




```
(define (make-instance)
  (let ((handler #f))
    (lambda (msg)
      (cond
        ((eq? msg ‘set-handler!)
         (lambda (handler-proc)
           (set! handler handler-proc)))
        (else (get-method msg handler)))) ))

(define (create-instance maker . args)
  (let* ((instance (make-instance) )
         (handler (apply maker instance args) ))
    (ask instance ‘set-handler! handler)
    instance))
```

В модели окружений

(create-person ‘cindy) =>
(create-instance make-person ‘cindy)



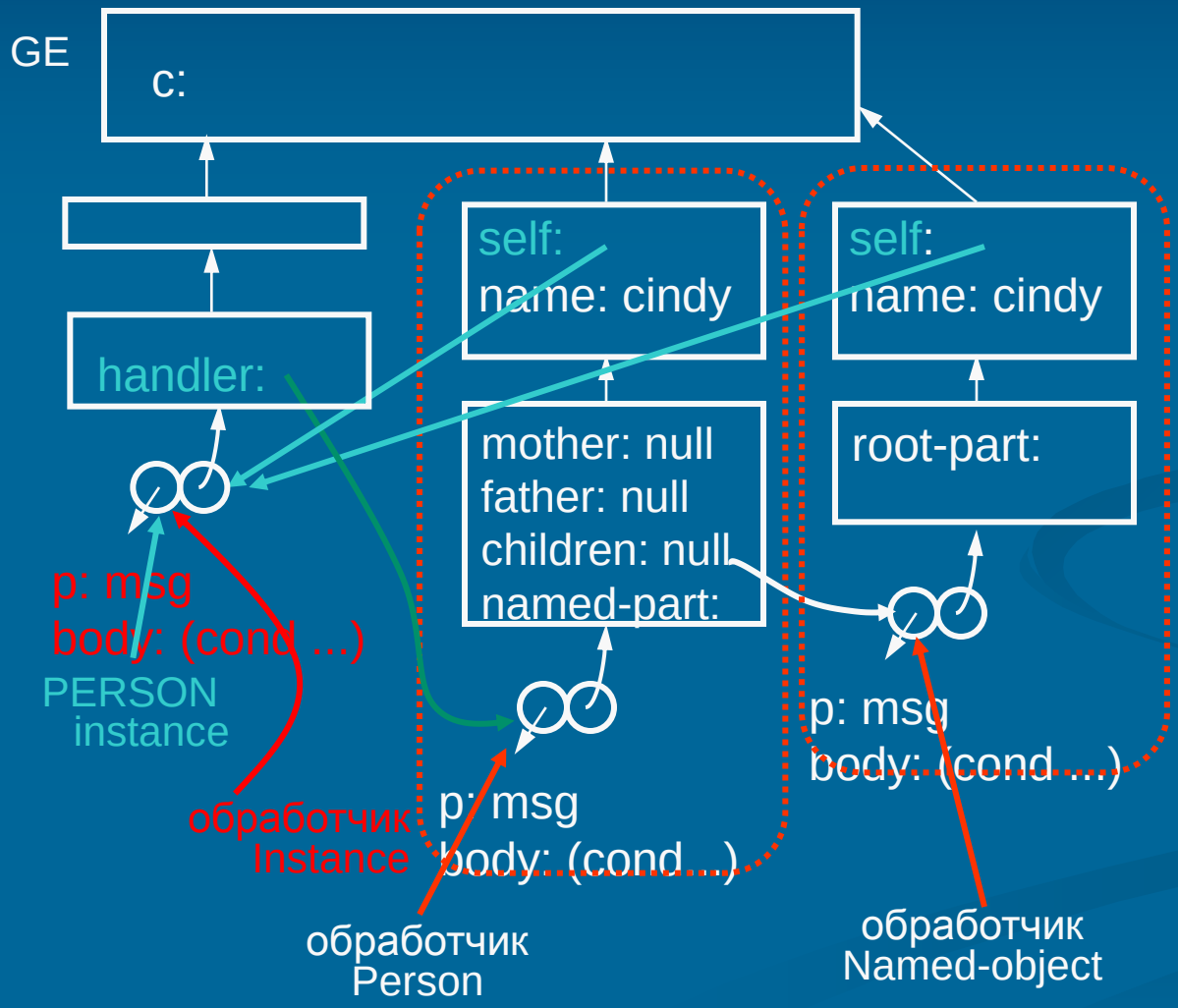

```
(define (person self name)
  (let ((named-part (make-named-object
self name))
        (mother nil)
        (father nil)
        (children nil))
    (lambda (message)
      (case message …))))
```

```
(define (create-instance maker . args)
  (let* ((instance (make-instance) )
         (handler (apply maker
                         instance args) ))
    (ask instance ‘set-handler! handler)
    instance))
```

В модели окружений

(create-person ‘cindy) =>
(create-instance person ‘cindy)

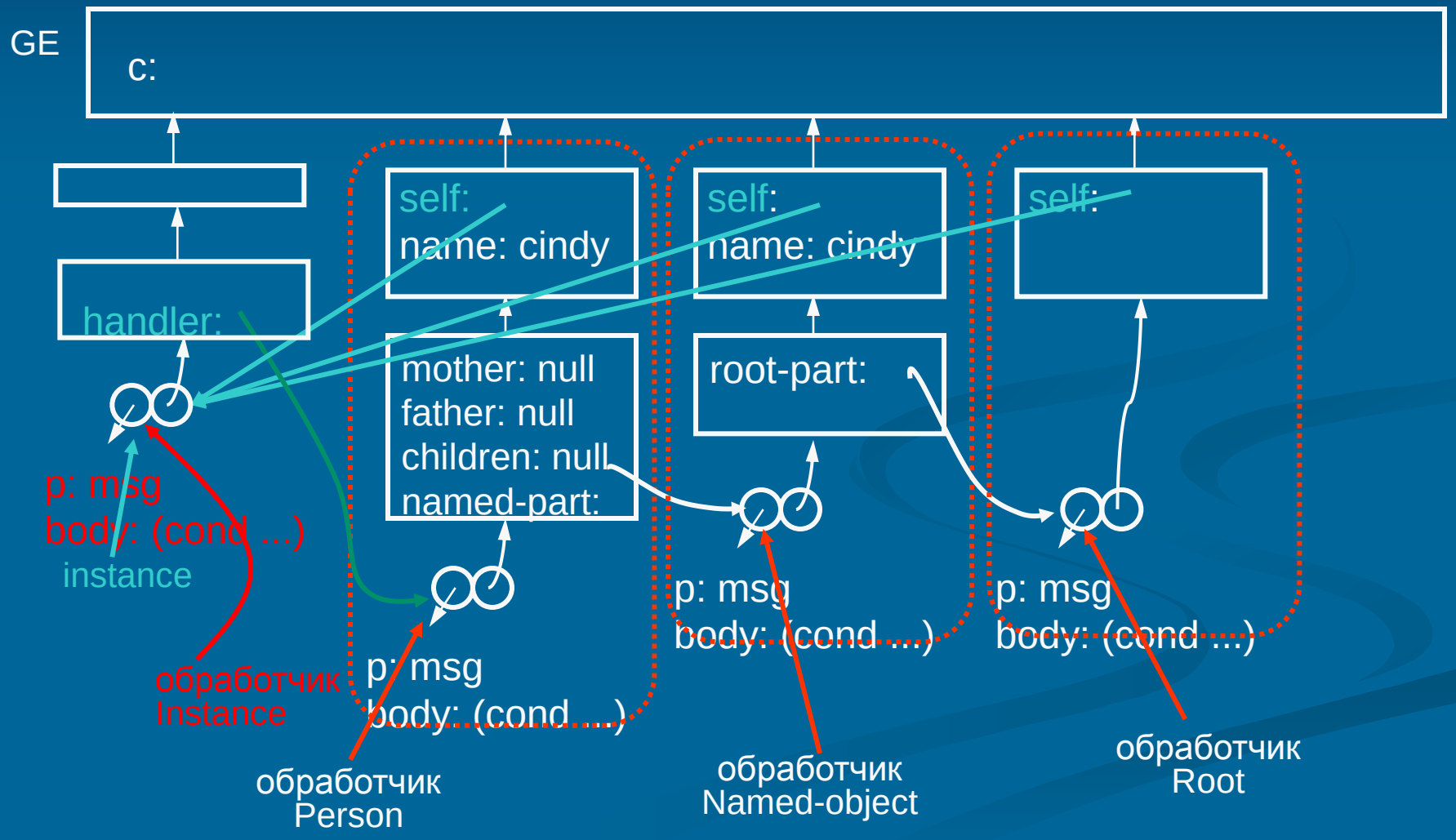
```
(define (named-object self name)
  (let ((root-part (root self)))
    (lambda (msg)
      ( cond…))))
```



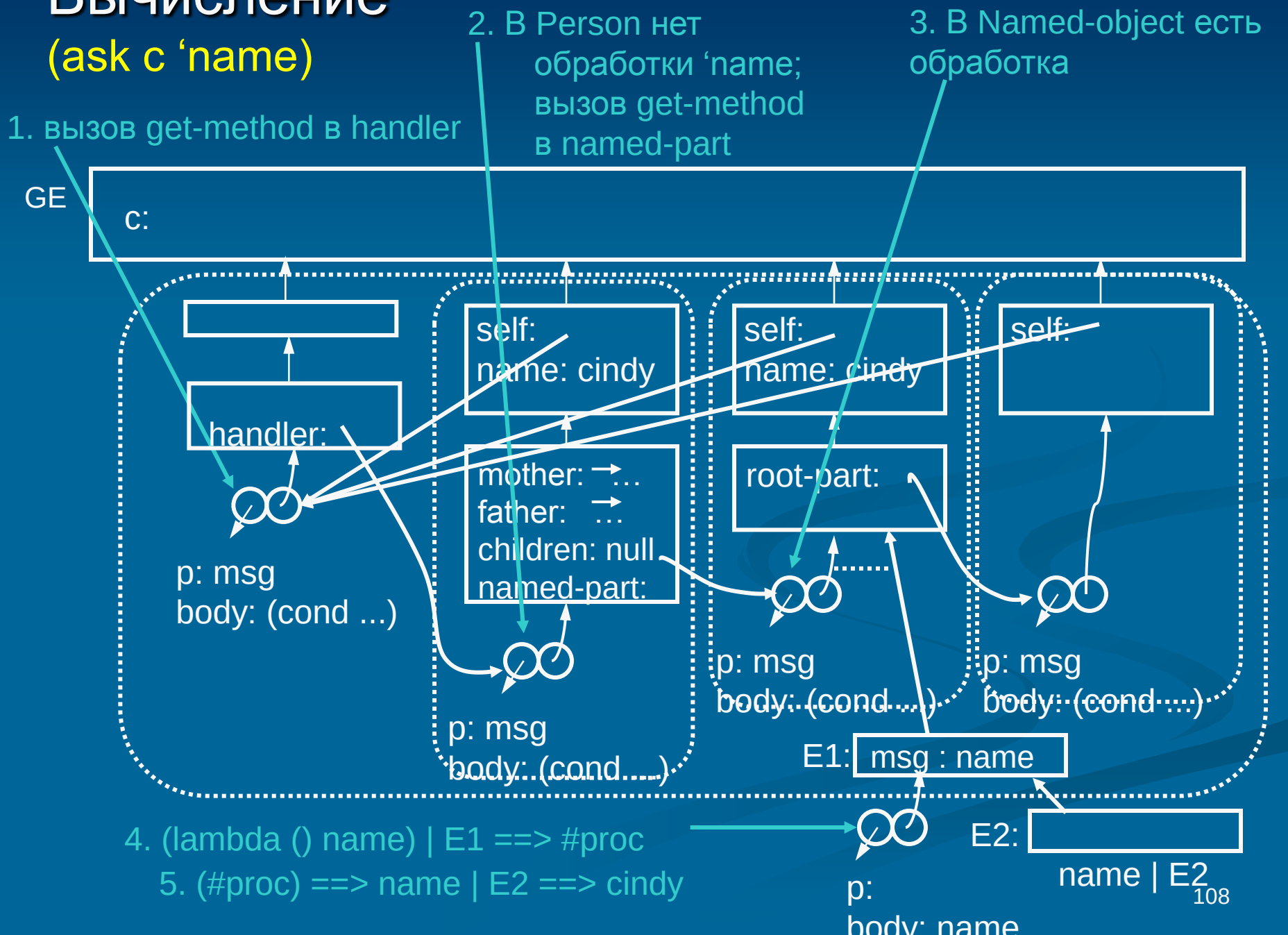
В модели окружений

(create-person ‘cindy) =>
(create-instance person ‘cindy)

```
(define (root self)
  (lambda (msg)
    ( cond…))))
```



Вычисление
(ask c ‘name)



Итог лекции 7

- Классы: описывают общую структуру и поведение
- Экземпляры содержат:
  - «заглавное звено» (self)
  - цепочку обработчиков
- Иерархия классов
  - Наследование структур и поведения суперклассов
  - Множественное наследование: правила поиска операций
- ООП
  - **Модель:** диаграммы классов и объектов
  - **Использование:** способы описания классов и создания экземпляров
  - **Реализация:** отображение понятий ООП в Scheme