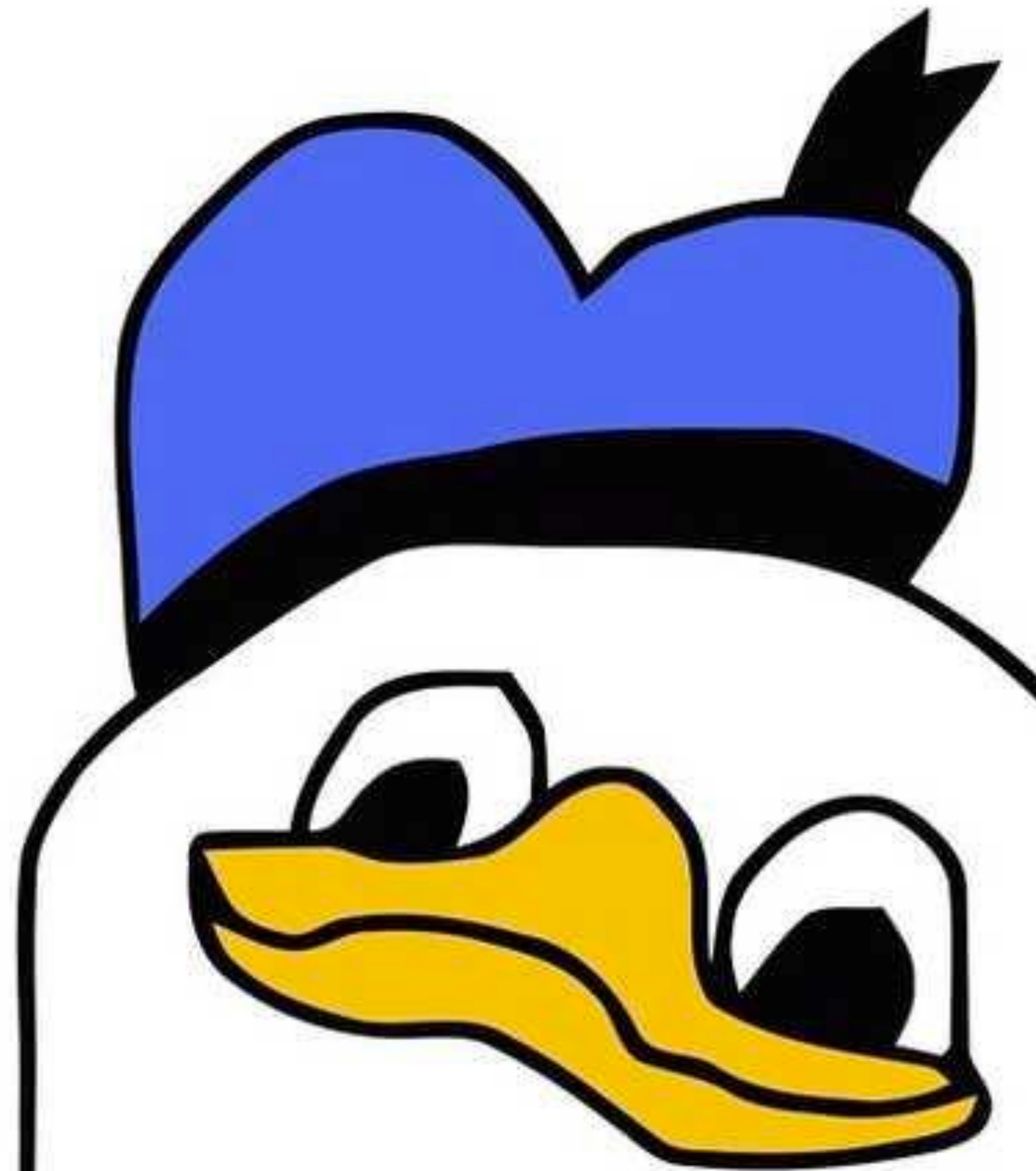
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция № 2 / 11
05.12.2017 г.

Всё очень



плохо

MemesMix.net

Модульность, повторное использование кода?

«Проблема с ОО-языками в том, что они тащат с собой весь этот невидимый багаж. Вам нужен банан, а вам дают обезьяну, держащую банан, и в довесок к ней джунгли целиком»

– Джо Армстронг, создатель языка Erlang

Всё есть объект? Полиморфизм по одному типу?

«ООП кажется мне технически необоснованным. Оно пытается осуществить декомпозицию мира на интерфейсы, которые варьируются по одному типу. Чтобы иметь дело с реальными задачами, нужны множественные алгебры — семейства интерфейсов, захватывающих несколько типов. Я нахожу ООП и философски необоснованным. Оно утверждает, что всё есть объект. Даже если это так, это не очень интересно: сказать, что всё есть объект — значит не сказать ничего»

– Александр Степанов, создатель STL

Модульность, говорите?

«Популярность ООП в больших компаниях связана с тем, что оно соответствует их стилю разработки ПО. В них работают большие (и часто меняющиеся) группы программистов-посредственностей. ООП создаёт дисциплину, чтобы никто не мог наломать слишком много дров. Цена такова, что в программах слишком много протоколов (*базовых классов*) и дублирования. Но это не очень высокая цена для крупных компаний, поскольку их ПО и так перегружено, да и дублирования в нём всё равно предостаточно»

– Пол Грэм, венчурный капиталист, создатель Viaweb

Всё есть объект?

«ООП ставит во главу угла *Существительные*. Зачем из кожи вон лезть, чтобы поставить одну из частей речи на пьедестал? Почему одна концепция должна превалировать над другой? ООП отнюдь не уменьшает важность глаголов в том способе, котором мы думаем. Оно даёт странный, ассиметричный взгляд на вещи»

– Стив Йегги, разработчик и блоггер

«ОО-концепция показала себя ценной в разработке графических систем, GUI и в моделировании. Но … значительные преимущества в других областях не проявились. Полезно понять почему.

… ООП слишком часто называлось Единственно Верным Решением для управления сложностью.

… ОО-языки делают абстракцию лёгкой, даже слишком. Они приветствуют архитектуры с толстым слоем «клея» и тщательно созданными слоями. Это хорошо, когда задача сложна… но даёт неприятный эффект, если разработчики делают простые вещи сложным способом, просто потому что могут.

… Правила простоты, ясности и прозрачности нарушаются повсеместно…

… Одна из главных сложностей в дизайне в стиле Unix — как соединить вместе преимущества отделённости (упрощение и обобщение задач…) с преимуществом тонкого уровня «клея» и неглубоких, плоских, прозрачных иерархий кода и дизайна».

– Эрик Рэймонд, разработчик Unix, сторонник OpenSource

Всё есть объект?

«Объектные системы — слишком упрощённые модели реального мира. ООП неспособно правильно моделировать время, что становится проблемой, поскольку программные системы приобретают всё больший параллелизм».

– Рич Хики, создатель языка Clojure

ВИДЫ СЛОЖНОСТИ

- Сложность, присущая самой задаче *(inherent complexity)*:
  - Сложный алгоритм.
  - Сложные, вариабельные структуры данных.
- Побочная сложность *(incidental complexity)*:
  - Порождается теми средствами, которые мы используем для решения задач (языки, трансляторы и т.д.)
  - Обратная сторона «простоты».

```
Foo *myFunc(...);  // C++
```

- Простота:
  - Один и тот же синтаксис указателей на объекты в куче и не в куче.
  - Легко работать с указателями.
- Сложность:
  - Надо всегда понимать, кто и когда будет освобождать память.
  - Ответственность за управление памятью лежит на разработчике.

Нет автоматического управления памятью.

```
Date foo(...);  // Java
```

- Простота:
  - Объект всегда лежит в куче.
  - Сборщик мусора.
  - Тот же синтаксис для изменяемых и неизменяемых объектов.

- Сложность:
  - Можем ли мы «запомнить» значение?
  - Если мы изменим состояние объекта, на кого ещё это повлияет?

Нет автоматического управления временем.

ЭФФЕКТЫ ПОБОЧНОЙ СЛОЖНОСТИ

- Сложно разобраться в большой программе.
- Сложно понять, что будет затронуто вносимыми нами изменениями.
- При добавлении параллелизма всё становится ещё хуже.

ЧИСТЫЕ ФУНКЦИИ

- Принимают и возвращают *значения*.
- Нет побочных эффектов.
- Эффект времени исключён.
- Идемпотентность (те же входные данные — тот же результат).
- Легко понимать, тестировать, изменять и соединять вместе.

НО НЕ ВСЁ ВЫРАЖАЕТСЯ ЧЕРЕЗ ЧИСТЫЕ ФУНКЦИИ!

- Google(x) – не чистая функция.
- Реальные программы больше похожи на процессы, контактирующие с миром (взаимодействие с пользователем, API, БД и т.д.)

ВРЕМЯ В ООП

- Его нет! Хотя объекты претендуют на моделирование мира.
- Равенство значений подменяется равенством ссылок.
- Нельзя взять ссылку и спокойно с ней работать.

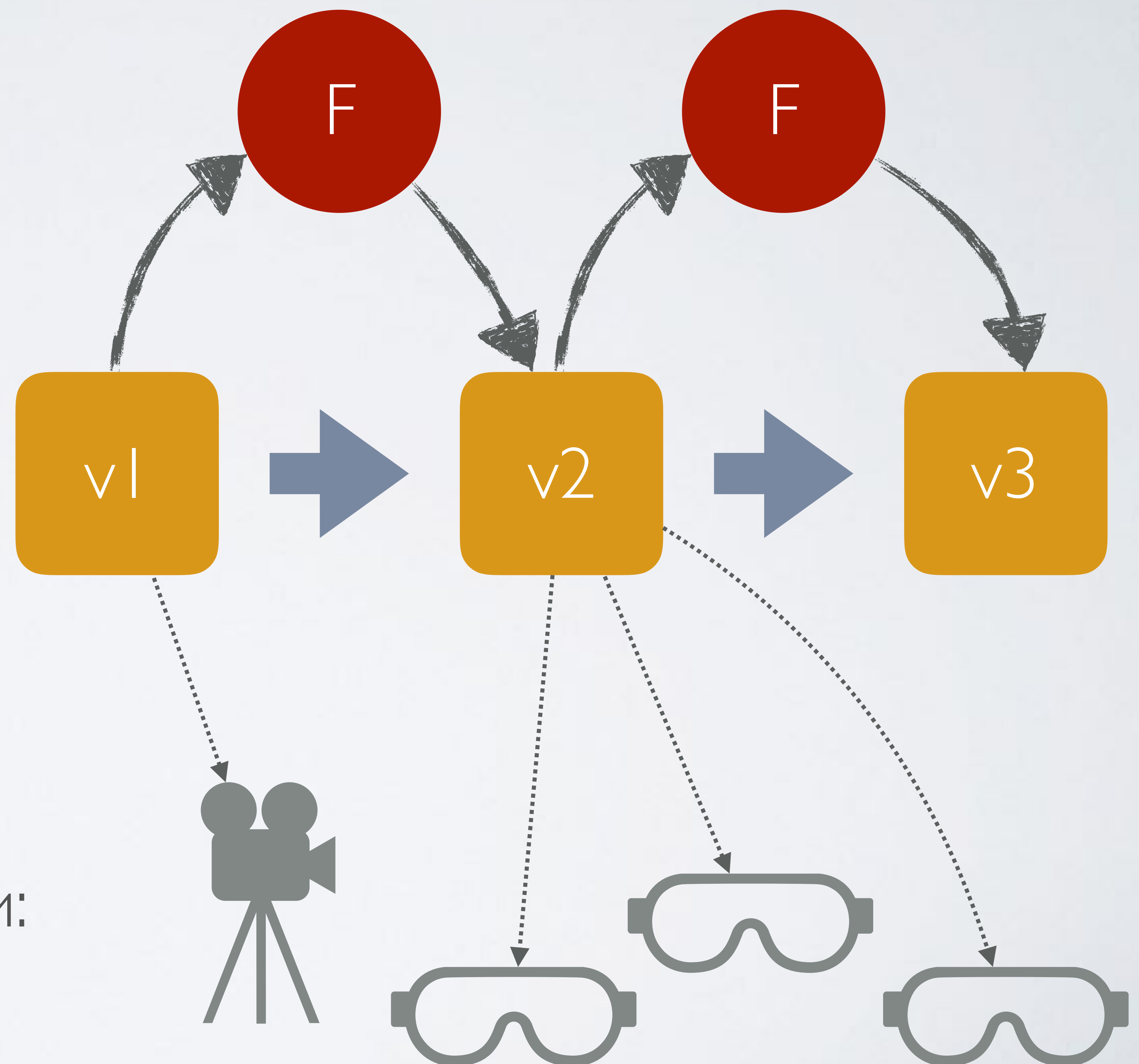
```
Foo *a1 = new Foo();

a2 = a1;

a2.changeSomething();

if (a1 == a2) {
  // Объект тот же, но другой
  // Начальное состояние утеряно
}
```

«НЕЛЬЗЯ ВОЙТИ В ОДНУ РЕКУ ДВАЖДЫ»

- На самом деле все сущности являются неизменяемыми.
- Будущее — функция прошлого, оно не меняет его.
- Процесс создаёт будущее на основе прошлого.
- Идентичность — ряд значений, связанных отношением причинности.
    - Полезная штука, но из неё не следует, что существует изменяемая сущность.
- Время — атомарная последовательность событий в процессе.

События процесса (чистые функции)

Идентичность
(последовательность
неизменяемых состояний)

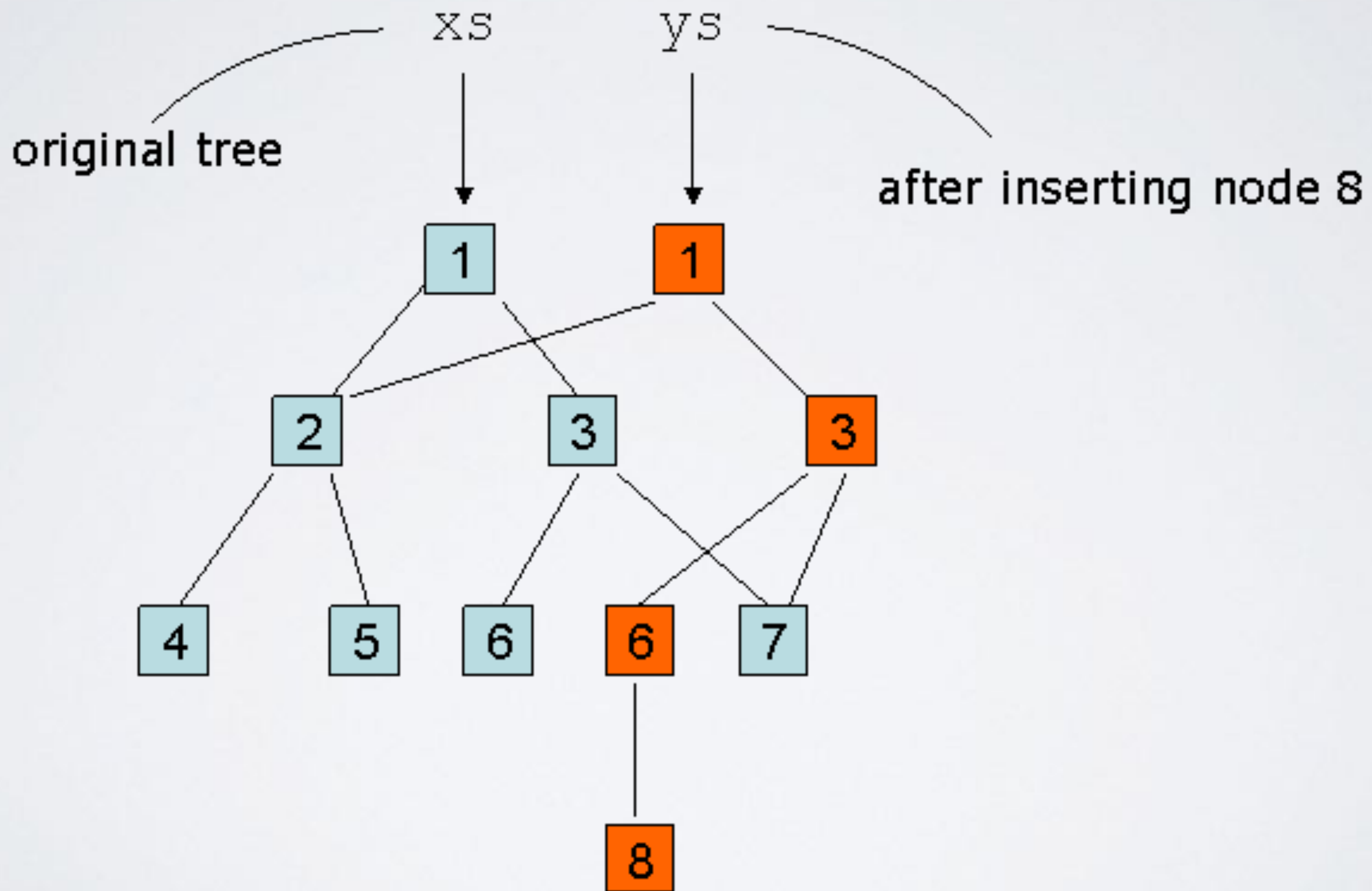


Наблюдатели:

ПЕРСИСТЕНТНЫЕ СТРУКТУРЫ ДАННЫХ

- Неизменяемость (Value Object).
- Переиспользование структуры («новые» значения переиспользуют структуру старых, минимизируя копирование).
- Порождение новых значений никак не влияет на старые и на те места, где они сохранены.

ДЕРЕВЬЯ!



- Деревья используются для всех структур данных!
- Неиспользуемые старые значения очищаются сборщиком мусора.
- Работает достаточно быстро. Внутри функции **F** (порождающей новое состояние) можно использовать и изменяемые данные.
- Реализации:
  - Clojure.
  - Scala.
  - Immutable.js (для JavaScript).
  - Immer (для C++!).

- Вся сложность переместилась в реализацию персистентных структур данных.

- Отказ от инкапсуляции — состояние проще моделировать вложенной структурой из vector/map/set/…

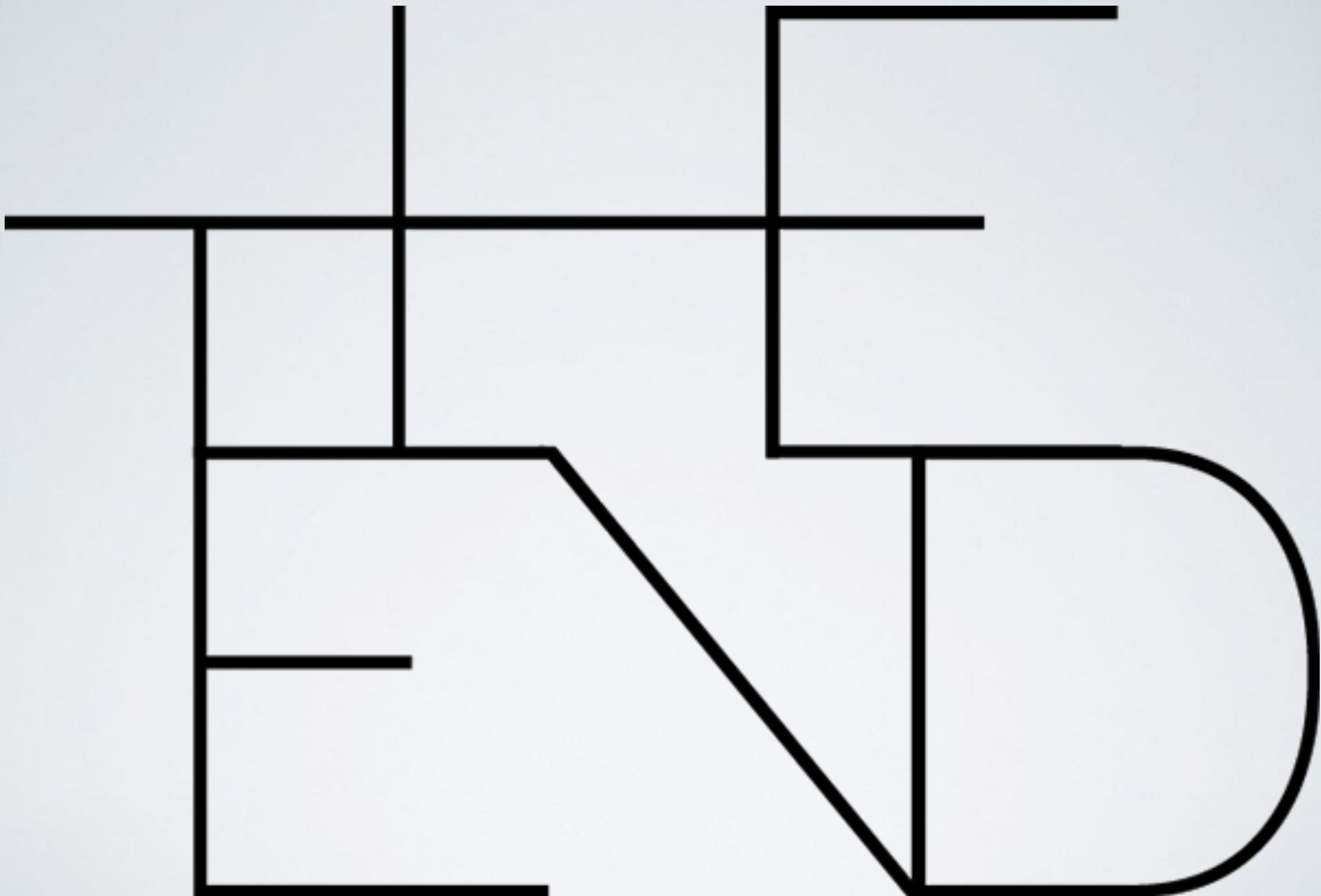
- Методы заменяются на чистые функции.

- Изменяется диспетчеризация.

Мультиметоды на Clojure

```
;; Определение мультиметода. Он выбирает реализацию
(defmulti greeting (fn [x] (get x "language")))

;; Реализация для English
(defmethod greeting "English" [params] "Hello!")

;; Реализация для French
(defmethod greeting "French" [params] "Bonjour!")

;; Реализация по умолчанию
(defmethod greeting :default [params]
 (throw (IllegalArgumentException.
          (str "I don't know the " (params "language") " language"))))

(def english-map {"id" "1", "language" "English"})
(def  french-map {"id" "2", "language" "French"})
(def spanish-map {"id" "3", "language" "Spanish"})

=> (greeting english-map)
"Hello!"
=> (greeting french-map)
"Bounjour!"
=> (greeting spanish-map)
 java.lang.IllegalArgumentException: I don't know the Spanish language
```

```clojure
(defmulti bat
  (fn ([x y & xs]
    (mapv class (into [x y] xs)))))

(defmethod bat [String String] [x y & xs]
  (str "str: " x " and " y))
(defmethod bat [String String String] [x y & xs]
  (str "str: " x ", " y " and " (first xs)))
(defmethod bat [String String String String] [x y & xs]
  (str "str: " x ", " y ", " (first xs) " and " (second xs)))
(defmethod bat [Number Number] [x y & xs]
  (str "number: " x " and " y))

;; Примеры вызова

(bat "mink" "stoat")
;; => "str: mink and stoat"

(bat "bear" "skunk" "sloth")
;; => "str: bear, skunk and sloth"

(bat "dog" "cat" "cow" "horse")
;; => "str: dog, cat, cow and horse"

(bat 1 2)
;; => "number: 1 and 2"
```


THE END