プログラミング言語設計論

2014年度

第3回: オブジェクト指向プログラミング(1)

担当: 増原英彦

問題 (20分)

1. 「オブジェクト指向プログラミング(以下OOP)」などで言われる「指向(orientation)」の意味を定義せよ
2. オブジェクト指向言語を特徴付ける機能を列挙せよ
3. オブジェクト指向言語と抽象データ型に共通する機能を列挙せよ

それが無いとOOP言語とは言えなくなるもの

指向 (orientation)

- 定義: それを中心にして物事を考えること
- 用例: オブジェクト指向プログラミング = 問題領域に登場するモノを中心にしてプログラムを作ること

= 者 + 物 [所93]

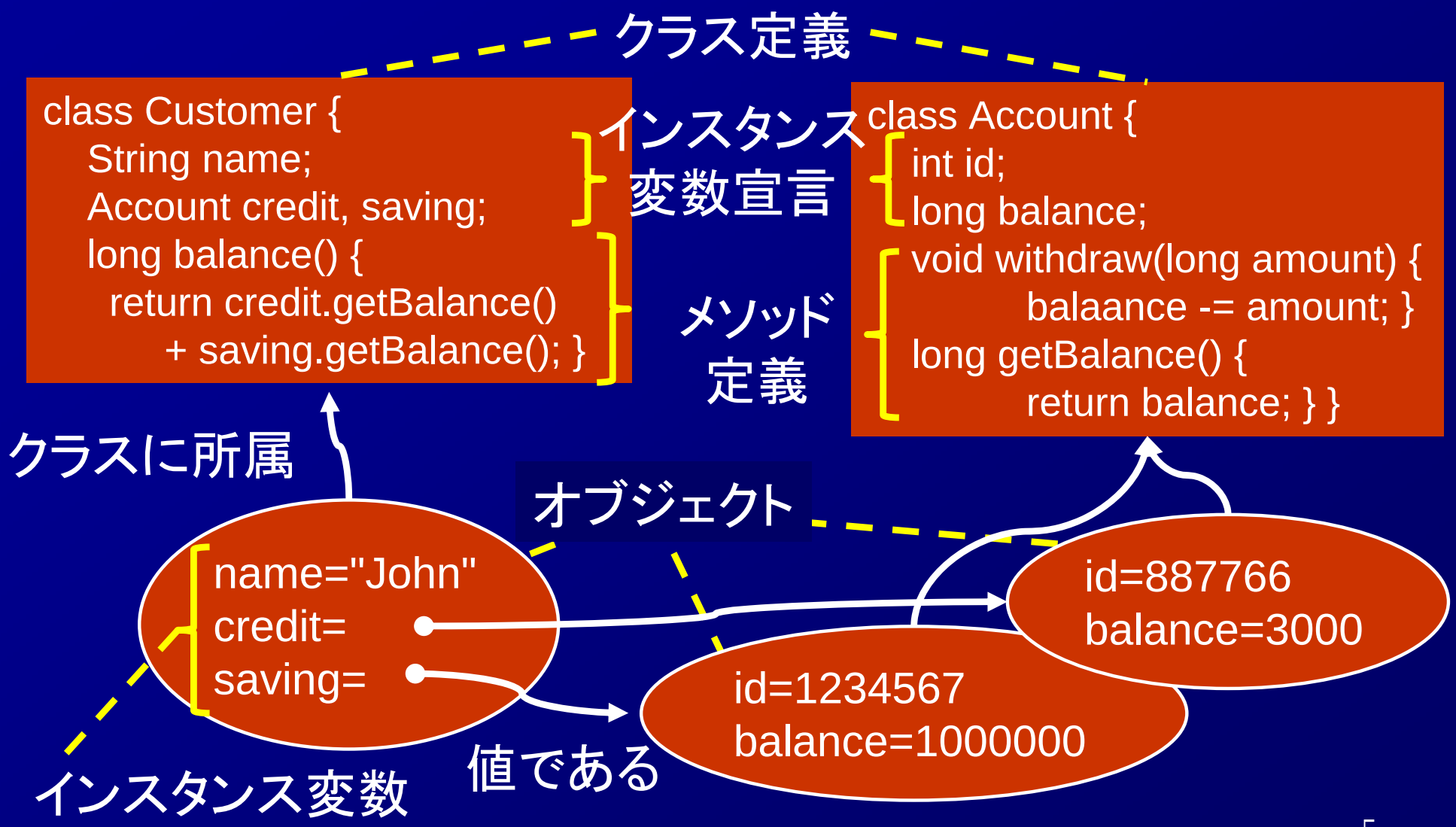
注: ○「関数型プログラミング」
×「関数指向プログラミング」

注: プログラミングに限らない
例: オブジェクト指向設計

典型的なOOP言語の例
例題: 銀行口座と顧客

- モノ
  - (銀行: 顧客と口座がある)
  - 顧客: 名義、当座口座と定期口座を持つ
  - 各座: 口座番号と残高がある
- 振舞い
  - 顧客の合計残高を求める
  - 口座から引き出す
  - 口座の残高を求める

典型的なOOP言語のプログラム



典型的なOOP言語
クラスとオブジェクト

- クラス: 以下をまとめた定義
  (= プログラムとして書かれる)
  - メソッド定義: オブジェクトの振舞いを決める
  - インスタンス変数宣言: オブジェクトが持つ状態を決める
- オブジェクト: (実行時に作られる)
  - 値である (変数に格納される)
  - クラスに所属する
  - インスタンス変数を持つ

典型的なOOP言語
メソッド呼出

```
class Customer {
    String name;
    Account credit, saving;
    long balance() {
      return credit.getBalance()
          + saving.getBalance(); } }
```

```
class Account {
    int id;
    long balance;
    void withdraw(long amount) {
            balaance -= amount; }
    long getBalance() {
            return balance; } }
```

john.balance() => 1003000

name="John"
credit=
saving=

メソッド呼出:

- レシーバのクラスから探す
- 関数として実行
  - インスタンス変数が見える

典型的なOOP言語
メソッド呼出

- メソッド呼出: john.balance()
  - 関数呼出のようなもの
  - レシーバ: 0番目の引数
- メソッドディスパッチ (dispatch):
  - レシーバが所属するクラスを見つける
  - クラスの中から同名のメソッド定義を探す
- メソッドの実行環境
  - インスタンス変数が見える
  - 「自身」を指す疑変数がある (this, self)

典型的なOOP言語: 継承

```
class FixedDeposit extends Account {
    Date maturity;
    void withdraw(long amount) {
      if (maturity.isExpired())
          super.withdraw(amount);
      else
          error; } }
```

継承

```
class Account {
    int id;
    long balance;
    void withdraw(long amount) {
            balaance -= amount; }
    long getBalance() {
            return balance; } }
```

saving

credit

id=1234567
balance=1000000

id=887766
balance=3000

credit.balance()=>3000
saving.balance()=>1000000

credit.withdraw(1000) => OK
saving.withdraw(1000) => error!

典型的なOOP言語: 継承

```
class FixedDeposit extends Account {
    Date maturity;
    void withdraw(long amount) {
      if (maturity.isExpired())
          super.withdraw(amount);
      else
          error; } }
```

saving

- 継承
  - あるクラスに要素を追加・上書きしたクラスを作る
  - 親/super・子/subクラス
- メソッドディスパッチ
  - 所属するクラスから探す
  - ない場合は親クラスを辿る
- super呼出
  - 親クラスの定義を呼び出す

credit.balance()=>3000

saving.balance()=>1000000

credit.withdraw(1000) => ok

saving.withdraw(1000) => error!

言語の機能と「指向」

- OO = モノを中心に考える

↔

- OOP言語の機能: クラス・メソッド・インスタンス変数・メソッド呼出・継承・再定義・・・

何故OOを可能にするのか?

- OOにとって必須なのか?

OOP言語の性質: カプセル化

- 1つのモノに注目する
  - 自他の区別をする: どこまでが「自分」か?
    → 「オブジェクト」が1つの値として扱える
    「カプセル化(encapsulation)」と言う
  - 他のモノの詳細には関心がない:
    自身のインスタンス変数にアクセス
    他のモノはメソッド呼出を通して働き掛け
    → 抽象データ型に類似

OOP言語の性質: カプセル化

- 1つのモノに注目
  - 自他の区別をする
    → 「オブジェクト」
    「カプセル化(enca
  - 他のモノの詳細には関心がない:
    自身のインスタンス変数にアクセス
    他のモノはメソッド呼出を通して働き掛け
    → 抽象データ型に類似

```
class Point {
  int x, y;
  boolean equals(Point other) {
    return this.x == other.x
        && this.y == other.y; } }
```

これは1つのモノに
注目しているのか?

OOP言語の性質: 多相性

- 1つ1つのモノは異なる振舞をする
  - 同じ働き掛けをしてもモノによって反応が違う「多相性(polymorphism)」と言う
    → メソッドディスパッチによって達成
  - 例: saving.withdraw(10000) => error
    credit.withdraw(10000) => OK

注: 広義の多相性: 異なる種類のデータを同じプログラムに適用できること

  - 関数型言語の型多相性の例: 整数のリストでも文字列のリストでも「List.length」に適用できる

OOP言語の性質: 継承

- 似たモノどうしを1まとめにする
  - 同じ性質のモノの定義は1つで済ませる
  - 少し性質の異なるモノは「差分」を定義

  → クラス + 継承によって達成
- 「階層分類」・知識フレームで行われてきたこと
  例:「ペンギンは鳥だけど飛べない」

しかし、クラスでなくても「モノ中心」に
考えることはできる → まとめ方に色々な方式

クラスの無いOOP言語: SELF [US87]

credit

id = 12345
balance = 10000
withdraw: =

| :amount |
balance:
(balance - amount)

インスタンスに基づくOOP
(vs. クラスに基づく)

■オブジェクト
=インスタンス変数の集合

■メソッドも値 (「ブロック」)

■インスタンス変数にブロックを格納できる

credit withdraw: 3000.

■メソッド呼出
= インスタンス変数の参照 +
ブロックの実行 (selfが自身を指す)

SELF: オブジェクトの作り方

- インスタンス変数の集合を作る構文
  aPoint <- (| x = 100. y = 100. |)

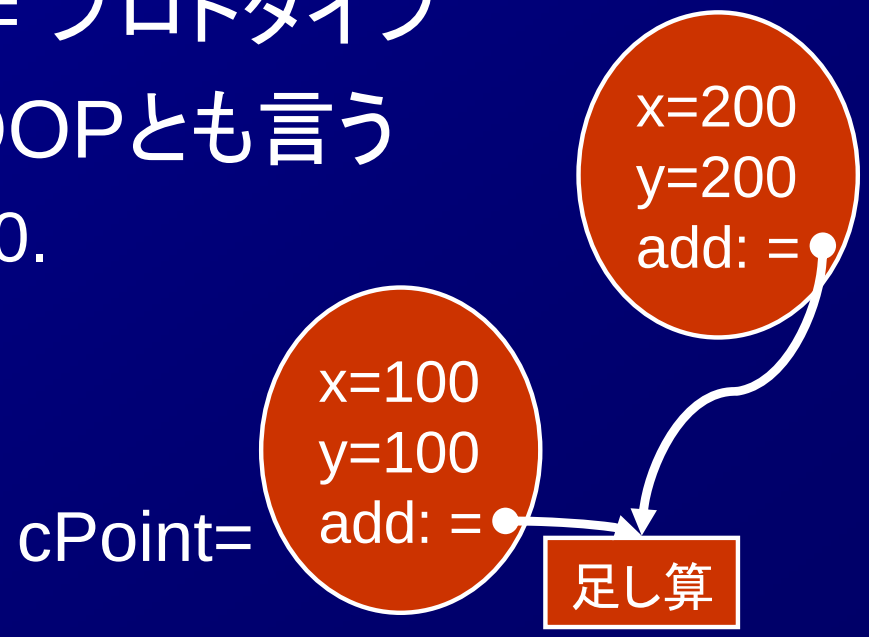
aPoint= x=100 y=100

- 既存のオブジェクトをコピーするメソッド
  bPoint <- aPoint copy.

bPoint= x=100 y=100

SELF: 似たもの同志をまとめる

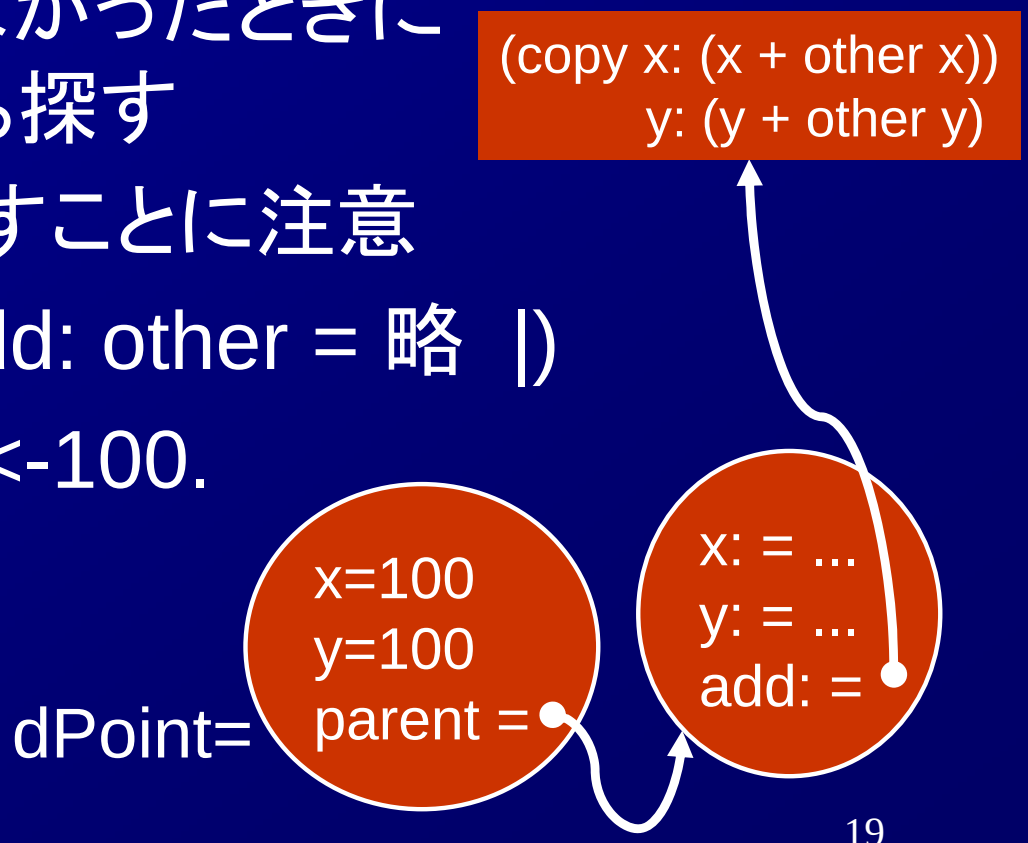
- 似たものの元となるオブジェクトを作っておき、そこから毎回コピーすればよい
  - 元となるオブジェクト = プロトタイプ
  - プロトタイプに基づくOOPとも言う
- cPoint <- (| x = 100. y = 200.
  add: other = (
    (copy x: (x + other x))
          y: (y + other y)).
  |)



- cPoint add: cPoint.

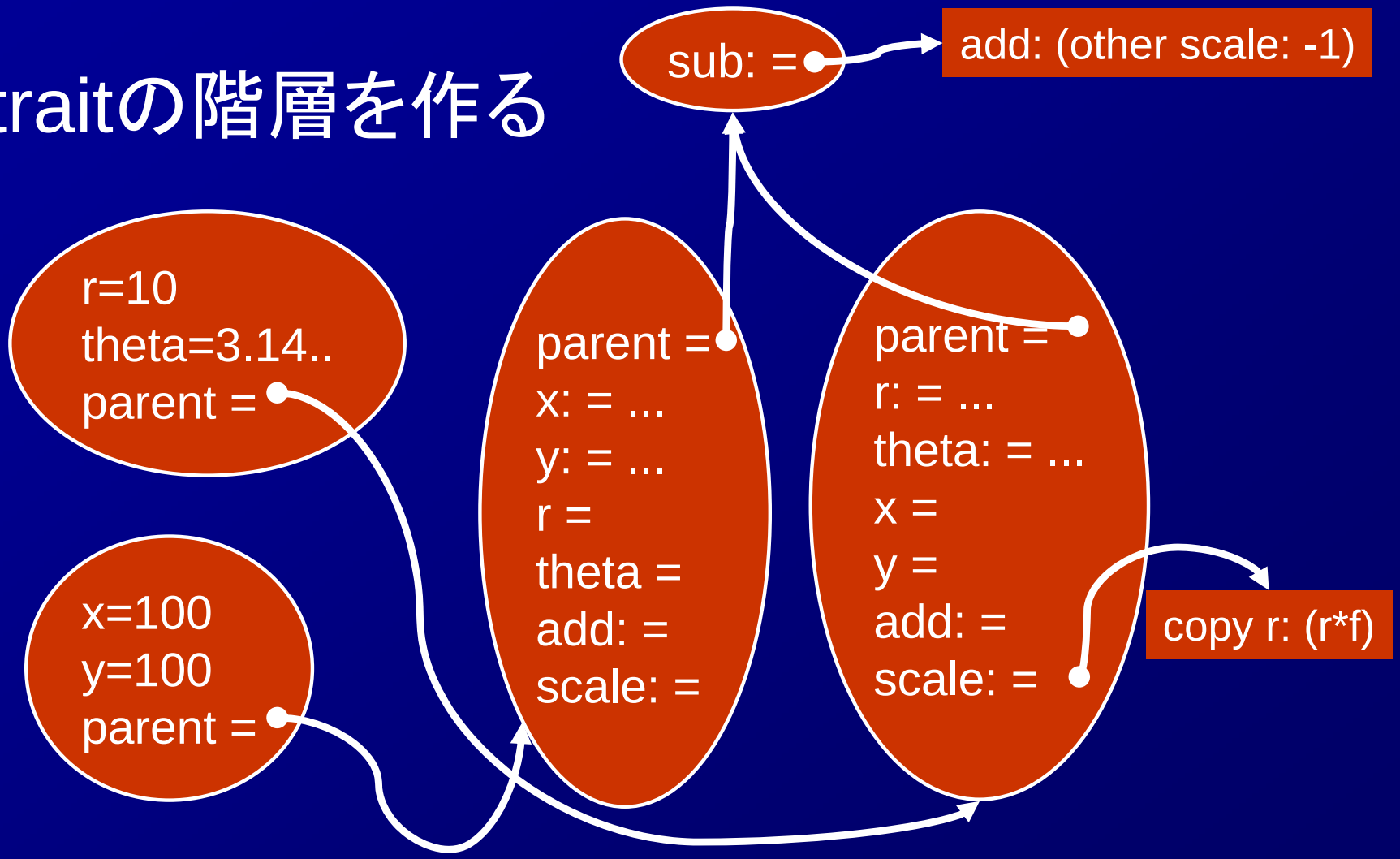
SELF: 似たものをまとめる

- 委譲 (delegation)
  - メソッドが見つからなかったときに別のオブジェクトから探す
  - selfはレシーバを指すことに注意
- pointTrait <- (| x. y. add: other = 略 |)
- dPoint <- (| x<-100. y<-100. parent*=pointTrait |)
- dPoint add: dPoint.



SELF: 階層分類

- traitの階層を作る



何故インスタンスに基づく?

- 「似て非なるモノ」を作る単位を選べる
  - 1個体だけの例外のために子クラスを作らなくてよい
- 言語要素が少ない
  - オブジェクト・インスタンス変数・委譲で全てが達成
- 充分高速に実現できるようになった
  - Polymorphic inline cache (PIC)や動的コンパイルの発明
- SELF以外の例: Simula, Javascript

参考文献

- [所93] 所 真理雄, "オブジェクト指向計算", in オブジェクト指向コンピューティング, pp.1-56, 岩波書店, 1993.
- [US87] David Ungar and Randall Smith, “Self: The power of simplicity”, in *Proceedings of OOPSLA*, 1987.