オブジェクトの広場

［Happy Squeaking!!］

－ オブジェクト指向再入門 －

[第一回] [第二回] [第三回] [第四回] [第五回]

第三回：オブジェクト指向の基礎固め　編（Part 2）

ＩＮＤＥＸ

記事に対するご意見・ご感想をお待ちしています。
umezawa@tyo.otc.ogis-ri.co.jp
気軽にお寄せください。

HOME TOP

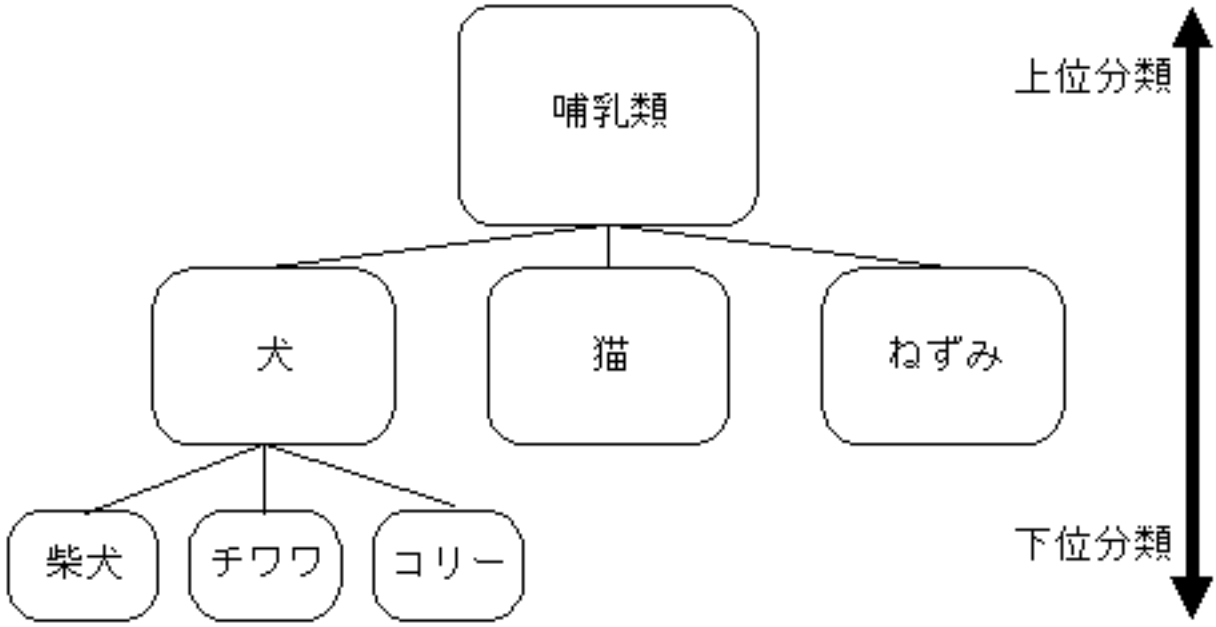
オブジェクトの広場

［Happy Squeaking!!］

1．継承

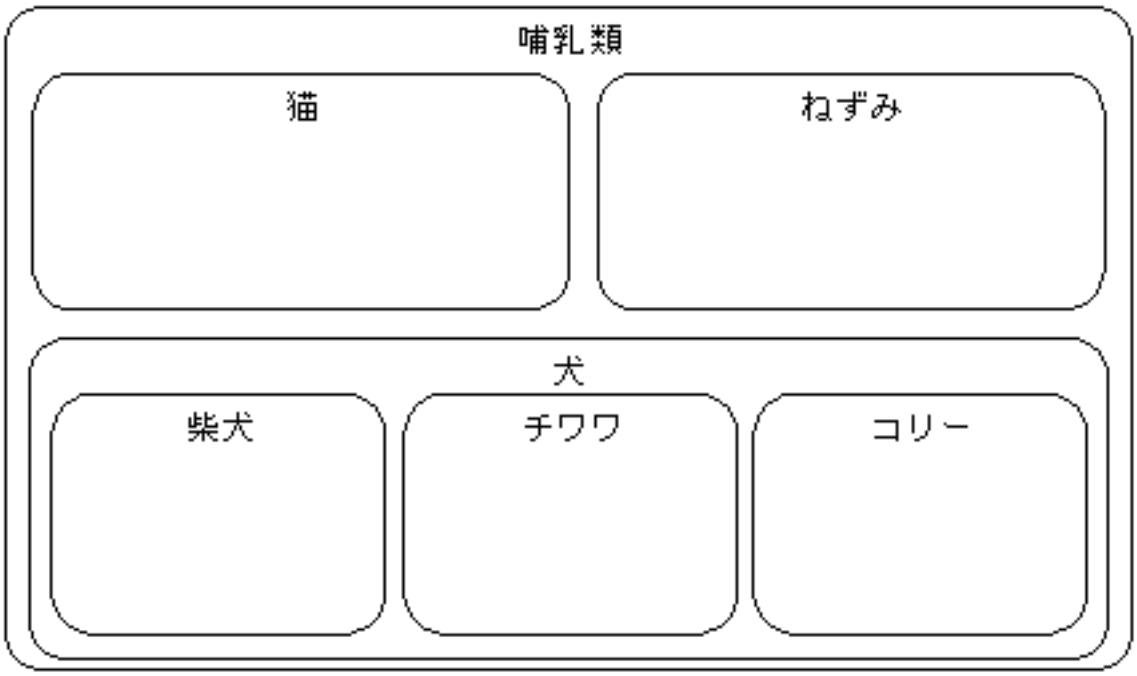
1．1　分類の階層化

個々のインスタンスの共通性に着目すると、それを分類する「もの」としてクラスを考えることができました。例えば「ポチ」や「ラッシー」は「犬」クラスに属し、「タマ」や「ミケ」は「猫」クラスに属するというわけです。
しかしこうした分類は、それを行う人の観点によってその粒度が様々に異なってきます。「ポチ」や「タマ」を見たときに、「哺乳類」クラスに属するとおおざっぱに捉える人もいるでしょうし、ペットブリーダーであれば、「ポチ」を見て「柴犬」クラスに属するとより細かく考えるかもしれません。
このように、場合によって様々な分類が存在するということになると、やはりそれを管理するための枠組みが必要になります。「おおざっぱな分類」と「細かな分類」を、通常私たちは「上位概念に基づく分類」、「下位概念に基づく分類」という形で階層化させることで整理します。



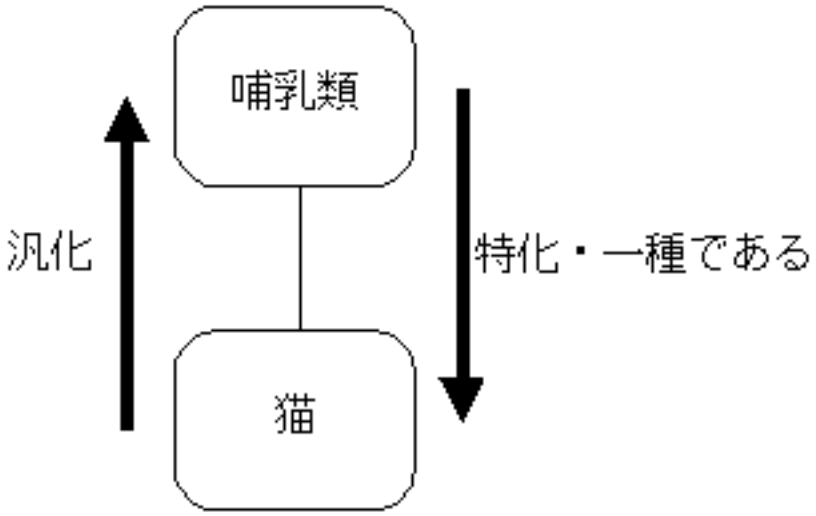
分類の階層化

ベン図ふうに書くと以下のようになります。この場合は外側にいくほど上位の概念上の分類になります。



分類の階層化（ベン図ふう）

下位にいくほど分類の条件が細かくなり、そこに分類される個々のインスタンスの集合は特定されたものになります。
こうした分類概念のの上下の関係は、オブジェクト指向の世界では、「特化(specialization)」や、「汎化(generalization)」といった言葉で表されます。
例として、「猫」は「哺乳類」を「特化」したものとして考えることができます。
一般に上位と下位の分類では、下位は上位の「…(特殊化された)一種である」(a kind of)という関係がなりたちます。
これは、下位の分類からの見方ですが、反対に「哺乳類」は「猫」をさらに一般的にとらえたものすることもできます。この場合には、「哺乳類」は「猫」を「汎化」したものであると呼ばれます。

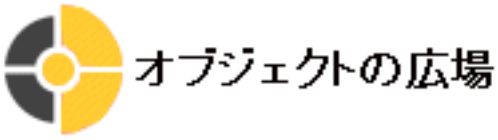


汎化と特化

分類の上下間の関係は、「分類の条件となる情報」の多さによって決まります。例えば「ポチ」が「柴犬」として分類されるためには、「哺乳類」(胎生、恒温動

物、etc)としての条件を満たし、「犬」（ワンとなく、etc）としての条件を満たし、かつ「柴犬」（小型の日本産の犬、etc）としての条件を満たしている必要があります。
つまり下位の分類は、上位の分類に対しさらに特化した分類を指定するため、より多くの条件の情報を持っていることになります。

Index Next



［Happy Squeaking!!］

1. 継承

1. 2 クラスの継承

「分類するためのもの」としてのクラスは、分類のための指定情報を個々に持っていたのでは効率の悪いことになります。

「柴犬」というクラスを考えた場合に、インスタンスの定義情報として、

- 胎生
- 恒温
- 日本産
- 小型
- ワンとなく

といったものをすべて待たせることもできますが、「犬」や「哺乳類」といったより上位の分類を意味するクラスでも同じような定義が存在するため、情報としては冗長になってしまいます。

一方で、

「哺乳類」

- 胎生
- 恒温

「犬」

- ワンとなく
- (上位の「哺乳類」の定義が暗に含まれる)

「柴犬」

- 日本産
- 小型
- (上位の「犬」の定義が暗に含まれる)

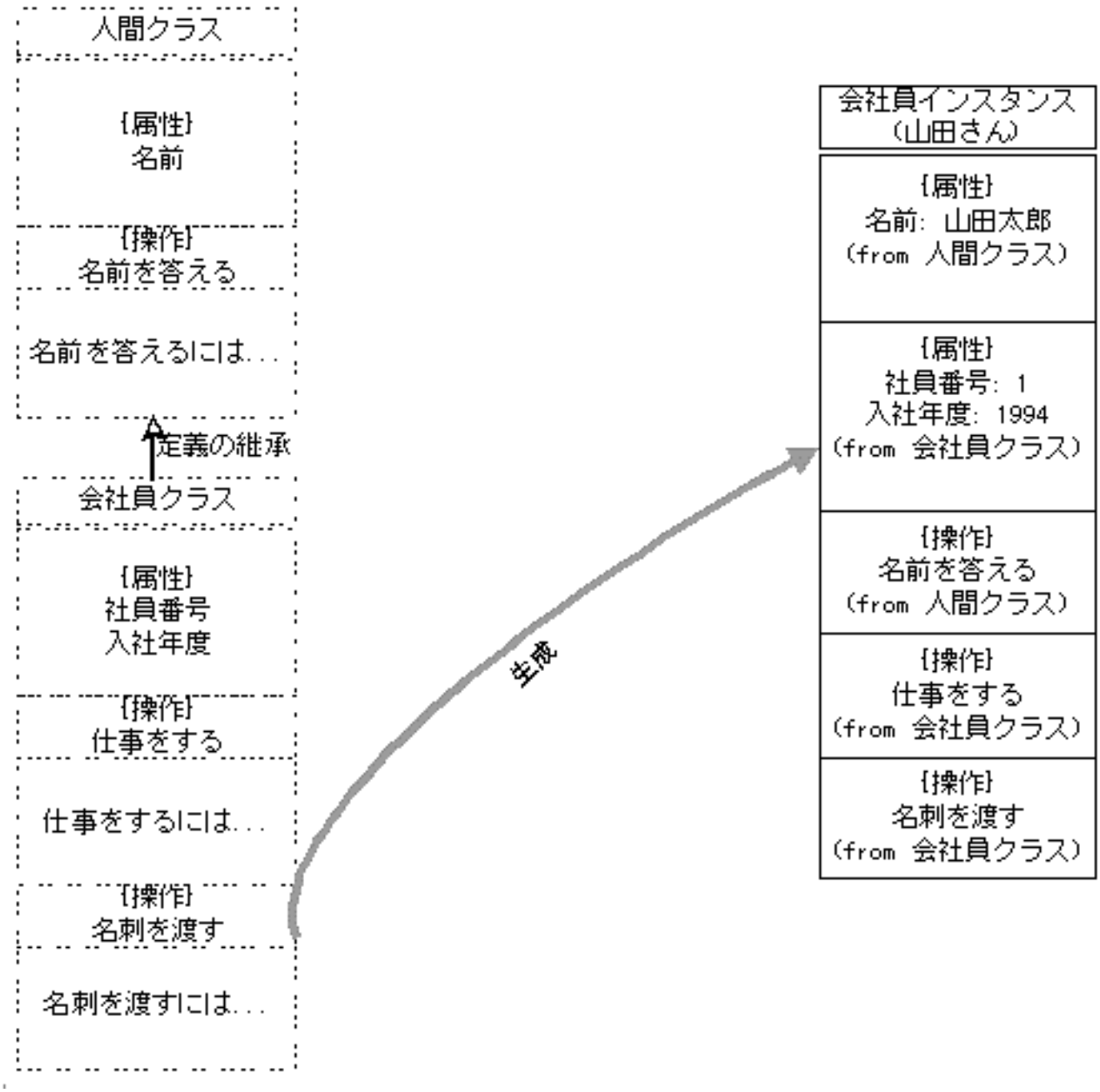
として、上位クラスで規定されているインスタンス定義情報が下位クラスでそのまま暗に含まれることにすれば、情報の冗長化を防ぐことができます。

このように、上位のクラスが持つインタンスについての定義情報を、その下位のクラスがそのまま引き継いで自分のインスタンス定義情報として使えることを、オブジェクト指向の世界では「(上位のクラスからの定義情報を)継承する」という言葉によって表します。

クラスの上下関係を定めることにより継承されるものは、属性、操作の定義にとどまりません。クラスをサポートするオブジェクト指向言語では、通常操作の実装はクラスが保持しており、インスタンスは、センダからのメッセージが来た場合に、自分の属するクラスに存在する操作の実装(メソッド)を起動します。この操作の実装も下位のクラスは継承して手に入れることになります。

複数のインスタンスの持つ共通性は、「クラス」という形でまとめあげられましたが、各クラスが持つインスタンス定義情報の共通性は、「クラス間の継承」という階層化の仕組みによりまとめられていくのです。

継承を使ったクラス定義から、インスタンスが生成されるイメージは以下のようになります。(今度は会社員の例を使っています)



インスタンス生成のイメージ

インスタンスは、上位クラス群での定義情報をすべて含んだ形で生成されます。特定のクラスから見て上位のクラスは通常「スーパークラス」、下位のクラスは「サブクラス」というふうに呼ばれます。(これは相対的な概念です)。

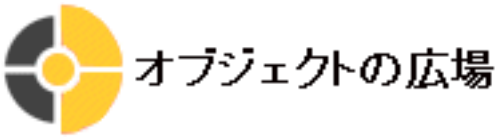
サブクラスで生成されたインスタンスは、あたかも自分のクラスで属性、操作の全ての定義情報があったかのような構成になるのですが、実際の定義は部分的にスーパークラスに存在することになります。
サブクラスのインスタンスにメッセージが来た際には、インスタンスは、対応する操作の実装(メソッド)を、自分のクラスだけでなく、スーパークラスからも参照して起動します。この場合には、操作の実装が継承されて起動されることになります。

操作においては、サブクラスがスーパークラスと同じ名前の操作をわざと再定義することもできます。サブクラスのインスタンスは自分のクラスの方にある操作を優先して起動するので、スーパークラスは操作の振る舞いの仕方を書き換えることになります。これは通常オーバーライド(上書き)と呼ばれています。

継承によって「属性」「操作」の多くの情報のコピーが様々なクラスに散在することが避けられるようになります。開発者にとっては、属性や、操作は、上位クラスからの変更、および追加したい部分だけを下位クラスで再び書き下せばよくなるので、これを「差分プログラミング」と呼ぶ場合もあります。

クラス群を上手に階層化することによって、概念上の区分がきちんと為され、かつ冗長なコードの存在しないシステムが作られていきます。
一方で、操作の実装の再利用だけにとらわれ、概念上の上下関係を無視すると、継承の誤った使用により保守性を低めることがあるので注意が必要です。(これについてはより説明の進んだ時点でふりかえります)。

Prev. Index Next



［Happy Squeaking!!］

2. Squeak演習：クラスの継承関係作成

2. 1 定義されるモデル

先ほどの図で示した会社員の例をそのままSqueak上に表現してみることにします。

作成されるモデルは、
クラス: Human
属性: name
操作: getName、setName
クラス: Employee (Humanを継承)
属性: id、employedDate
操作: giveNameCard

といった内訳になります。

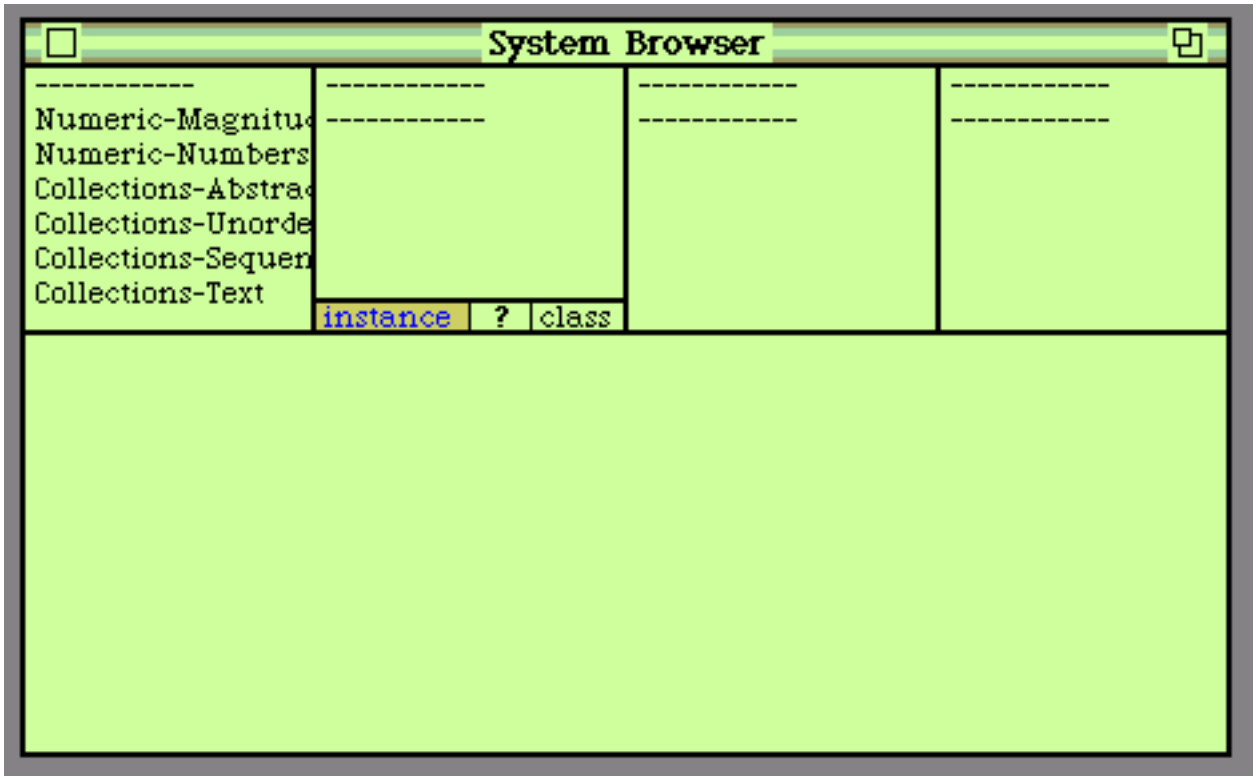
オブジェクトの広場

［Happy Squeaking!!］

2. Squeak演習：クラスの継承関係作成

2. 2 Humanクラスの作成

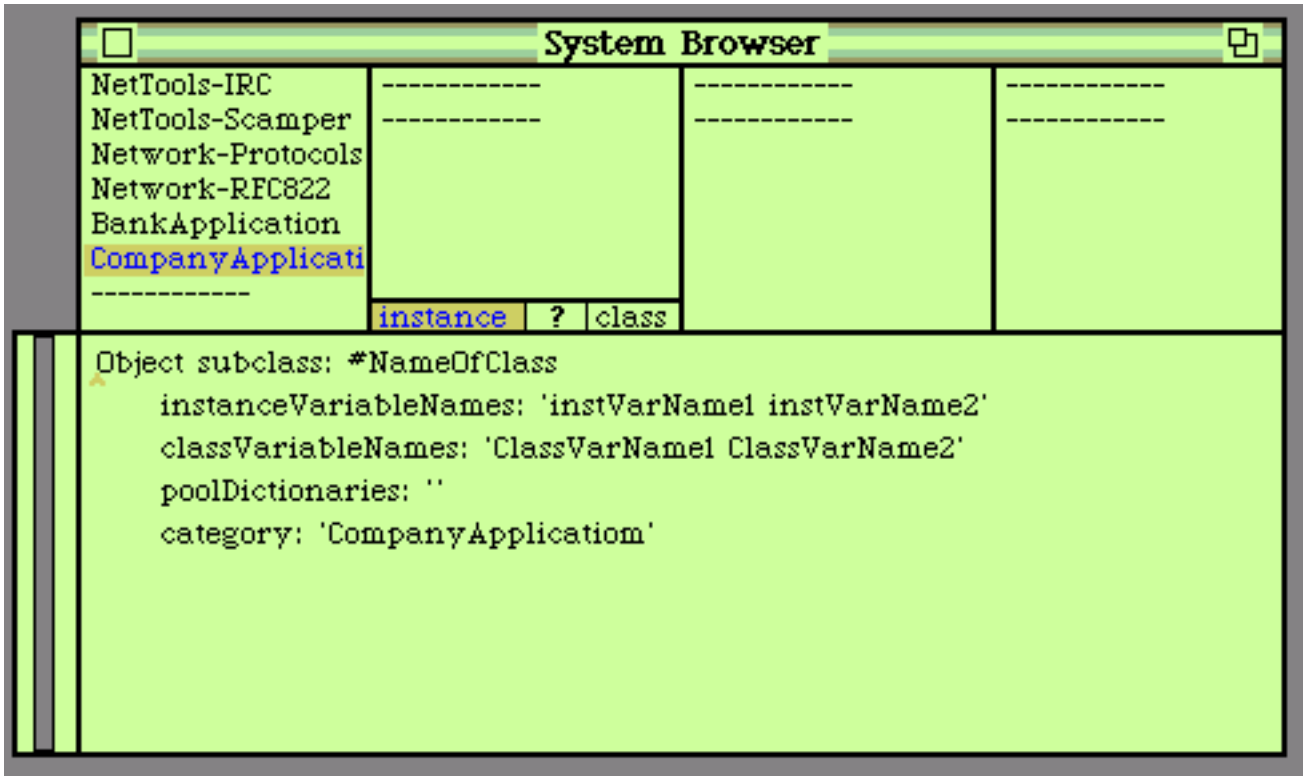
今回は、Human、Employeeの二つのクラスを作成しますので、クラスブラウザではなく、システムブラウザを起動します。
(メインのポップアップメニューから、"open.." -> "browser"で立ち上がります。)



システムブラウザの起動

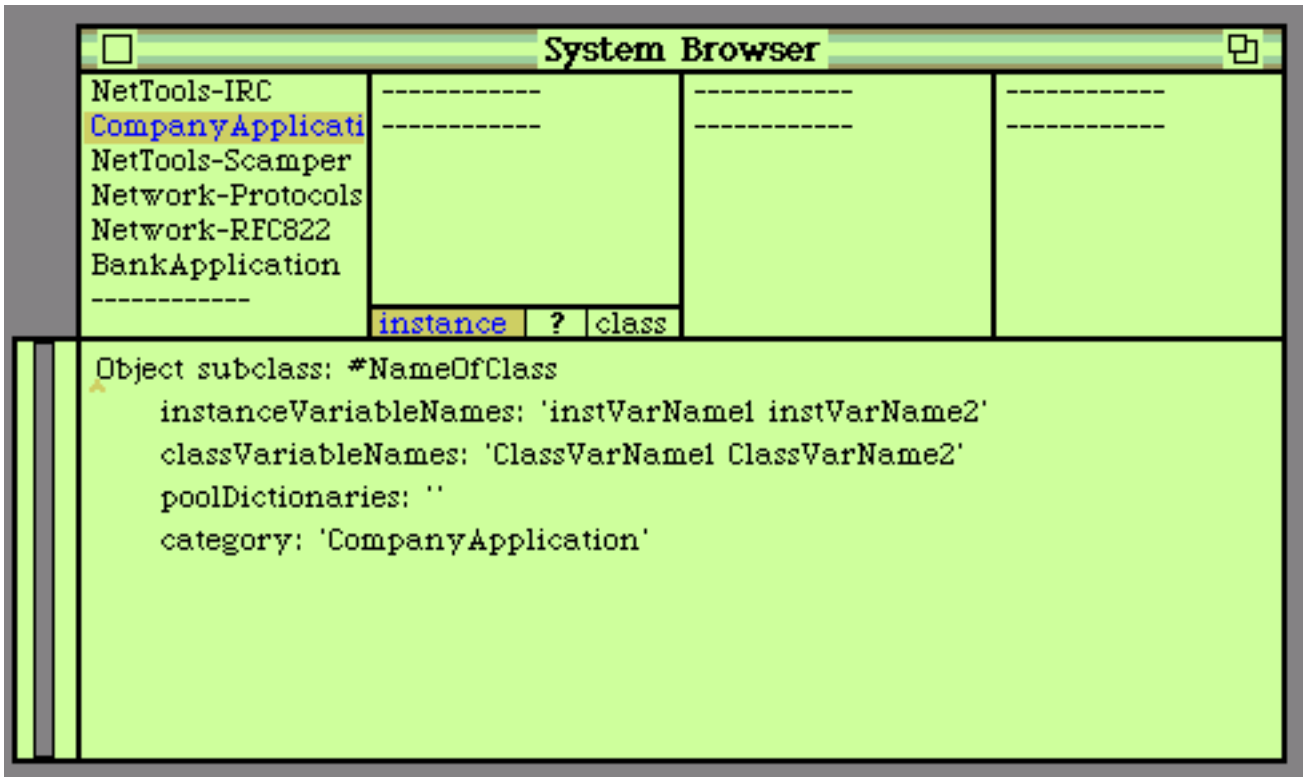
基本的な操作なクラスブラウザと同じです。
左上には、クラスカテゴリがリストされています。今回はリストの最後尾に、CompanyApplicationというクラスカテゴリを追加することにします。

左上の部分でカテゴリリストを何も選択していない状態のままポップアップメニューを出し、"add item…"を選択します。カテゴリ名を聞かれるので、CompanyApplicationと入力すると、新たなクラスカテゴリが追加できます。



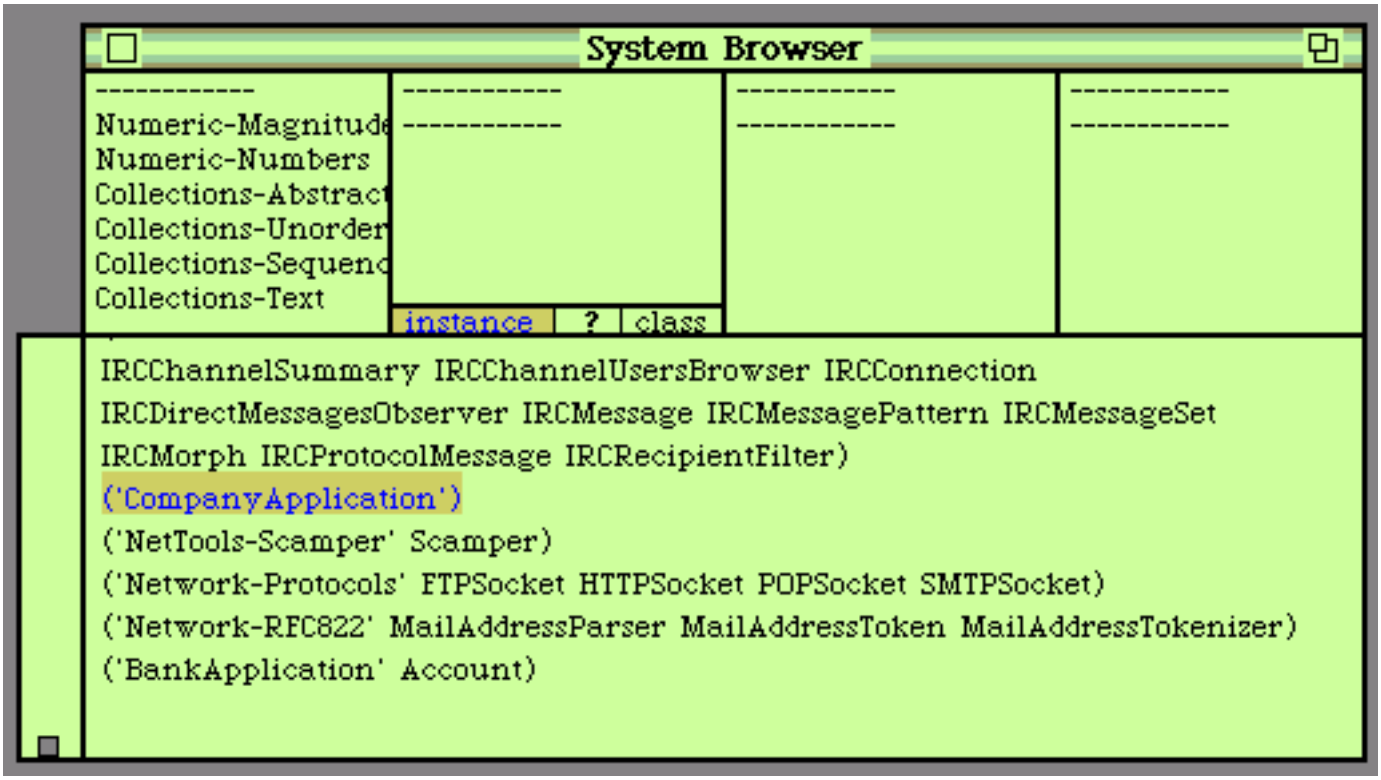
追加されたCompanyApplicationカテゴリ

カテゴリリストを選択した状態で上記の操作を行ってしまった場合には、リストの最後ではなくリストの選択した位置にカテゴリが追加されてしまいます。



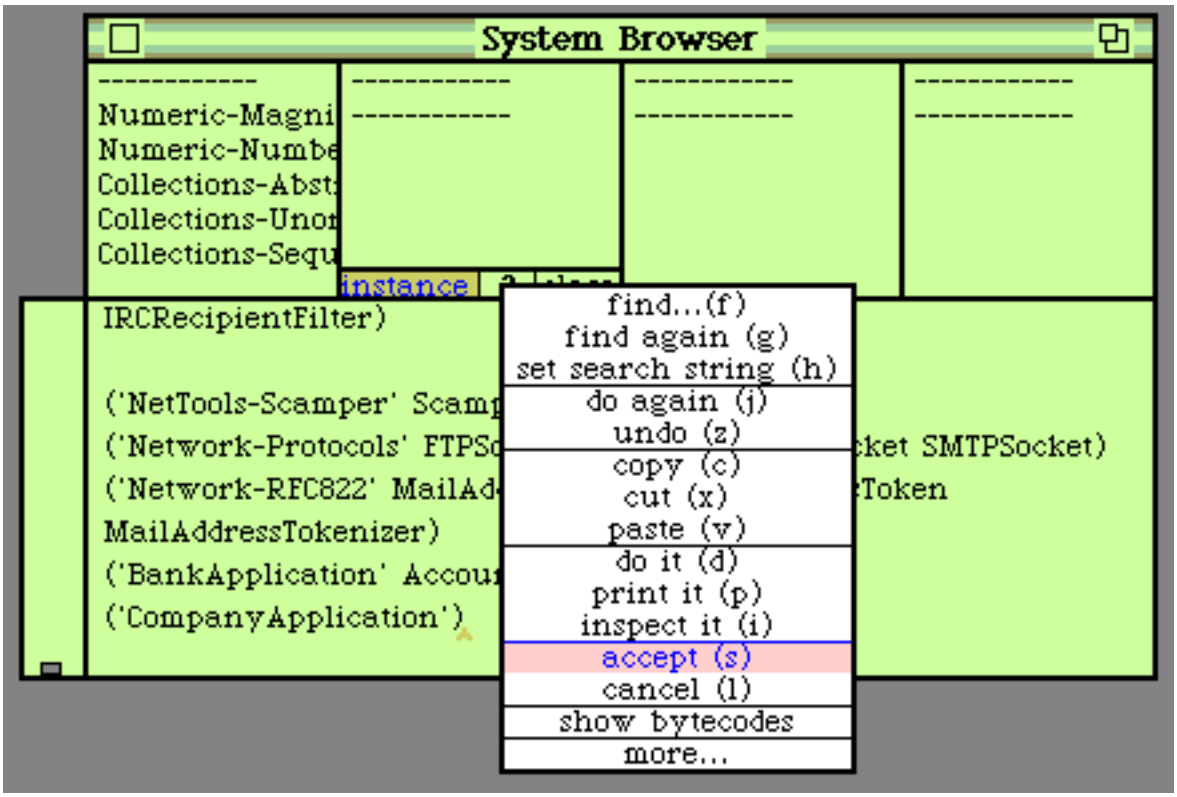
途中に追加されてしまったComapanyApplicationカテゴリ

これでは見栄えが悪いので、その場合にはカテゴリリストのポップアップメニューから "reorganize"を選択して位置を調節してください。



"reorganize"によるクラスカテゴリ位置の調節

間に入ってしまっている"CompanyApplication"の部分を選択し、最後尾にカット・ペーストして、"accept"します。



reorganize"によるクラスカテゴリ位置の調節(2)

クラスカテゴリの並びかえは、今後、Squeakを使い込んで自分で定義したカテゴリが増えてきた場合、必ず必要になりますので覚えておきましょう。

次にCompanyApplicationクラスカテゴリ下にHumanクラスを作成することにしま

す。

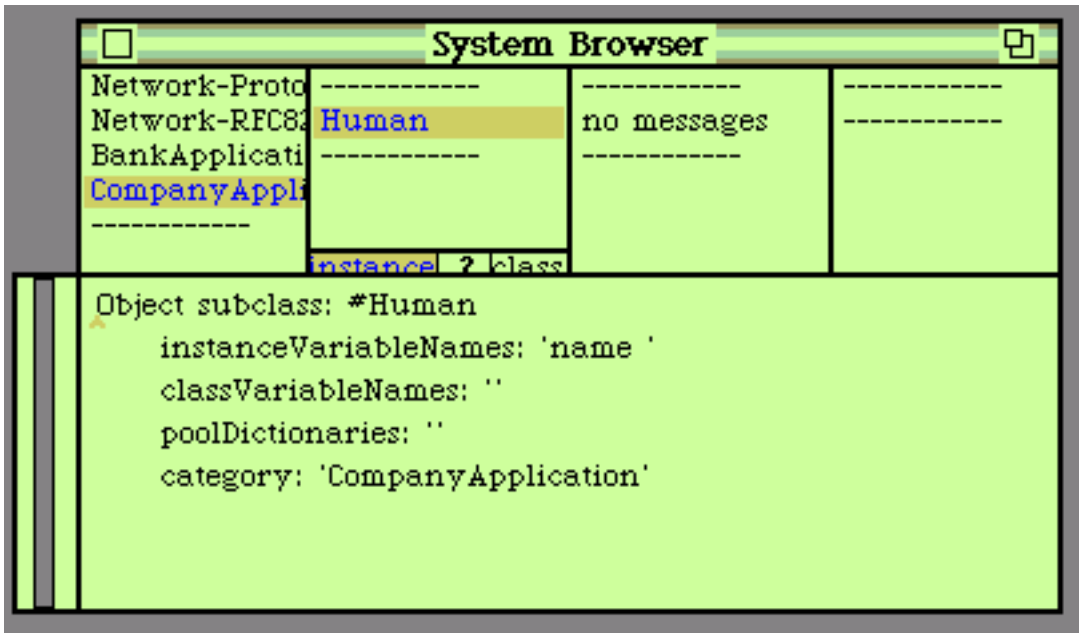
クラスカテゴリを新規作成すると、ブラウザの下部には、クラス作成用のテンプレートか表示されています。

```
Object subclass: #NameOfClass
        instanceVariableNames: 'instVarName1 instVarName2'
        classVariableNames: 'ClassVarName1 ClassVarName2'
        poolDictionaries: ''
        category: 'CompanyApplication'
```

このテンプレートを以下のように書き換えます。

```
Object subclass: #Human
        instanceVariableNames: 'name '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CompanyApplication'
```

ブラウザ下部のポップアップメニューの"acept"で、Humanクラスが実際にシステムに登録されます。



作成されたHumanクラス

あとは前回と同様に、メッセージカテゴリ "accessing"を追加して、getName、setName: aNewName の二つのメソッドを追加してください。

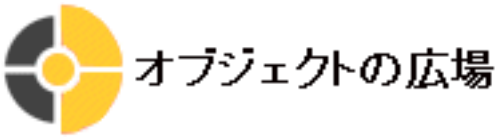
それぞれの実装は以下のようになります。

Humanクラスの getNameメソッド

```
getName
        ^name
```

Humanクラスの setName: メソッド

```
setName: aNewName
        name := aNewName
```



［Happy Squeaking!!］

2. Squeak演習：クラスの継承関係作成

2. 3 Employeeクラスの作成

次にHumanのサブクラスとしてEmployeeクラスを作成します。

同様にシステムブラウザを使ってクラス作成用のテンプレートを書き換え、"accept"します。

```
Human subclass: #Employee
        instanceVariableNames: 'id employedDate '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'CompanyApplication'
```

特に一行目の先頭に注意してください。今まではObjectでしたが、今回は、Humanになります。これはHumanのサブクラスとして、Employeeを指定するということを意味します。

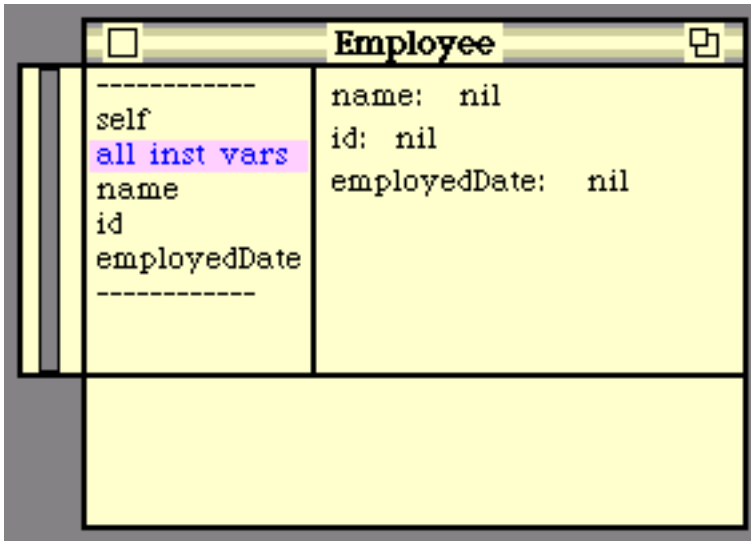
> Objectは継承のツリーのルートとなっている一番上位にあるクラスです。通常Smalltalkでは、どのクラスも継承しないということはなく、最低でもルートであるObjectから継承することになります。

> SmallTip: Smalltalkでは、継承ツリーのルートはObjectである

まだ操作を追加していないですが、とりあえずこの状態で、ワークスペース上でEmployeeのインスタンスを生成して観察してみましょう。

```
"従業員 山田さんの作成、観察"
| yamada |
yamada := Employee new.
yamada inspect
```

以下のようなインスペクタが立ち上がります。



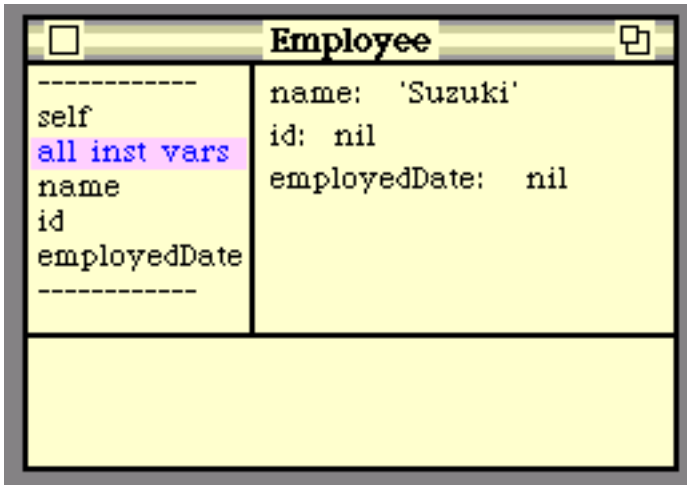
Employeeインスタンス(山田さん)のインスペクト

インスペクタの左側に注目してください。Employeeクラスの属性としては記述されていないnameフィールドがあることが確認できます。これはEmployeeクラスのスーパークラスであるHumanの属性であるnameの定義を引き継いだことを意味しています。

属性だけでなく操作も同様に継承されています。今度は従業員 鈴木さんのインスタンスを作成し、setName: メッセージを送ってみましょう。

"鈴木さんインスタンスの作成、名前付け、観察"
```
| suzuki |
suzuki := Employee new.
suzuki setName: 'Suzuki'.
suzuki inspect
```

以下のようなインスペクタが立ち上がります。



Employeeインスタンス(鈴木さん)のインスペクト

Employeeクラス自体では、setName: のメッセージに該当する操作が定義されていないのですが、そのスーパークラスでsetName: 操作の定義がされているため、問題なくname変数に値が設定されることになります。

それでは次にEmployeeクラスに対し、giveNameCardの操作を追加することにしましょう。

Employeeクラスに"actions"というメッセージカテゴリを追加します。

giveNameCardの実装は以下のようになります。

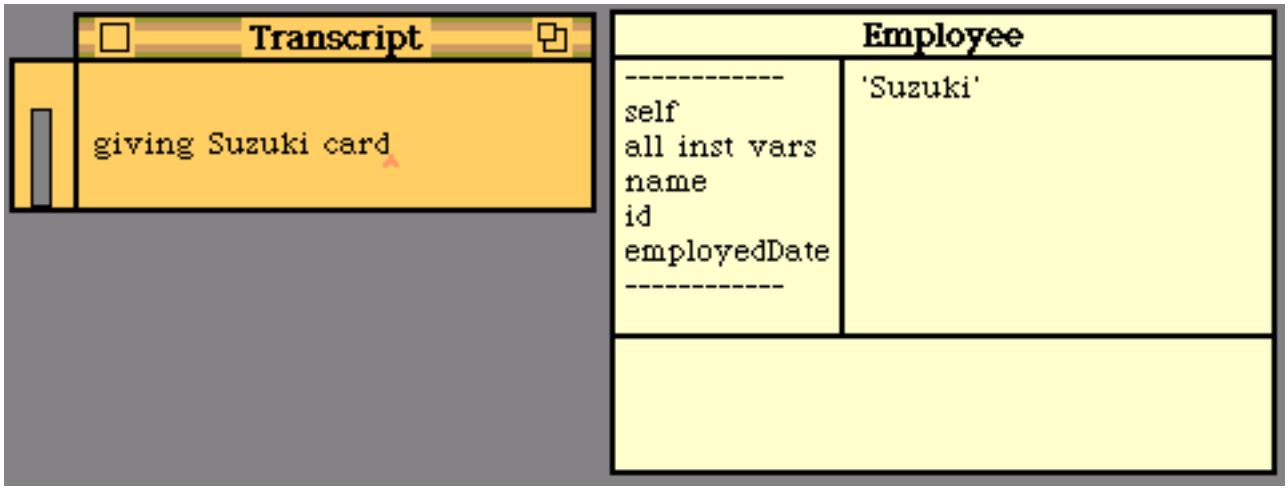
```
giveNameCard
        Transcript cr;
                show:  'giving ' ,  name, ' card'.
```

Transcriptに送っているcrは改行させるためのメッセージです。

giveNameCardのメソッド本体で、name変数を参照しています。これは、スーパークラスで属性として定義されているために問題なく使えることになります。

```
| suzuki |
suzuki := Employee new.
suzuki setName: 'Suzuki'.
suzuki giveNameCard.
suzuki inspect
```

Transcriptを開いてから実行してみてください。以下のように表示されます。



Employeeインタンスに giveNameCardメッセージを送る

インペクタのnameの値は、やはり'Suzuki'になっています。

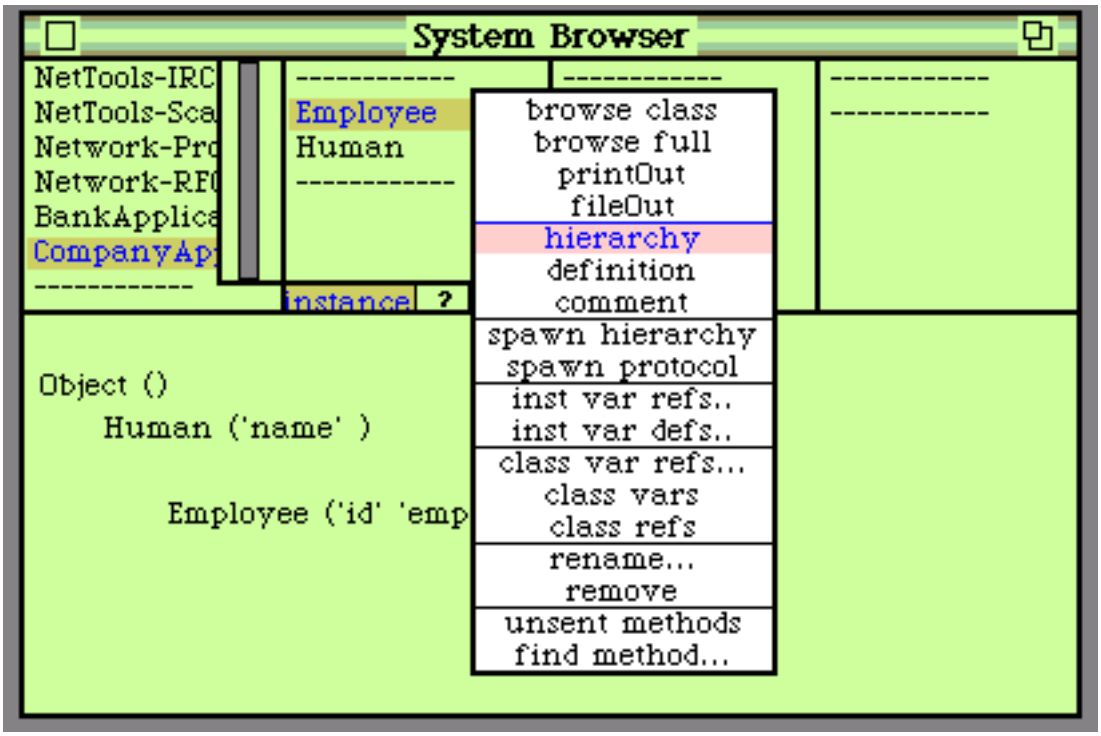
オブジェクトの広場

［Happy Squeaking!!］

2. Squeak演習：クラスの継承関係作成

2. 4 階層ブラウザの利用

Humanクラスも、Employeeクラスも一応すべての実装をおこなったので、ここでもう一度全体像を把握してみましょう。
システムブラウザで、Employeeクラスを選択し、ポップアップメニューで"hierarchy"を選択します。



"hierarchy"の選択

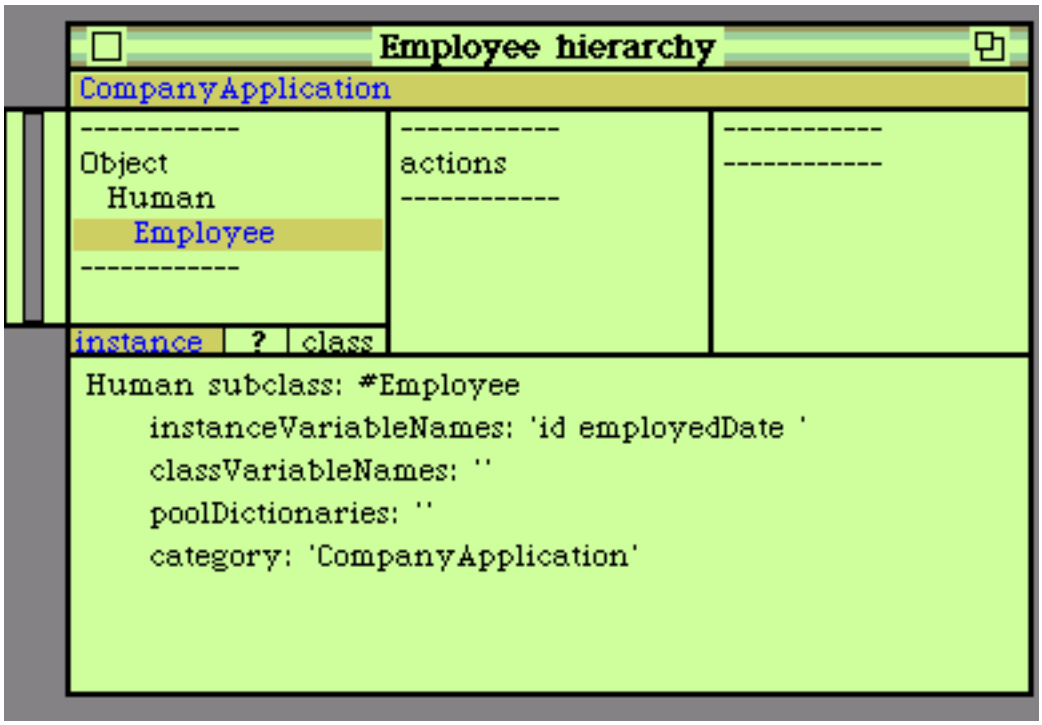
下部に、継承関係を示すテキストがインデントされて表示されます。

```
Object ()
        Human ('name' )

                Employee ('id' 'employedDate' )
```

Objectが継承木のルートであり、それをサブクラスとしてHumanがあり、さらにそのサブクラスが、Employeeになっています。

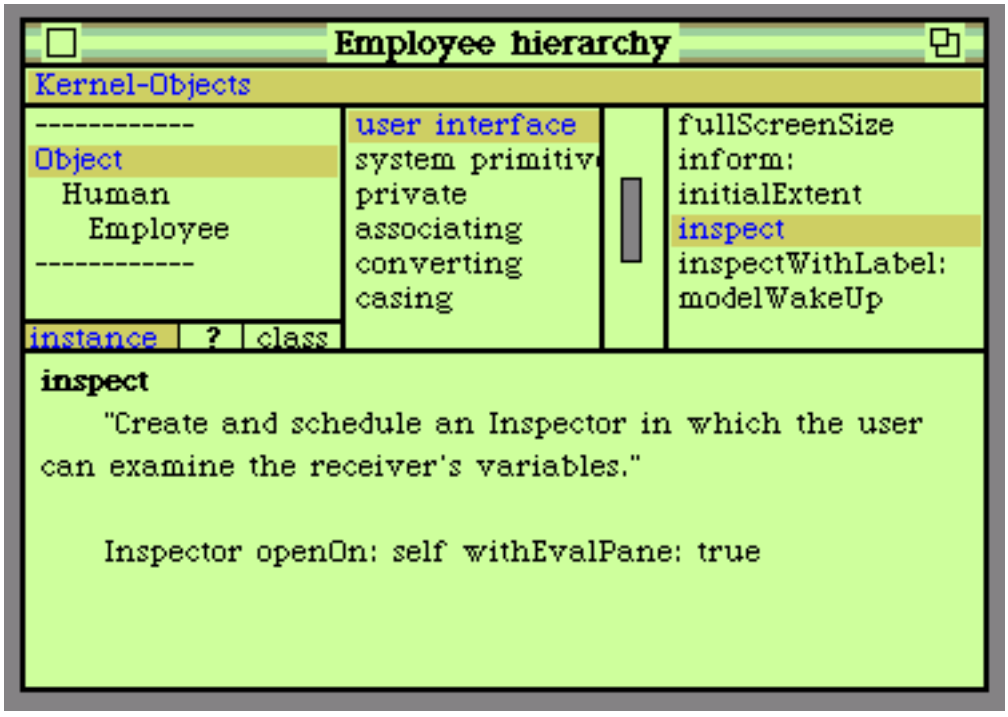
さらに詳しい情報を見たい場合には、システムブラウザから、Hierarchy Browserを開くことができます。
今度はポップアップメニューで、"spawn hierarchy"を選択してください。
階層ブラウザが立ち上がります。



階層ブラウザの起動

階層ブラウザは、クラスの継承関係をたどりながら、属性や操作の定義を見ていくときに非常に便利です。

一番上のObjectを選択してみましょう。実に多くの操作が定義されています。
いままで、任意のインスタンスに対し、inspectのメッセージを送ることでその中身を見ることができましたが、これも実はObjectクラスでinspectのメソッドが定義されていたことによります。



Objectクラスで定義されているinspectのブラウズ

オブジェクトの広場

［Happy Squeaking!!］

2. Squeak演習：クラスの継承関係作成

2. 5 getNameメソッドのオーバーライド

それでは次に、Employeeクラスに、Humanクラスで定義されているのと同じ操作である、getNameを再定義することにします。

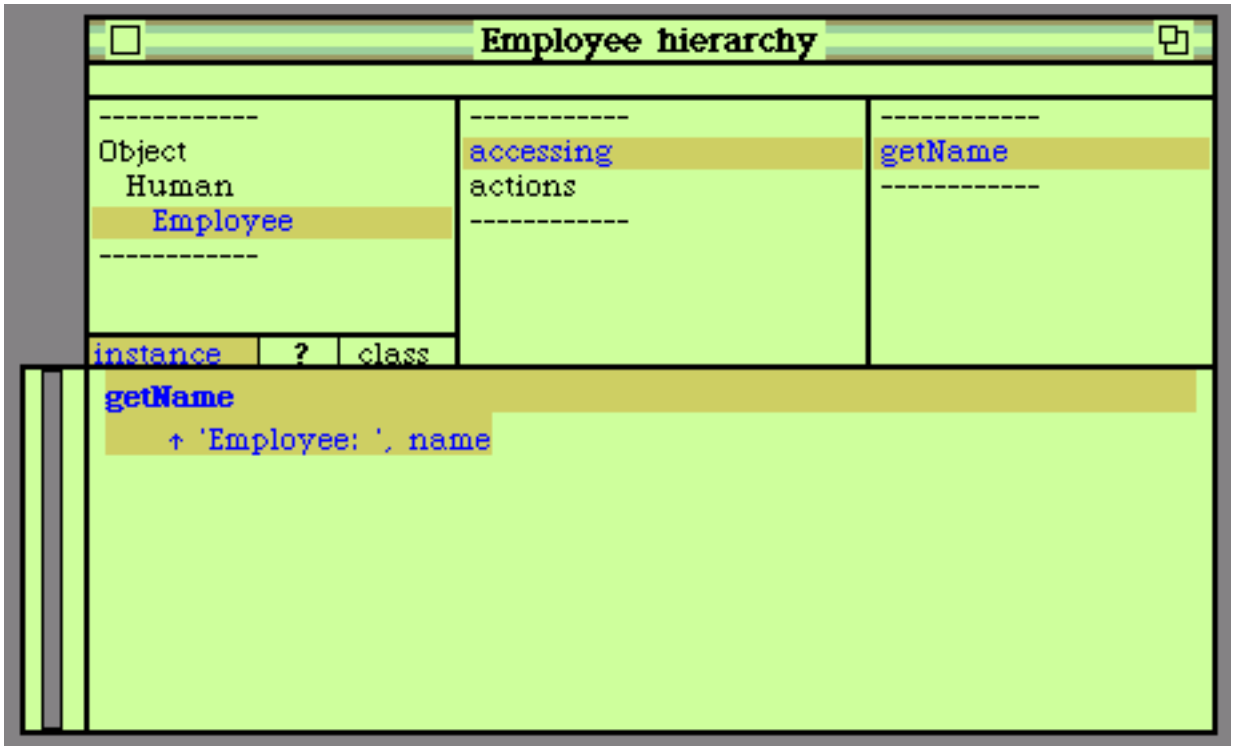
階層ブラウザで、Employeeを選択し、メッセージカテゴリ 'accessing'を追加します。

getNameの実装は、以下のように、スーパークラスEmployeeのgetNameの実装とは少し変えてみます。

先頭に 'Employee: 'と文字列が追加された形で文字列が返るようにします。

```
getName
        ^ 'Employee: ', name
```

'accept'してください。
これでオーバーライドができたことになります。



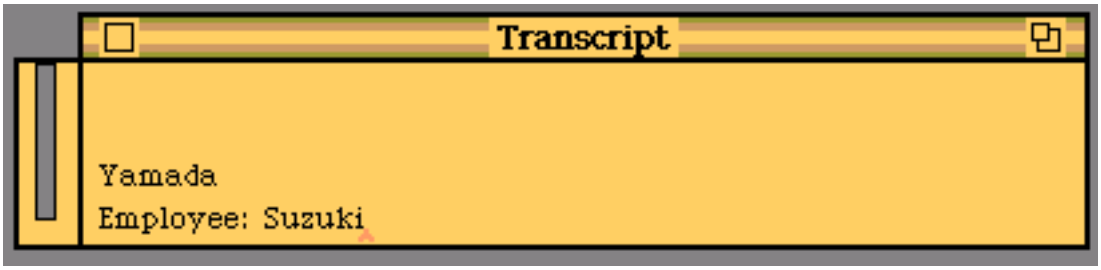
HumanクラスのgetNameをEmployeeクラスでオーバーライド

実際に振る舞いが、上書きされているかどうかを確かめるため、簡単なコードをワークスペース上で実行してみます。

```
| yamada suzuki |
yamada := Human new.
yamada setName: 'yamada'.
suzuki := Employee new.
suzuki setName: 'Suzuki'.
Transcript cr; show: yamada getName.
Transcript cr; show: suzuki getName.
```

山田さんはHumanのインスタンスですが、鈴木さんはEmployeeのインスタンスです。EmployeeクラスはHumanクラスを継承しているために、setName: の操作は共通の動作が行われます。一方で、getNameに関しては、Employeeクラスでメソッドの再定義(オーバーライド)を行ったので、鈴木さんは山田さんとは違う動作をすることになります。

Transcriptを開いたうえで実行してみてください。以下のような結果になります。



オーバーライドされたgetName操作の実行結果

各インスタンスがそれぞれの所属するクラスでのメソッドに従って動作しているのが確認できます。

Prev. Index Next

オブジェクトの広場

［Happy Squeaking!!］

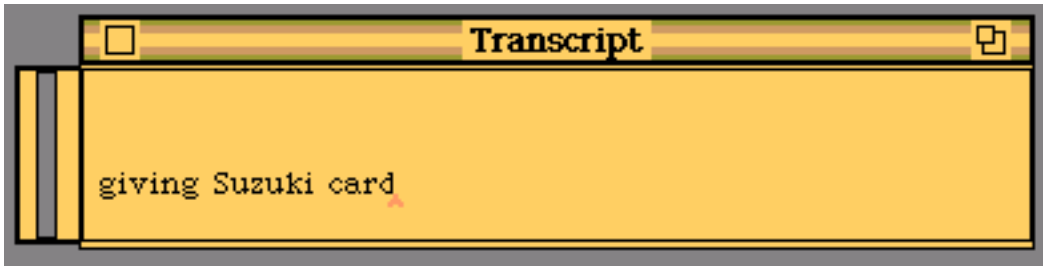
2. Squeak演習：クラスの継承関係作成

2. 6 selfの利用

ワークスペース上で再び以下のコードを実行してみてください。

```
| suzuki |
suzuki := Employee new.
suzuki setName: 'Suzuki'.
suzuki giveNameCard.
```

Transcriptでは、やはり以下のように表示されます。



EmployeeのgiveNameCardの実行結果

これを先ほどのgetNameの実行のように、名前の部分で 'Employee: 'という文字列が追加された形で表示されるようにするにはgiveNameCardメソッドにどのような変更を加えればよいでしょうか。

一つは以下のようにする方法があります。

```
giveNameCard
        Transcript cr;
                show:  'giving ' , ('Employee: ', name), ' card'.
```

メソッドgetNamenの場合と全く同じように'Employee: ', name と書き換えました。これでも期待する結果は得られますが、全く同じコードが二個所に分散しているのは保守上いかにも関心できません。

では、giveNameCardメソッドの中で、先ほど定義しなおしたgetNameメソッドが起動できるとしたらどうでしょうか。

つまり、

```
giveNameCard
        Transcript cr;
                show:  'giving ' , (## getNameの呼び出し ##) , ' card'.
```

のように記述できれば、(## getNameの呼び出し ##) 部分が実行され、その結果

'Employee: 'が追加された文字列が返ってきて、うまくいくはずです。

実はSmalltalkでは、自分のクラス内で定義されたメソッドを起動するときに、以下のように書くことができます。

```
giveNameCard
        Transcript cr;
                show:  'giving ' , ( self getName ) , ' card'.
```

メソッドの起動を行うには、通常、なんらかのオブジェクトに対してメッセージを送らなければなりません。しかし上の場合は、getNameメッセージの受け手は自分自身に他なりません。getName操作を定義しているのは自分のクラスなのですから。

このように外部からではなく内部的にメソッドを起動したい場合には、自分自身を表すオブジェクトが必要になります。これがselfなのです。

SmallTip: 自身を表すための特別なオブジェクトとしてselfがある

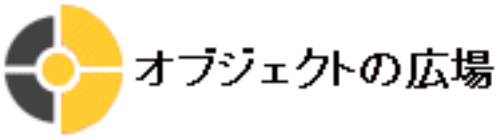
C++や、Javaでは、selfではなくthisというキーワードを使いますが、同じ物を表しています。

self オブジェクトを使ってメソッドを書きなおし再度実行してみましょう。結果は当然ながら、

```
giving Employee: Suzuki card
```

となります。

Prev. Index Next



[Happy Squeaking!!]

2. Squeak演習：クラスの継承関係作成

2. 7 superの利用

「論よりコード」で、まず以下のようにEmployeeの giveNameCardを変更してみましょう。

```
giveNameCard
        Transcript cr;
                show:  'giving ' , ( super getName ) , ' card'.
```

実行結果は、以下のように

```
giving Suzuki card
```

と、元に戻ってしまいます。

自分自身に対する呼び出しを行う際に、selfの代わりに superを使用することができます。この場合、メッセージの対応するメソッドの検索は、自身の属するクラスを無視して、一つ上のスーパークラスから行われることになります。上記の例では、getNameメソッドが自分のクラスでオーバーライドされているにも関わらず、それが無視されて一つ上のスーパークラスであるHumanからgetNameメソッドが検索され、起動されるのです。

superは、スーパークラスとサブクラスでメソッドの実装の差分を記述する際に有効です。
例として初期化のメソッドinitializeをHumanクラスとEmployeeクラスに追加することにします。

Humanクラスのinitializeでは、自分のクラスで定義されている属性、nameの初期化を行います。

```
Human クラスの initiallize (initialize-releaseカテゴリ)

Initialzie
        name := 'NO NAME'.
```

Humanのサブクラス、Employeeでは、新たに id、employedDateの属性が加わっているので、その部分の初期化を行うようにします。

```
Employee クラスの initiallize (initialize-releaseカテゴリ)

Initialize
        Id := 0.
        employedDate := Date today.
```

Date todayは、Dateというクラスにtodayメッセージを送ることで「今日の日付」インスタンスを生成しています。(通常はnewとするところですが、Dateの場合はわかりやすさのためにこういったメソッドが使えます)

SmallTip: Dateは日付を表すクラス。todayメッセージを送ることで、「今日の日付」インスタンスを得ることができる。

このような状態で以下のコードをワークスペースで実行してみます。

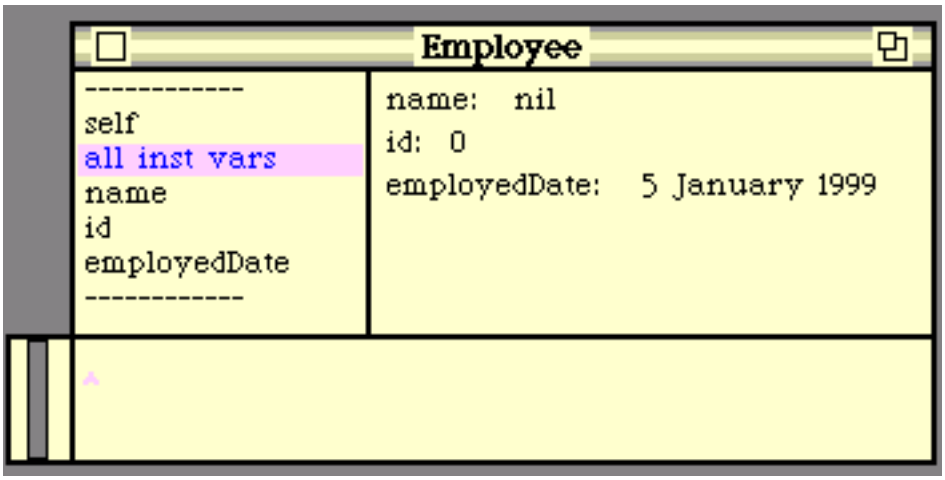
```
| emp |
emp := Employee new.
emp initialize.
emp inspect.
```

(Smalltalkの書き方に慣れてきた方は以下のように

```
Employee new initialize inspect.
```

と短くしていただいてかまいません)。

立ち上がったインスペクタをみてみましょう。



Initializeの記述が不十分なEmployeeインスタンス

Employeeのインスタンスにinitializeメッセージを投げた場合、initializeメソッドがオーバーライドされているため、当然、自分のクラスで定義したinitializeが起動されます。しかし、そのinitializeの中ではスーパークラスから定義を継承しているはずのname属性の初期化を行っていないために、nameの値はnilのままになっています。

ひとつの解決策として、Employeeのinitializeメソッドに、nameの初期化部分を書いてしまう方法があります。

```
Employee クラスの initiallize (initialize-releaseカテゴリ)

Initialzie
        name := 'NO NAME'.  "<= 追加部分"
        Id := 0.
        employedDate := Date today.
```

しかしこれでは、前述のselfでの例のように、同じコードが二箇所に分散すること

になります。

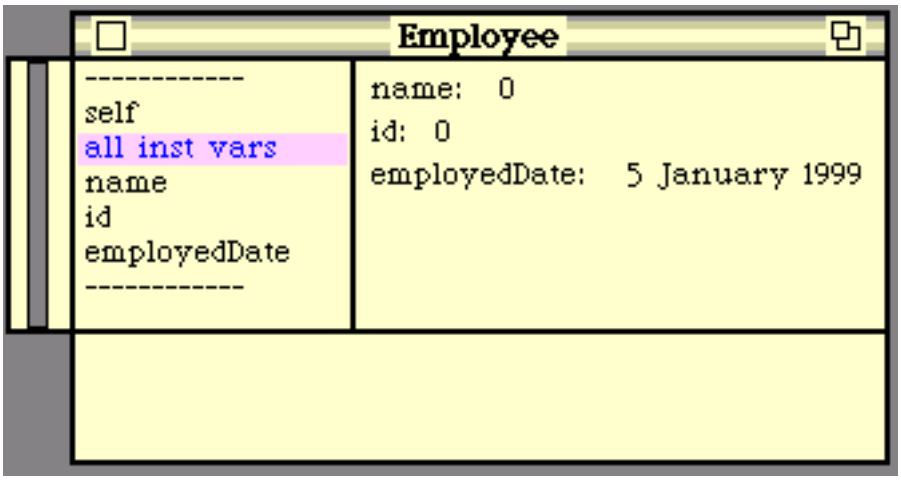
これを防ぐには「自分のinitializeの中でスーパークラスのinitializeを呼び出して、次に残りの処理を行う」とすれば良いことになります。

つまり、

```
Initialzie
        super initialize..  "<=スーパークラスのinitializeを起動"
        Id := 0.
        employedDate := Date today.
```

となります。

このようにEmployeeのメソッドを変更すると、インスペクタの結果は、



super initializeを含むEmployeeのインスタンス

となり、無事に初期化を行わせることができました。

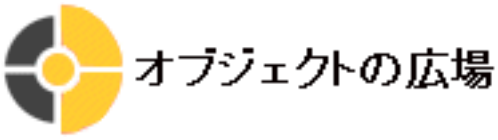
superはサブクラスでメソッドの前処理や後処理を付け加える際に使うことができます。

スーパークラスであるメソッド、operationが定義されているとして、サブクラスでは、

```
operation
        "前処理の記述"
        *.
        super oeration
        "後処理の記述"
```

といった具合です。実装コードが継承により再利用できていることが確認できるかと思います。

SmallTip: 自身を表すオブジェクトはselfの他superがある。この場合、メッセージに対応するメソッドの検索は自分のスーパークラスを参照して行われる



［Happy Squeaking!!］

3. 継承（2）

3. 1 抽象クラス

分類概念の上下関係にもとづいてクラスの継承をきちんと作成していくと、「実際にはインスタンスを作成する必要のないクラス」が存在することになります。

例えば、ペットブリーダーを支援するアプリケーションを作成していたと考えてみましょう。様々な動物をプログラム上管理する必要があるため、

哺乳類
　　犬
　　　　柴犬
　　　　チワワ
　　　　コリー
　　猫
　　　　…

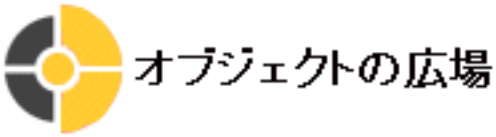
といった形でクラスを定義していたとします。

実際に、ペットブリーダーが育てるのは、「(芝犬の)ポチ」や「(コリーの)ラッシー」です。この場合に、「哺乳類」「犬」「猫」といったクラスは、主として分類の適切な階層を形成するために存在し、実際のインスタンスは作成しないことになります。(なんだかよくわからない「哺乳類のインスタンス」をペットブリーダーが対象にすることはありえません)。

こうしたクラスを抽象クラスと呼びます。抽象クラスの「抽象」とは、決して「具体的なインスタンスが作成されない」という意味になります。(これに対し、実際にインスタンスが作成されるクラスを「具象クラス」とよぶ場合があります)。
ただし、抽象クラスの持つ「属性の定義」、「操作の定義」、「操作の実装」は依然としてサブクラスで継承され、利用されることになります。
例えば「哺乳類」には、「平均体温」といった属性の定義を行っておくことができ、「犬」には「お手」「おすわり」といった操作の定義、実装をすることができます。

抽象クラスは分類の概念をきちんと整理するのに役立ち、また、継承によるコードの再利用も促進します。

Prev. Index Next



［Happy Squeaking!!］

4. Squeak 演習：抽象クラスのある継承の作成

4. 1 サンプルコードの読み込み

それではSqueakを使い、抽象クラスの活用例をみてみることにしましょう。

今回は、

```
Animal(動物)
        Bird(鳥)
                Parrot(オウム)
                Swallow(ツバメ)
```

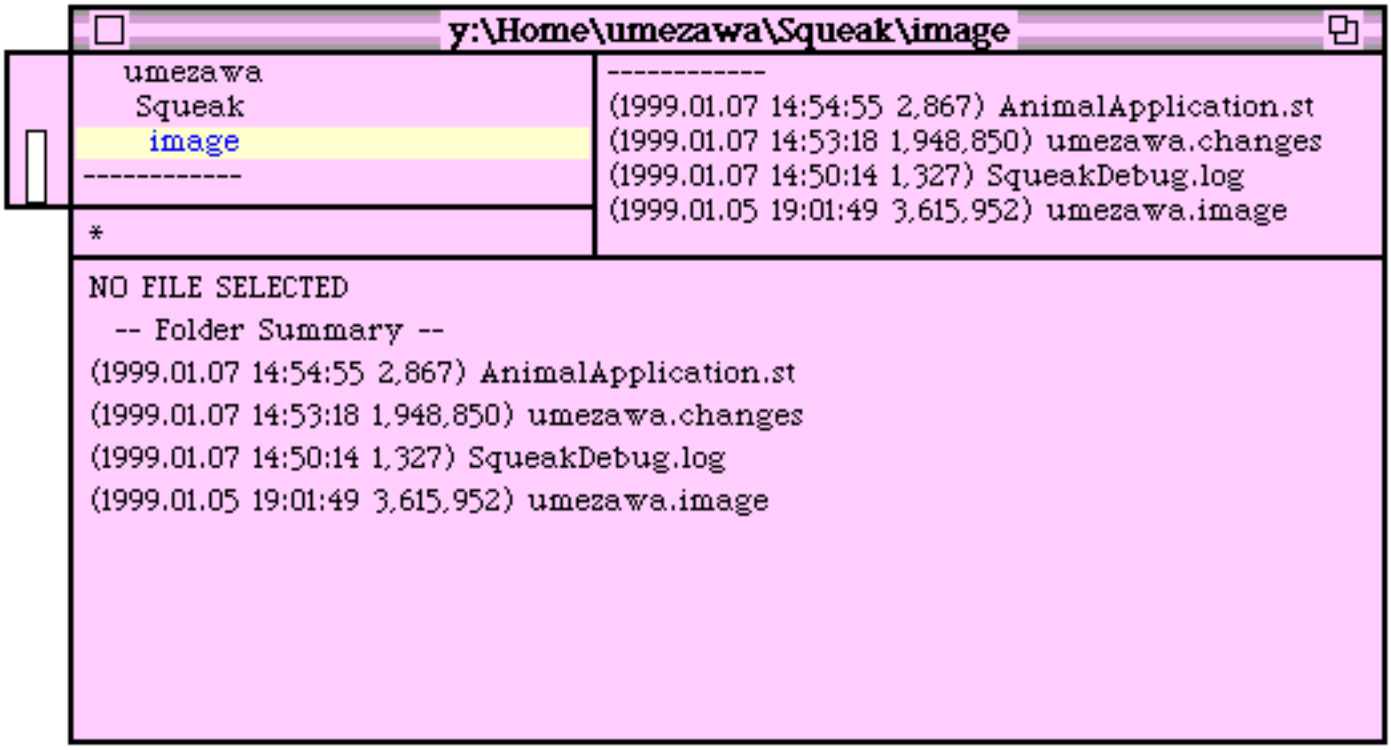
といった階層関係を作ることにします。

これだけのクラスをすべて定義していくのは大変ですので、できあいのソースコードをSqueakから読み込んでブラウズすることにします。

以下のリンクからソースのダウンロードをしてください。
AnimalApplication.st (<= CLICK NOW)

読み込みに便利なように、Squeakを起動したディレクトリに置いてください。
stファイルはテキストファイルでSmalltalkのソースコードが記述されています。Squeakでは、このファイルの読み込みは、フィイルリストのツールを用いて行うことができます。

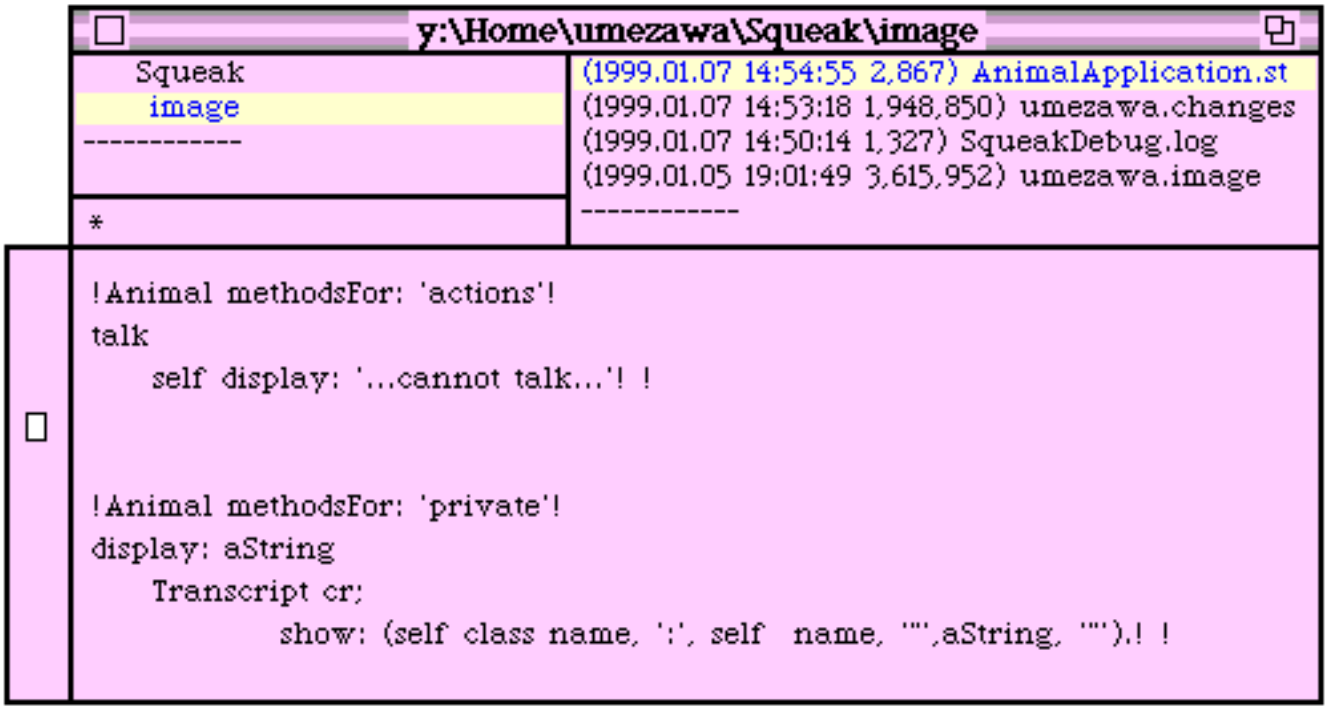
まず、メインのポップアップメニューから"open" -> "file list"と選択し、ファイルリストを起動します。



ファイルリストの起動

左上の小さなペイン(小窓)がディレクトリ構造を表しています。通常はSqueakを起動したディレクトリになります。
右上のペインでは、そのディレクトリ下のファイルをリストしています。

AnimalAppliation.stを選択すると、下に内容の表示がされます。



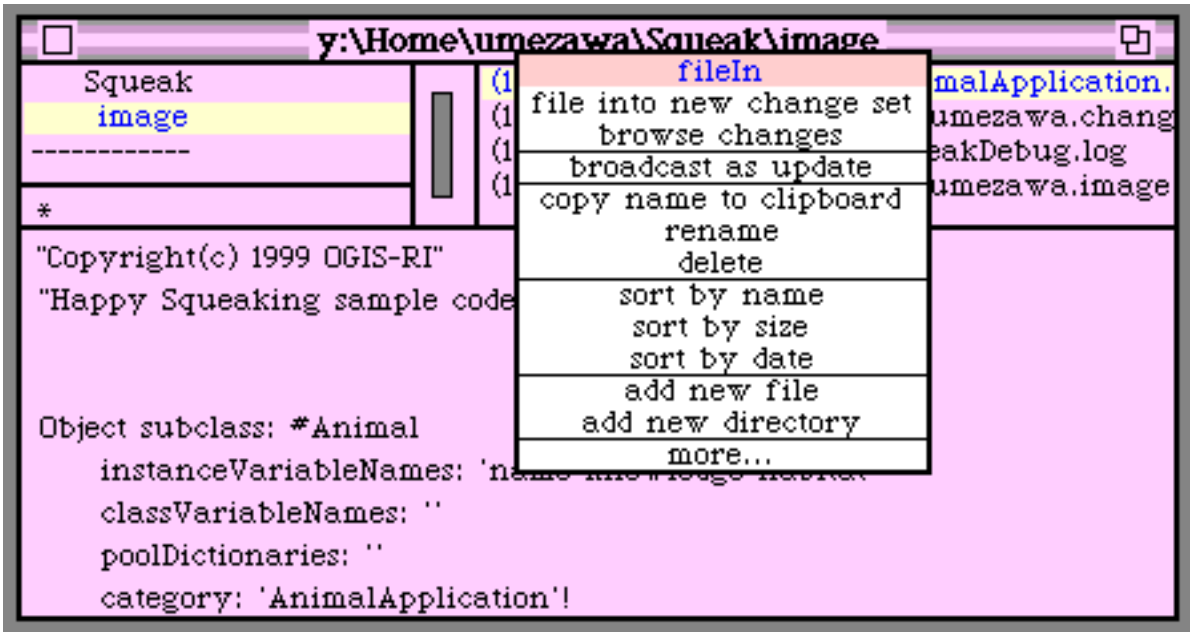
AnimaiApplicationの内容の表示

stファイルは、各クラスやメソッドの定義を!で区切った独特のフォーマットをしています。これはチャンクフォーマットとよばれます。チャンク(chunk)とは、肉の切り身などの断片を指します。Smalltalk内で一つのオブジェクトとして存在するクラ

スを、切り身にしてテキストファイルに展開したというわけです。

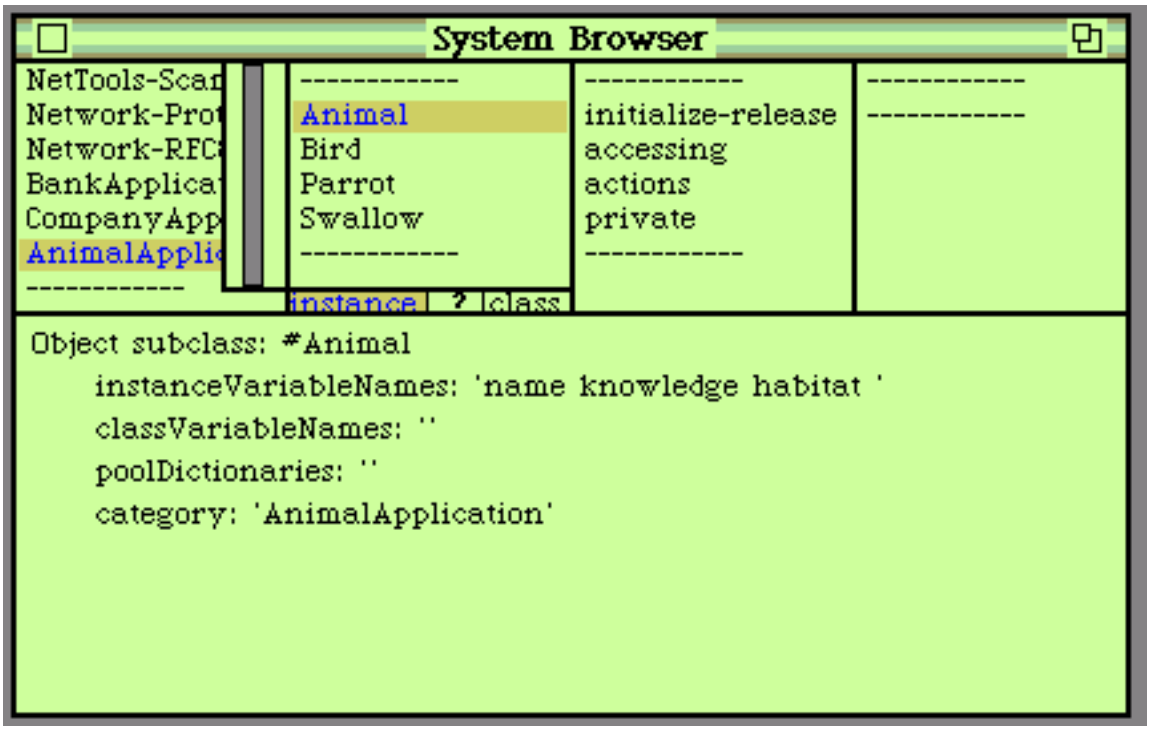
読み込みは、AnimalApplication.stを選択して右クリックによりポップアップメニューを出して行います。

ポップアップメニューの "file in"を選択してください。



fileInの選択

システムブラウザを更新すると、AnimalApplicationのクラスカテゴリが作られているのが確認できます。



読み込まれたAnimalApplicationクラス群

ちなみにソースコードの書き出しはシステムブラウザからポップアップメニューを出し、"file out"で行なうことができます。

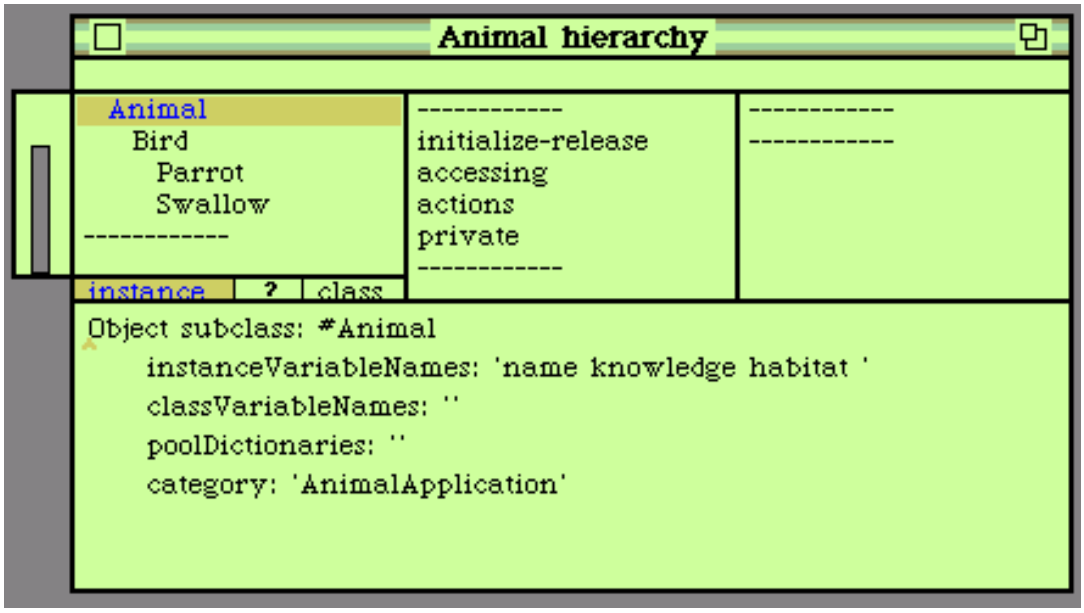
オブジェクトの広場

［Happy Squeaking!!］

4. Squeak 演習：抽象クラスのある継承の作成

4. 2 抽象クラスのブラウズ

まずAnimalについて"spawn hierarchy"で階層ブラウザを開きます。



Animalについての階層ブラウザを開く

Animalクラスでは、name, knowledge, habitatといった属性、talkといった操作が定義されています。

initializeメソッドでは、

```
initialize
        name := ''.
        knowledge := 0.
        habitat := ''.
```

といった形で、それぞれの属性を初期化しています。

また、talkメソッドでは、

```
talk
        self display: '...cannot talk...'
```

のような定義がされ、selfにメッセージを送ることで自分のクラスで定義した、display: メソッドを起動しています。

displayメソッドでは単純に、Transcriptに、自分のクラスの名前、自分の名前、引数として渡された文字列を表示します。

```
display: aString
        Transcript cr;
                        show: (self class name, ':', self  name, '"',aString, '"').
```

self class nameとself nameの違いに注意してください。前者は、インスタンスから、その属するクラスをclassメッセージで取り出し、そのクラスオブジェクトに対して、nameメッセージを起動しています。後者は、インスタンスに対して、nameメッセージを起動しています。クラスに対するメッセージ送信はnewのほかにも幾つかあり、nameはその例です。

では次にAnimalのサブクラスであるBirdを見てみます。

Birdでは、新たに canFlyという属性が付け加わりました。
操作としては、 flyが追加されています。

```
fly
        self canFly ifTrue:[
                self display: '*** FLYING ***'
        ]
```

ifTrue:[] という書き方がはじめて出てきましたが、とりあえずif文はこのように書くと覚えておいてください。self canFlyの返り値となるオブジェクトがtrueであれば、[]以下が実行されるということです。

例を示します。

```
( 'ABCD' size = 4 ) ifTrue:[ Transcript show: 'TRUE' ]  .
true ifTrue:[ Transcript show: 'TRUE'].
```

また、if文には幾つかバリエーションがあります。

```
false ifFalse:[ Transcript show: 'FALSE'].
( 'ABC' size = 3 )
        ifTrue:[Transcript show: TRUE'] ifFalse:[Transcript show: 'FALSE'].
```

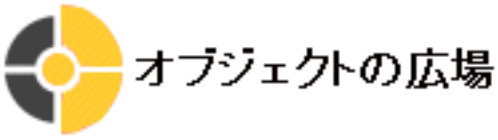
SmallTip: if文はifTrue:[]で記述できる
SmallTip: 真偽を表すオブジェクトはそれぞれ、true, false

initializeメソッドはオーバーロードされています。

```
initialize
        super initialize.
        habitat := 'sky'.
        canFly := true
```

superによりスーパークラスAnimalでの初期化を行ったあとで、変数 habitatの初期値が異なるので値を設定しなおしています。また、追加されたcanFlyをtrueオブジェクトに初期化しています。

Prev. Index Next



［Happy Squeaking!!］

4. Squeak 演習：抽象クラスのある継承の作成

4. 3 具象クラス Parrot のブラウズ

Animalや、Birdはどちらも抽象クラスであり、動物園アプリケーションでは実際にそのインスタンスが作られて使われることはありません。しかしこうすることで概念の整理ができアプリケーションの構造がわかりやすくなります。また、これらのクラスの中で属性や操作が定義、実装されているからこそ、Parrotといった具象クラスで、大幅にコードの量を減らすことができます。

Parrotでは、新たにvocabraryという属性が付け加わりました。
操作としては、initialize、talkのオーバーライドを行っています。

```
initialize
        super initialize.
        knowledge := 30.
        habitat := 'bird cage'.
        vocablary := ''
talk
        vocablary = ''
                ifTrue: [super talk]
                ifFalse: [self display: vocablary]
```

initializeにしてもtalkにしてもsuperを使い、有功なコードの再利用ができています。

抽象クラスにおいて、うまく属性、操作を定義し、実装しておくことができたのであれば、それを継承した具象クラスは、自らの性質として付け加わった最小限の部分だけを記述していけばいいということが確認できるでしょう。

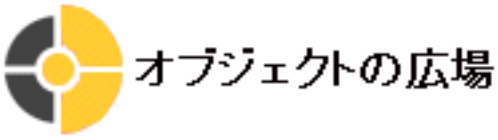
それでは、実際にオウムのオタケさんを生成し、しゃべらせてみます。

```
| otake |
otake := Parrot new initialize.
otake name: 'otake'.
otake talk.
otake vocablary: 'Hello...'.
otake talk.
otake fly
```

トランスクリプトの出力結果は以下のようになります。

```
Parrot:otake"...cannot talk..."
Parrot:otake"Hello..."
Parrot:otake"*** FLYING ***"
```

Prev. Index Next



［Happy Squeaking!!］

5. ポリモルフィズム

5. 1 多様性、多相性、多態性

「ポリモルフィズム」聞きなれない言葉です。ギリシャ起原の英語で、「多様の(poly)」と「形態をもっている状態(morphism)」の合成語です。オブジェクト指向を解説する日本語の書籍でも、この言葉は「多様性」、「多相性」、「多態性」いった様々な訳がされており、定まっていません。原語にせよ訳語にせよ言葉だけきくと非常に難しそうですが、内容は実はたいしたことはありません。

例えば、犬であるポチと、カンガルーであるパンチに対して、「走れ」と命じた場合、ポチも、パンチも、それぞれ「犬らしい」もしくは「カンガルーらしい」走り方で走ることになります。つまり、同一のメッセージ「走れ」に対して、それを受け取った側が、自分自身のクラスに記述されている「走り方」(メソッド)に従って、独自の動作をし、多様な振る舞いを見せることを指します。

以上がポリモルフィズムの説明です。非常に単純なことなので拍子抜けしてしまうほどです。

ただし、単純だからといって侮ってはいけません。この単純さこそが、ポリモルフィズムのパワーの源となっているのです。

オブジェクト指向の理想的な姿は、オブジェクトがそれぞれインテリジェントな実体として存在し、それらに対し、単にメッセージを送りさえすれば処理が行われるというものです。ポリモルフィズムは、このような姿の実現に役立つことになります。つまり、送る側が、複雑な部分を一切考慮せずに、「よきにはからえ」的な画一的メッセージを送れば、後は受け捕り側のオブジェクトでそれぞれが別個の多様な動作を行い、複雑な仕事をこなすわけです。

具体的に、疑似コードの例で示してみましょう。

例えば高速道路のゲート通過の際に車両に応じて料金を計算するシステムを開発していたとします。

トラックと乗用車、バイクでは、それぞれ料金の計算の仕方が変わってきます。この場合、計算結果を取り出す側(システムを使う側、以後クライアント側といいます)では、ポリモルフィズムが使えなければ以下のようなコードを強いられます。

```
//クライアント側（ポリモルフィズムが使えない場合）
calculate() {
Vehicle v := Gate.getVehicle(); //乗り物が取り出されるとする
Result ret; //計算結果の入る変数
Type type := v.getType(); //乗り物がなにかわかるようにタイプを取り出す。
        If (type == 'Truck') {
```

```
                ret := caluculateTrackOf( v ); //トラック用計算
        } elseif ( type == 'Car' ){
                ret := caluculateCarOf( v ); //乗用車用計算
        } elseif ( type == 'Bike' ) {
                ret := caluculateBikeOf( v ); //バイク用計算
        }
}
```

なんとも手続き的です。クライアント側は乗り物の種類に応じて計算結果をif文によって振り分けてあげることになります。
さらに乗り物の種類が新たに増えた場合（例えばBusが追加される）、クライアント側では、対応するif文を付け加えてあげなければなりません。

しかし仮に以下のようにクラスの継承関係が作られていたとします。

```
Vehicle
        Truck
        Car
        Bike
```

(Vehicleの下にTrack,Car,Bikeサブクラスが定義されている)。

そしてVehicleのサブクラスで独自の計算をするcalculate()操作の定義を行っていたとしたらどうでしょう。

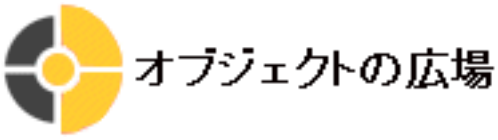
この場合、クライアント側は、ポリモルフィズムを利用して以下のように記述することができます。

```
//クライアント側（ポリモルフィズムが使える場合)
calculate(){
Vehicle v := Gate.getVehicle(); //乗り物が取り出されるとする
Result ret; //計算結果の入る変数
        ret := v.calculate(); //  「計算して」と頼むのみ
}
```

クライアント側の煩わしい世話やき部分(if文、if文の連続で処理を分ける)が消滅し、コード量が激減しています。
また、仮にBusが新たな乗り物として追加されたとしてもクライアント側では一切の変更の必要がありません。

継承は、メソッドの差分定義などによりクラスを提供する側のコードの再利用性を高めます。
一方で、ポリモルフィズムは、クラスを利用する側(クライアント側)のコードの再利用性を高めることになります。呼び出される側の変更の影響を受けにくくなり、クライアント側のコードはより長き時間、場面において使われていくことになります。

Prev. Index Next



［Happy Squeaking!!］

5. ポリモルフィズム

5. 2 明示的ポリモルフィズムと暗黙的ポリモルフィズム

実はポリモルフィズムには、2種類あります。明示的なものと暗黙的なものです。

例で説明しましょう。Web上で、本の販売を行うシステムを開発することを考えてみます。

本を著すBookクラスを作成し、クライアント側では、本の値段を取り出して表示する際に、Bookのインスタンスに calculatePrice()というメッセージを送っていたとします。

```
PassRegister(){
Result result := book.calculatePrice();
}
```

いままではこれだけで問題なかったのですが、Web上の販売が好評なので新たにCDも扱うようになりました。
CDの場合は、本と違って割り引きを行うことができます。
このためcalculatePrice()では、特定の割引率をかけた値をクライアント側に返さなければなりません。

「特に難しいことはない。単にcalculatePrice()の操作を、CompactDiscクラスに定義すればいいじゃないか」と思われることでしょう。

原則としては確かにそうなのですが、明示的なポリモルフィズムを要求する言語では、これだけではうまくいきません。
CompactDiscとBookの共通のスーパークラスとして、calculatePrice()操作の定義されたなんらかのクラスが必要になるのです。

解決策としては、CalculatableProductといった抽象クラスを用意し、これにcalculatePrice()という定義だけで実装のない操作(抽象メソッド)を定義しておきます。
そのサブクラスとしてBook、CompactDiscを定義し、calculatePriceを各々がオーバーロードすることになります。

このように、明示的なポリモルフィズムは、ポリモルフィズムが有効である範囲を共通のスーパークラスの宣言という形で示します。
スーパークラス、サブクラスの関係がある場合でのみポリモルフィズムが働くというわけです。

```
PassRegister(){
CalculatableProduct prod;
```

```
        //ポリモルフィズムを有効にする共通のスーパークラスを宣言
Result result := prod.calculatePrice();
}
```

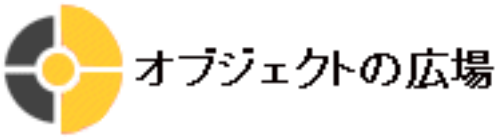
一方で、暗黙的なポリモルフィズムでは、このような制限はありません。BookがPublication(出版物)のサブクラス、CompactDiscがMusicMediaのサブクラスであり、お互いに共通のスーパークラスがなかったとしても、それぞれがcalculatePrice()の操作を定義していさえすればいいということになります。

どちらの立場と捕るべきかは一概にはいえず、それぞれに長所、短所があります。暗黙的ポリモルフィズムの言語では、クライアント側の再利用性は非常に高まることになります。当初は予想されていなかった新たなクラスが、呼び出し先で定義され、そのインスタンスがクライアントにやってきた場合でも、クライアントに変更の必要が生じないからです。

一方で明示的ポリモルフィズムのように、スーパークラスの宣言という形でポリモルフィズムの有功な操作を明示的に宣言し示しておいたほうが、よりわかりやすくなると考えることもできます。また、コンパイラの都合ですが、起動する操作の有効範囲が限定されているので、より効率の高い実行コードを作成できる可能性もあります。

Smalltalkでは、暗黙的なポリモルフィズムを採用する立場をとっています。これは、オブジェクト指向の再利用性、柔軟性を最大限に活かすという設計思想にもとづいたものと考えられます。一方で実行効率の良さを追求すると、C++やJavaのように明示的ポリモルフィズムを採用することになります。

Prev. Index Next



［Happy Squeaking!!］

6. Squeak 演習：ポリモルフィズムに親しむ

6. 1 Ｉｆなしのプログラム

ポリモルフィズムをうまく使うことによって、クライアント側のコードからif文による条件分岐を全くなくしてしまうことができます。

その極端な例はif文自体をポリモルフィズムでシミュレートしてしまうことです。

以下に例を示してみましょう。

まず、条件があった場合に起動されるアクションを表現するために、以下のようなクラスを定義します。

```
TranscriptAction( トランスクリプト表示アクション {抽象クラス})
        ShowHeloWorldAction ('HelloWorld'と表示するアクション)
        ShowByeAction ('Bye'と表示するアクション)
```

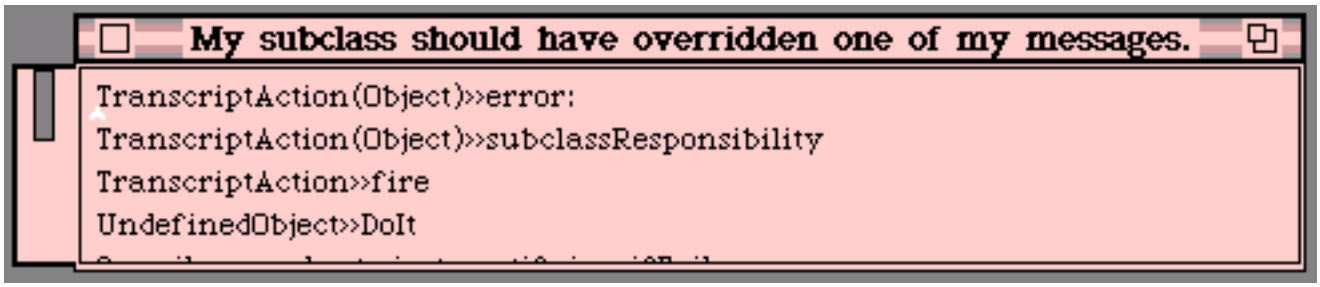
そしてそれぞれ

```
TranscriptActionのfireメソッド (actionsカテゴリ)
fire
        "抽象メソッド、サブクラスが実装すべきなのでエラーを返す"
        self subclassResponsivility..
ShowHelloWorldActionのfireメソッド (actionsカテゴリ)
fire
        Transcript cr; show: 'HelloWorld'
ShowHelloWorldActionのfireメソッド (actionsカテゴリ)
fire
        Transcript cr; show: 'Bye'
```

のようにメソッドの実装を行ないます。

Smalltalkは、暗黙的なポリモルフィズムをサポートするので、この場合特にTranscriptActionクラスの定義は必要ありませんが、わかりやすさのためにあえて作成しています。
fireメソッドは、各サブクラスで適切に記述されるので、抽象クラスのレベルでは実装する必要がありません。宣言だけで実体のない、抽象メソッドとなります。この場合、fireメソッドの実装を何も書かない状態にしておいてもいいのですが、fireをサブクラスで適切にオーバーライドしてほしいということを示すために、self subclassResponsivilityと記述しておいたほうが親切です。

なぜならself subclassResponsivilityと書かれたメソッドが起動されると、Squeakでは以下のようなノーティファイアが開くようになっているからです。



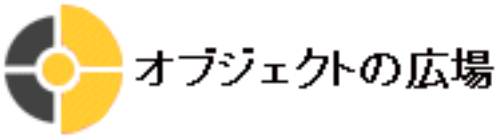
subclassResponsivilityメソッド起動によるノーティファイアの出現

仮にサブクラスでfireメソッドを定義し忘れたとすると、スーパークラスTranscriptActionでのfireメソッドが起動され、実行時に注意を促すことになります。

SmallTip: 抽象メソッドであることを示すには、メソッド本体に、self subclassResponsivilityと記述する

とりあえずここまでのソースをFileInしましょう。
FileIn: IfLess-Action.st (<=Click)

Prev. Index Next



［Happy Squeaking!!］

6. Squeak 演習：ポリモルフィズムに親しむ

6. 2 条件クラスの作成

次に真、偽の条件を表すために以下のようなクラスを定義します。

BooleanCondition (真偽条件 {抽象クラス})
TrueCondition (真 条件)
FalseCondition (偽 条件)

BooleanConditionのconditionTrue:conditionFalse:メソッド (actionsカテゴリ)

```
conditionTrue: oneAction conditionFalse: theOtherAction
        self subclassResponsibility
```

TrueConditionのconditionTrue:conditionFalse:メソッド (actionsカテゴリ)

```
conditionTrue: oneAction conditionFalse: theOtherAction
        oneAction fire
```

FalseConditionのconditionTrue:conditionFalse:メソッド (actionsカテゴリ)

```
conditionTrue: oneAction conditionFalse: theOtherAction
        theOtherAction fire
```

真の条件(TrueCondition)の場合は、conditionTrue:conditionFalse:メソッドの、conditionTrue: 側の引数として与えられたoneActionオブジェクト(TranscritpActionのサブクラスのインスタンス)に対してfireメッセージを送っています。
偽の条件(FalseCondition)の場合は、逆にconditionFalse:の引数として与えられたtheOtherActionオブジェクトに対し、fireメッセージが送られることになります。

FileIn: IfLess-Condition.st (<= Click)

最後に条件を擬似的に生み出すためのテスト用のクラスを作成します。

BooleanConditionMakerというクラスを作成します。

BooleanConditionMakerのrandomConditionメソッド (actionsカテゴリ)

```
randomCondition
        10 atRandom < 5
                ifTrue:
                        [Transcript cr; show: 'TRUE'.
                        ^ TrueCondition new]
                ifFalse:
                        [Transcript cr; show: 'FALSE'.
                        ^ FalseCondition new]
```

randomConditionメソッドの内部では、1から10までの乱数を発生させ、場合に応じ

てTrueConditionやFalseConditionを返すようにしています。
(この中ではやむをえず本当のif文を使っています)

FileIn: IfLess-Test.st (<= Click)

> 整数オブジェクト にatRandomメッセージを送ると乱数を得ることができますが、これはSqueakに用意されている独自のAPIといえます。標準的なSmalltalkの書き方では、Random new nextで乱数を得ることになります。(Squeakでもこの方法が使えますが、単純化のためにatRandomの方を使っています)。

これで全ての準備が整いました。ワークスペースで実行してみましょう。
Smalltalk組み込みのifTrue:ifFalse:文(もどき)のように見えますでしょうか。

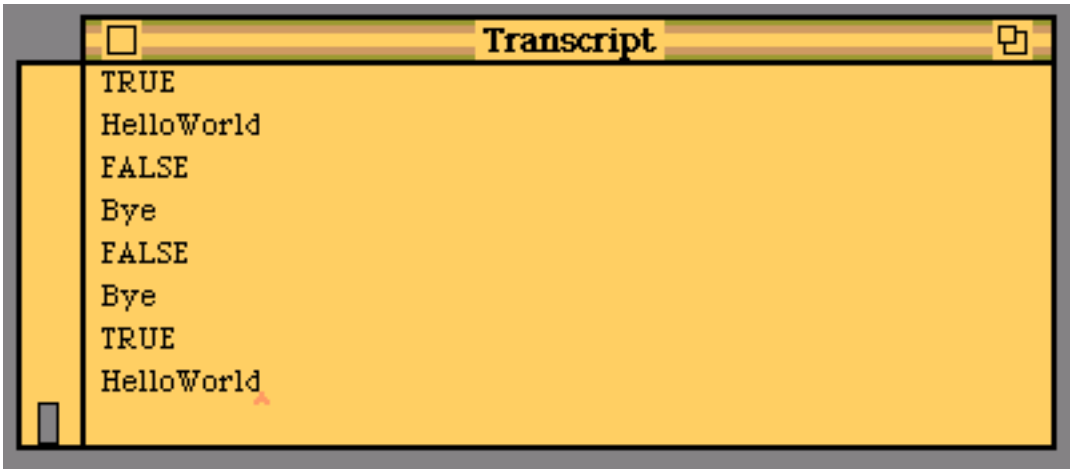
```
| condition |
condition := BooleanConditionMaker new randomCondition.
condition conditionTrue: ( ShowHelloWorldAction new)
            conditionFalse: ( ShowByeAction new).
```

この場合、条件側も、アクション側も、両方ポリモルフィズムを使うことになります。流れを説明しましょう。

BooleanConditionMakerは乱数によってTrueConditionか、FalseConditionのいずれかのインスタンスを作ります。
変数conditionにはそのインスタンスが入り、それに対してconditionTrue:conditionFalse:のメッセージが投げらます。
ここでまず条件側(BooleanConditionのサブクラス群)でポリモルフィズムを使うことになります。
つまりBooleanConditionのサブクラスは、それぞれのやり方でconditionTrue:conditionFalse:を処理します。(TrueConditionならばconditionTrue:側の引数として入ってきたオブジェクトにfireを送る。FalseConditionならばその逆)。

引数には、TranscriptActionのサブクラスのインスタンスが入っており、fireが送られたときに("HelloWorld"や"Bye"とTranscriptに表示するなど)それぞれの振る舞いをすることになります。ここで、アクション側でポリモルフィズムが使われています。

何度か"do it"してみてください。
条件に応じて異なるアクションが実行されることが確認できます。



if-lessプログラミングの実行結果

オブジェクトの広場

［Happy Squeaking!!］

6. Squeak 演習：ポリモルフィズムに親しむ

6. 3 実際の ifTrue:ifFalse は？

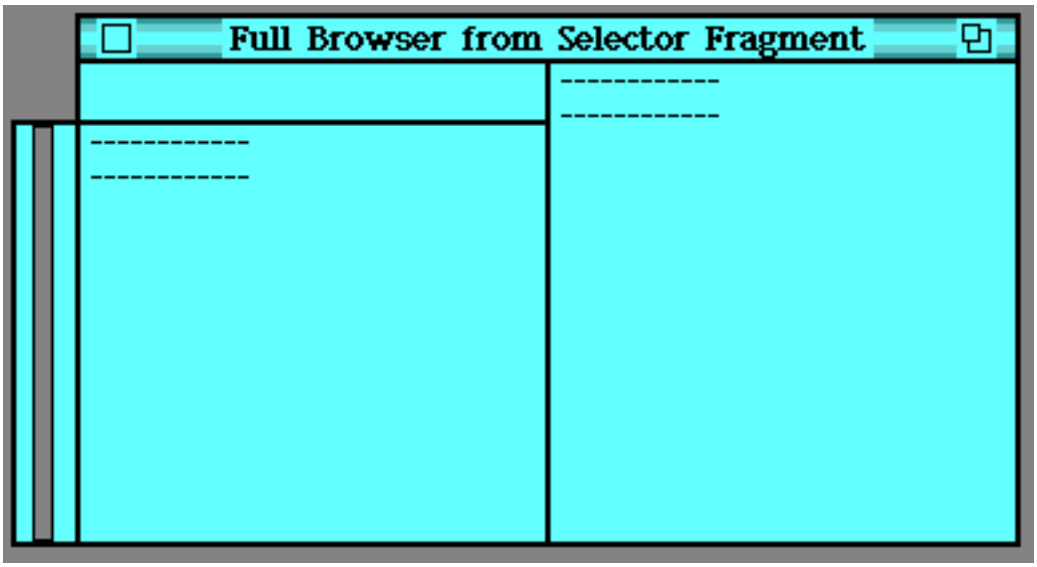
今までSmalltalkのif文といっていたifTrue:ifFalse: メソッドも、実は基本的に同じ方法を使用しています。

```
Boolean
        True
        False
```

というクラス階層が形成されており、ifTrue:ifFalse:といったメソッドがポリモルフィズムを使って定義されているのです。
さっそく調べてみましょう。
メソッドの名前がまずわかっていて、そこから実装されているクラスをみたいといったときには、Squeakでは、Selector Finderというツールを使うと便利です。

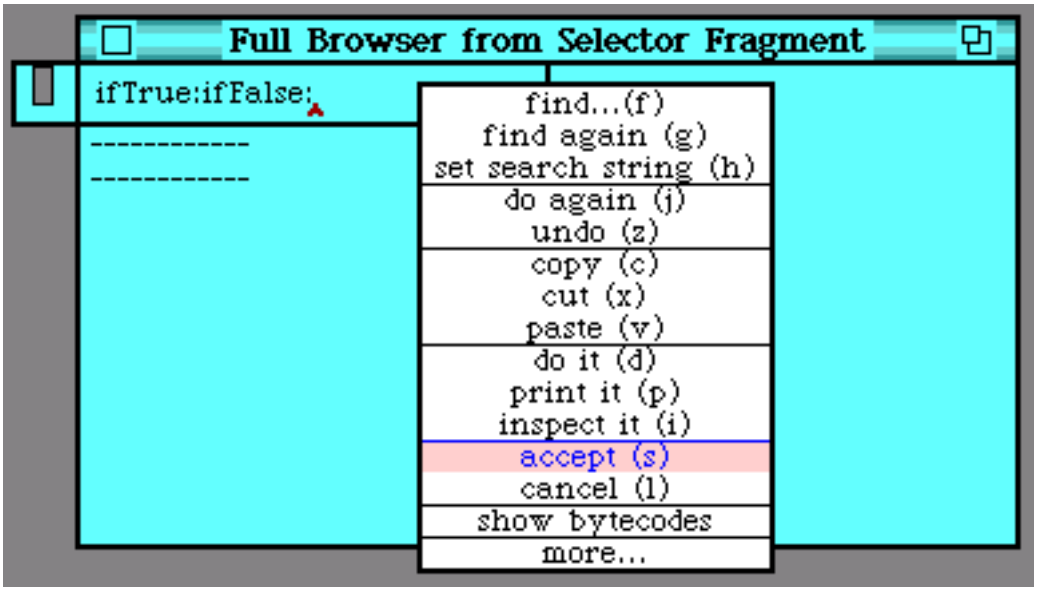
メインメニューの"open" -> "selector finder"を選択します。

以下のような画面が立ち上がります。



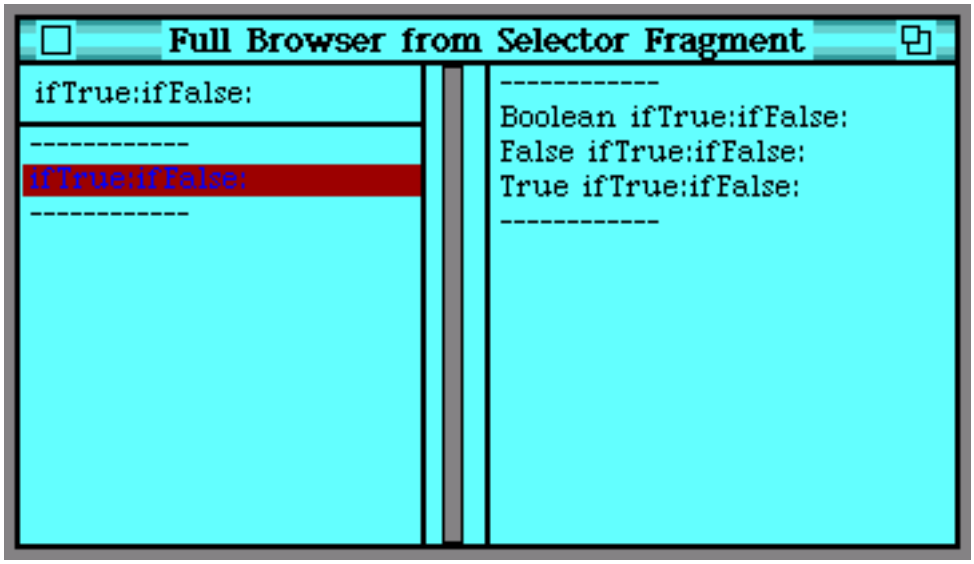
Selector Finderの起動

左上のペインにifTrue:ifFalse:と打ち込んで、右クリックでポップアップメニューを出し"accept"してみてください。



Selector FinderによるifTrue:ifFalse:の検索

検索結果は右のリストに表示されます。



検索結果の表示

クリックするとブラウザが開き、メソッドの内容を見ることができます。

先ほどのBooleanConditionとほぼ同じような実装がされていることが確認できるでしょう。

例えばTrueでは以下のようになっています。(コメントは省略)

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

        ^trueAlternativeBlock value
```

引数は先ほど作成したTranscriptActionではなく、BlockContextというクラスになります。
このクラスのインスタンスは、[]で括ることでリテラルとしてすぐに作成できます。[]の中には任意のSmalltalkの表現をかくことができます。valueメッセージが送られることで、中に書いた表現が実行されます。([]はSmalltalkではブロックと呼ばれます)。

例:

```
| helloBlock |
helloBlock := [ Transcript cr; show: 'HelloWorld' ].
Transcript cr; show: 'not yet done'.
helloBlock value.
```

上記の例では、一行目では単にBlockContextのインスタンスを生成したのみで、中身の実行はされていません。三行目のvalueが送られて、初めて実行になります。

> BloclkContext(実装によってはBlockClosure)はSmalltalkをSmalltalkたらしめている非常に強力な機構なのですが、ここではブロック自体についてこれ以上の解説はしません。

IfTrue: [] などで引数として与える[]も当然このブロックです。
以下のコードを実行してみてください。

```
| helloBlock |
helloBlock := [ Transcript cr; show: 'HelloWorld' ].
(1<3) ifTrue: helloBlock.
```

1<3の結果、返ってくるのは、trueオブジェクトです。これはTrueクラスのインスタンスに他なりません。(falseオブジェクトFalseクラスのインスタンスです)。True、Falseのそれぞれのポリモルフィズムを利用した振る舞いの変化によって制御構造が実現できるているのが確認できます。

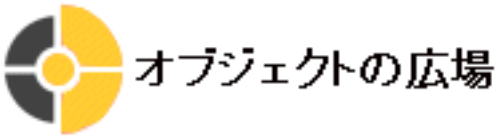
ポリモルフィズムをうまく使うことでif文の入り込む煩雑なコードを排除していくことができます。Smalltalkにおいては、if文ですらポリモルフィズムを使って実現しているので、実質if文はゼロということもできます。ポリモルフィズムを究極まで使ってみせている驚きの姿です。

> ご安心ください。実際には、ifTrue:ifFalse:のようなメッセージの送信を実行時に正直におこなっていたのでは、パフォーマンスが遅くなるので、Squeakを含むSmalltalkの実装の多くはコンパイル時にこの表現をインライン化して直接バイトコードにしています。(つまりこのメソッドは通常は起動されません)。perform:というインライン化を防ぐような書き方をすることにより、ifTrue:ifFalse:の本当の送信を行なわせることもできますが、今回の範囲外です。

###

さて、今回で基本部分は一応終了です。ポリモルフィズムまで理解できれば、オブジェクト指向の初級レベルはクリアできたといえるでしょう。
次回は、メタ機能を説明していきます。他のオブジェクト指向言語ではみられないSmalltakならではパワーも垣間みることができます。お楽しみに。

Prev. Index Next



［Happy Squeaking!!］

7. 参考文献

●和書

「サクサクSmalltalk」東京電機大学出版 1996

商用のSmalltalkであるVisualWorksで解説を行っている入門書。"The Art and Science of Smalltalk" Prentice Hall 1995 の翻訳 。MVCやアダプタについての解説もある。

「Smalltalkイディオム」 ソフトリサーチセンター 1997

日本のSmallalkerとしては第一人者である青木淳さんの書き下ろし。スレッドや例外処理などアドバンスな内容を含む。

● 洋書

"Smalltalk, Objects, and Design" Prentice Hall 1996

Smalltalkを学びながら、オブジェクト指向、デザインパターンについて学 んでいく。本講座のスタンスに最も近い。入門書でありながら洞察は深い。

"On to Smalltalk" Addison Wesley 1998

日本でも「ウィンストンの..」で多くの訳が出ているPatrick Henry WinstonによるSmalltalkの入門書。クックブックスタイルで簡潔に説明がされており実用的。日本語訳も近いうちに出版される。

"Handbook of Programming Languages Vol.1 Object-Oriented Programming Languages" Macmillan Technology Publishing 1998

Part II のSmalltalk編に、Adele GoldbergによるSmalltalkの歴史、 Allen Wirfs-Brocksによる文法解説がある。文法解説は非常に簡潔で一見の価値あり。

Let's "do it"!!

Prev. Index