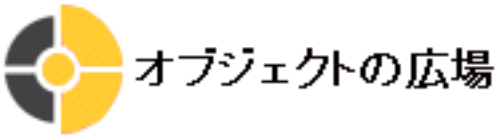


［Happy Squeaking!!］

― オブジェクト指向再入門 ―

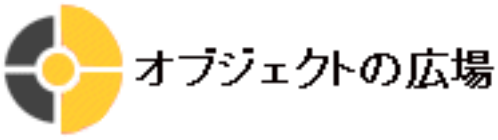
[第一回] [第二回] [第三回] [第四回] [第五回]

第四回：メタ機能との出会い

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



［Happy Squeaking!!］

1．メタ機能との出会い

1．1　"メタ"とは

Metaは、英語の接頭辞で、もともと「背後の、後ろの、より包括的な、超えた」といった意味があります。私たちが普通に持っている観点からは、少し見えにくい、超えてしまったところにメタの世界はあります。とはいえ、不必要というわけではありません。むしろ縁の下で私たちの世界を支えてくれている重要な役割を持っているのです。メタなるものの存在によって、私達は世界を混沌ではなく、整然とした秩序だったものとしてとらえ、無事にすごすことができます。

身近な例で見てみましょう。オブジェクト指向では、現実世界を様々な「もの」の集まりからなるというふうに考えます。「もの」にはいろいろな実体(インスタンス)や、インスタンスを分類するもの(クラス)があります。個々のインスタンスが成り立つための秩序を与えるものとして、おなじみのクラスが存在しています。
「犬」というクラスが「ポチ」の背後に存在することによって、はじめて「ポチ」が何たるを理解できるのです。「犬」という概念が全くない世界ではじめて「ポチ」を見つけたとしたら、私たちはそれを得体のしれないものとしてしかとらえられず気味が悪いと思うことでしょう。こうした時に「犬」クラスは「ポチ」インスタンスに対する「メタ情報」を与えていることになります。

記号風に書くと以下のようになります。
「ポチ」=meta =>「犬」(犬はポチが何たるかのメタ情報を与えてくれている)

これはインスタンス工場としての「犬」が、「ポチ」という実体を生成すると考えた場合の実体化（instantiate）の関係とはちょうど逆になります。
「ポチ」<=instantiate =「犬」

このような説明だと、「インスタンスからクラスへの関係がメタなのか」と思ってしまうかも知れません。
しかし、実は話はそれほど簡単ではありません。

山田さんは車の購入を考えているとします。山田さんのところに実際に納車されるのは「インフィニティ」という車種そのものではありません。「品川 XX-XX」ナンバーが張られた「インフィニティのインスタンス(山田号)」のはずです。
ここまでは、前の例と変わりません。「インフィニティ」は「山田号」に対しクラスとしてメタ情報を提供しています。このため町中で山田さんの車を見ても別段気味が悪いと思うこともなく平穏にすごせます。

「山田号(one of インフィニティ)」=meta=>「インフィニティ」

ところが今度は車を購入する場合を考えてみます。購入者は、「購入の基準」の要素として、「値段」や「燃費」、「安全性」といったものを考えます。山田さんの

場合は、ミーハーなのでとにかく「(インフィニティという)車種」で決めてやろうと思っていたとします。この場合、「インフィニティ」は、「車種の1つ」(インスタンス)であり、「車種」は「インフィニティ」が何たるかを説明するメタ情報をクラスとして提供していることになります。

「インフィニティ(one of 車種)」 =meta=>「車種」

こう考えると「インフィニティ」はある場合ではクラスで、またある場合ではインスタンスということになってしまいます。
まとめると、以下のような関係が成り立っています。

「山田号(one of インフィニティ)」=meta=>「インフィニティ(one of 車種)」=meta=>「車種」

この例が示すように、メタというのは相対的な考えであり、自分がどこを視点として選ぶかによって推移するものです。「山田号」を実体としてとらえればそのメタ情報を提供するものは「インフィニティ」というクラスです。「インフィニティ」を実体としてとらえれば、今度は「車種」がそのメタ情報を提供していることになります。(「山田号」から見れば「車種」はメタ情報のさらにメタ情報を提供しています。「インフィニティ」がクラスであり「車種」はクラスのさらにメタ情報を提供するもの、つまりメタクラスとして考えることができます)。

一般的な言い方をすれば、インスタンスに存在する基盤をクラスが与え、クラスが存在する基盤をメタクラスが与えていることになります。そうするとメタクラスが存在する基盤は?ということになるのですが、メタは相対的な考えなので、抽象度の視点をずらしていくことによって無限に続いていくものということになります。とはいえ実際にはあまりに抽象的なレベルになると、私達の思考をまさに「超えたもの」としてもはや意味を持たなくなります。

しかし妥当なレベルでのメタ的なものの抽出は、オブジェクト指向の分析、設計、また言語の実装においても非常に有功な効果をもたらします。以下、そうした例をみていくことにしましょう。

Index Next

オブジェクトの広場

［Happy Squeaking!!］

2. 自分を省みるオブジェクト

2. 1 イントロスペクション

山田さん宅に見知らぬ人がいきなり訪問してきたとします。山田さんはその人に対しどのようにコミュニケーションを試みるでしょうか?
まずは、その人が一体「何者であるか」という情報をつかみたいと思うのではないでしょうか。つまり「新聞の販売員」か、「健康食品のセールスマン」か、「医者」か「遠い親戚」なのかといった情報です。そこで山田さんは「どなたですか」といったメッセージによってその人に対するバックグラウンドの情報を得ました。その結果、「新聞の販売員」ということがわかったので、続いて「あと3ヶ月延長してください」といったメッセージを送ることができ、コミュニケーションが成立しました。
この場合、「尋ねてきた人」はインスタンスとして考えることができます。「山田さん」は「尋ねてきた人」にその人のクラスを尋ねるメッセージを送ることによって「尋ねてきた人」が「新聞の販売員」に属するというメタ情報を得たわけです。
このように、実体に対して「そもそもあなたは何なのか」といった質問を投げかけることができるのは非常に大事なことです。メタ情報が得られることで、インスタンスの正体がわかり、メッセージ送信側は、本来ダイレクトにメッセージが送れないような疎遠なインスタンスに対してさえ、柔軟にコミュニケートしていくことが可能になります。

今度はもうすこしコンピュータよりの例でみてみましょう。
RDB(リレーショナルデータベース)は、データを表の集まりとして蓄積するデータベースです。通常は、RDBから、既に知っている表構造(社員表)の中に存在する実データ (社員:山田) を取り出しています。しかし場合によっては、データを取り出す側が、RDB内の全ての表構造についての情報をあらかじめ把握できない場合があります。例えば、表の内容をテキスト出力する汎用的なデバッグ用ツールを作成する場合には、特定の表を想定するということはできません。また、データウェアハウスなど、非常に多量、多種のデータがデータベースに格納されている場合には、やはりクライアント側ですべての表構造を知っておくというのは困難になります。そこで使われるのがメタ情報です。RDBでは、現在格納されている表の名前、カラムの各項目の名前、型といった、データそのものではなく、そのデータを定義するためのメタなデータを必ず問い合わせることができるようになっています。社員そのもののデータではなく、その社員が存在するために必要な「社員表」の構造に関するデータを取り出すわけです。このような機構を提供しておくことで、クライアント側のデータベースに対する相互の依存関係を減らし、より柔軟なシステムの構築ができるようになるのです。

また分散オブジェクト(CORBA)の世界では、通常、相手先のオブジェクトのインターフェースをあらかじめ知った上でのオペレーション起動(静的起動)に加えて、相手先のインターフェスを尋ねてからのオペレーション起動(動的起動)がサポートされています。ここでは、インターフェースリポジトリというメタデータが一元管理され

るデータベースが背後で活躍しています。

疑似コード風に書くと以下のようになります。

```
"クライアント側"
server := aNamingContext.resolve('Server'). "'Server'という名のサーバを検索"
interfaceDef := server interface. "サーバのインターフェースを取り出す"
interfaceName := interfaceDef name. "インターフェースから名前を取得"
(interfaceName = "newspaperSales")  "新聞屋さんであれば"
        ifTrue:[
                request := interfaceDef operationAt: ' prolongNextMonths:'.
                result := request invokeWith: 3. ].
" prolongNextMonths: というオペレーションを引数3で起動"
```

分散システムや、Webのシステムのように非常に広範囲で動作するシステムの場合には、あらかじめ起動するクライアント側が、サーバの提供するサービスをすべて知っておくということは実質不可能なことです。このために、不特定のサーバという実体に対して、どのようなインターフェースを持つか(何者であるかを)尋ねる機能が非常に有効になるのです。

このように、システムが自分自身を省みて「自分は一体なんなのか」を把握できること機能を、イントロスペクションといいます。
イントロスペクションは「自らを振り替える(自己反省)」という意味です。オブジェクト指向の場合では、特定のオブジェクトに対して、「あなたはいったいどのようなオブジェクトですか」とメッセージが投げられたときに、その実体のメタ情報を得ることができれば、イントロスペクション機能かサポートできているということになります。

Prev. Index Next

オブジェクトの広場

［Happy Squeaking!!］

3. Squeak演習:イントロスペクション

それでは、Squeakを使い、オブジェクトのイントロスペクションを試みることにしましょう。
本演習では、Squeak内で最初から提供されているクラスである、OrderedCollection(順序付き集合)を、メッセージを送ることで、様々な角度から調べあげていきます。

イントロスペクションの提供により、あらかじめ情報のない疎遠なオブジェクトに対しても、段階的に機能を調べて使用できるようになる柔軟性と楽しさをつかんでください。

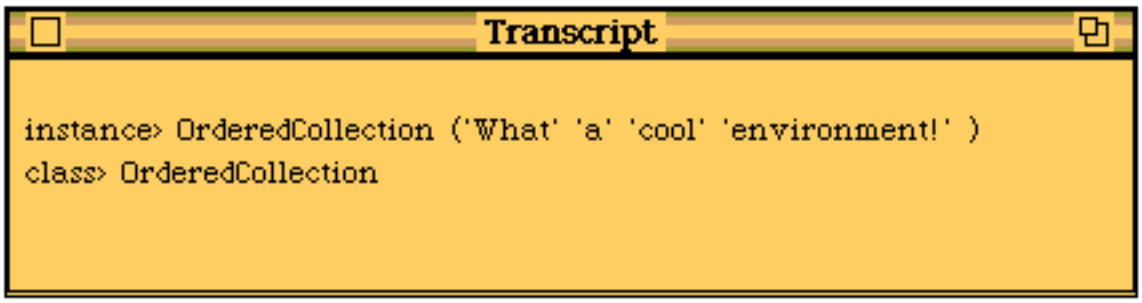
3. 1 オブジェクトの構造を調べる

クラスを取り出す

まずは、小手調べとしてOrderedColectionを生成してクラスを聞いてみます。
ワークスペースに以下を書いて実行してください。

```
| ord |
ord := OrderedCollection new.
ord add: 'What'; add: 'a'; add: 'cool'; add: 'environment!'. "要素を追加"
Transcript cr;  show: 'instance> ', ord printString. "インスタンスを表示"
Transcript cr;  show: 'class> ', ord class printString.
        "インスタンスからクラスを取り出し表示"
```

トランスクリプトには以下のように表示されます。



OrderedCollectionのインスタンスとクラス

OrderedCollectionは、追加した順番に要素を格納していく入れ物です。入れ物のサイズは要素の追加に従って自動的に変化します。(JavaでのVectorに該当)
Smalltalkでは使う機会の多いデータ構造系のオブジェクトです。

SmallTip: データを追加した順番に格納しておくにはOrderedCollectionを使う

以前に紹介したように、インスタンスに対して、「あなたを生成したクラスは?」と尋ねるときには class メッセージを使います。

SmallTip: インスタンスに対して、生成したクラスを問い合わせるには"class"メッセージを用いる

Smalltalkでは、クラスもれっきとしたオブジェクトですから、クラス自身がメッセージに答えられます。

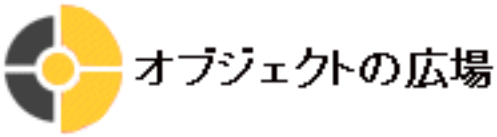
ですから上記の例は、わざわざインスタンスを生成しなくとも、

```
Transcript cr; show: 'class> ', OrderedCollection printString.
```

と書けば十分で、まったく同じ結果を得ることができます。

更につづく

Prev. Index Next



[Happy Squeaking!!]

3. Squeak演習:イントロスペクション

3. 2 オブジェクトの振る舞いを調べる

特定メッセージに答えられるかを聞く

次はOrderedCollectionのインスタンスに対して、特定のメッセージが答えられるかを尋ねてみることにしましょう。
ワークスペースで以下を"print it"してください。

```
| ord |
ord := OrderedCollection new.
ord respondsTo: #add:. "OrderedColectionにadd:メッセージが答えられるか聞く"
        => true
```

respondsTo: の後に書くのはメッセージ名(セレクタ)です。メッセージ名にパラメータの名前は含まれません。例えば引数が二つの場合は、#at:put:という形になります。

> 先頭に#のついた文字列は、通常の文字列ではなくSmalltalkではシンボルと呼ばれています。これは、書き換え不能な文字列でSmalltalkのVM内に同一内容の文字列が必ず一つしかないことが保証されるものです。(#add: というシンボルを作成したらそのコピーは存在できない)。高速な検索ができるため、辞書のキーなどによく利用されます。

respondtdTo: の後をいろいろと変えて試してみましょう。
うまくいけばadd:のほかにもOrderedCollectionインスタンスの理解できる操作が見つかるかもしれません。

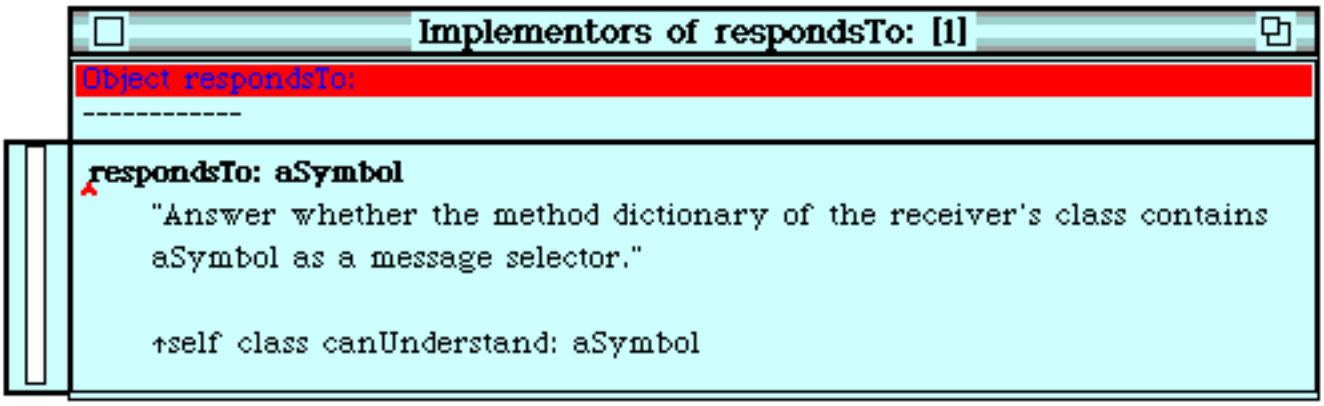
```
| ord |
ord := OrderedCollection new.
ord respondsTo: #fooBar.
        => false
```

OrderedCollectionのインスタンスを生成するまでもなく、OrderedCollectionのクラスに対しても、同様の問い合わせができます。

```
OrderedCollection canUnderstand: #add:
        => true
```

実は、respondsTo: は、canUnderstand: を使って実現しています。
以下を"do it"して確認してみてください。

```
Smalltalk browseAllImplementorsOf: #respondsTo:
```



respondtTo: の実装

respondsTo:はルートのクラスであるObjectにおいて定義されており、内部で自分のクラスを得て、そのクラスに対して、canUnderstandを送っています。これも、インタンスからクラスをイントロスペクションで得ることのできる機能をうまく利用した例といえます。

特定クラスの提供する操作のリストを得る

今度は、クラスが実装する操作の集合を得ることにします。

先ほどはOrderedCollectionインスタンスの理解できるメッセージを、あてずっぽうで調べましたが、以下のように、OrderedCollectionクラスに対し、クラスの提供する操作を問い合わせれば、そのような地道な努力をせずにすみます。

```
"OrderedCollction自身で提供された操作の集合を得る"
OrderedCollection selectors
"スーパークラスも含め、OrderedCollectionが提供できる全ての操作を得る"
OrderedCollection allSelectors
```

結果は省略します。OrderedCollection以外のクラスでもいろいろ試してみてください。

Prev. Index Next

オブジェクトの広場

［Happy Squeaking!!］

3. Squeak演習:イントロスペクション

3. 3 オブジェクトの状態を調べる

オブジェクトの属性にアクセスする

今度は生成されたインスタンスの状態を調べてみることにします。
インスタンスの属性の値を見ることができれば、インスタンスがどのような状態であるかがわかります。

以下を"print it"してみましょう。

```
OrderedCollection instVarNames
        =>  ('array' 'firstIndex' 'lastIndex' )
```

OrderedCollectionには、'array'、'firstIndex'、'lastIndex'の三つの変数が属性として順番に定義されていることがわかります。

これを踏まえた上で、次のコードの結果を見てみます。

```
| ord |
ord := OrderedCollection new.
ord add: 'a'; add:'b'; add:'c'.
ord instVarAt: 1.
        =>  (nil nil 'a' 'b' 'c' nil nil nil nil nil )
```

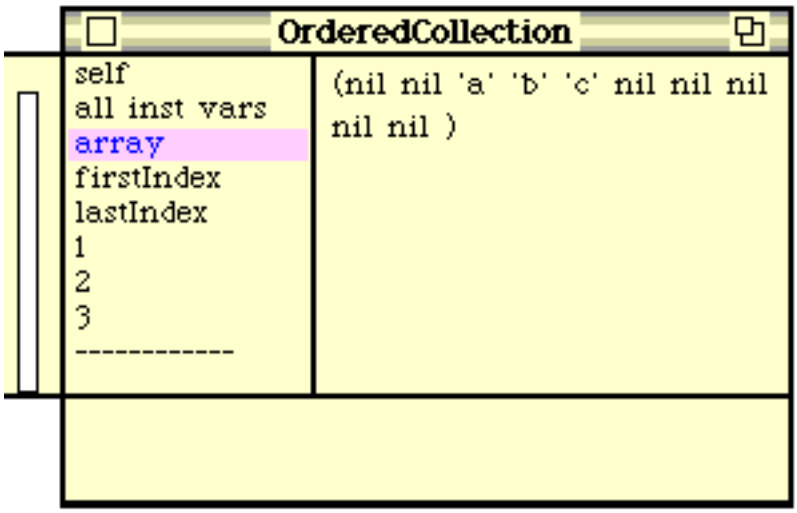
何が返ってきたか想像できるでしょうか。instVar:1によってインスタンスの持つ変数の一番目の値を指定しています。つまりOrderedCollectionインスタンスが'array'変数の中に保持している属性値が取り出されたのです。

Squeakの場合は、名前指定によるインスタンス変数へのアクセスもできます。ord instVarAt:1を以下のように書き換えて実行してみてください。

```
ord instVarNamed: 'firstIndex'. "変数の名前を指定して値を取り出す"
        => 3
```

これらのメッセージはおなじみのインスペクタで使われています。インスペクタがあらゆるオブジェクトの内部を覗けるのは、実はここで挙げたイントロスペクト用のメッセージを使っていたからなのです。

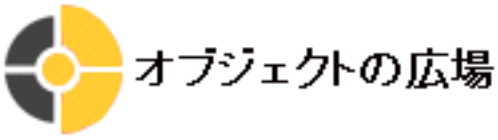
インスペクタを開くと同様の結果が確認できます。



OrderedCollectionのインスペクト

これは一種の「カプセル化」破りです。本来、インスタンス内部に保持されている変数は、変数アクセス用のメソッドをそのクラスに定義しない限り、アクセスすることは許されません。しかし、instVarAt: を使えば、アクセス用のメソッドがなくても値が取り出せ、かつ(後でやりますが)変更も可能です。これらのメッセージは、非常に強力ですが、使い方を誤ると、オブジェクト指向をくずすことになるので注意が必要です。

Prev. Index Next



［Happy Squeaking!!］

4. 自分を変えるオブジェクト

4. 1 「世界観」把握のためのメタモデリング

ソフトウェアを「現実世界の問題をコンピュータによって解決するもの」とすると、複雑な現実世界を、何らかの形で、より単純化されたものとしてモデリングする必要が出てきます。

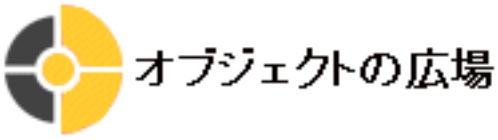
例えば、私達は、今まで現実世界を見る方法としてオブジェクト指向を選択してきました。オブジェクト指向の立場に立つことで、私達は、現実世界をオブジェクト指向的にモデリングしていくことができます。つまり現実世界は「クラス」や「インスタンス」といったモデル要素から成り立つと考え、その考えを元に「犬」や「ポチ」といった実際のモデル要素を見出していきます。

ところが、ここで立場を変えて「データ中心」の考えに立つとしましょう。
データ中心では世界を構成するものは「エンティティ」と「リレーション」です。したがって現実世界はこれらのモデル要素の集まりから成り立つようになります。「犬」はクラスでなくエンティティとして認識され、結果として「ワンとなく」といった犬の振る舞いは意味のないものとみなさます。

このように、私達が何らかの現実世界をモデリングするには、何がおおもとのモデル要素となるかを規定しているものが必要です。モデル要素の背後にあり、モデル要素が何かを定めている、いわば「モデルのモデル」(メタモデル)です。この「モデルのモデル」が変われば、私達が世界を把握する単位もまるで変わってしまいます。世界に関するもののとらえかた「世界観」が変われば、見える世界も違うものになるということです。
こうした「背後で意味を与えてくれているもの」が不安定だと、様々な世界構成要素の存在が危ぶまれてしまいます。例えば「資本主義」と「社会主義」は、経済世界に対する異なる世界観をそれぞれ表しています。資本主義の世界では、A社は株式会社として成り立っていましたが、社会主義世界に移った場合には、そのままでは操業していくことが困難です。また江戸時代の「封建制」で確かに存在した武士は、現在の平成の「民主制」の世では、自称武士ということはできても、社会の枠組みにはまった存在としてはありえないものです。なぜなら、個々の世界構成要素が存在するためのそもそもの枠組みが違っているからです。

「メタモデリング」とは、モデリングを行なう際にこうした混乱が起こらないように「そもそも何がモデル要素となるのか」をしっかりと定めることです。

単純な問題を扱うだけであれば、「メタモデル」はもはや「自明の利」であり、あえて着目されることもありませんでした。しかし、現在では、ソフトウェアが取り扱う問題は、非常に複雑化し、「メタモデル」なしでの現実の適切な分析(モデリング)は、難しいものになりつつあります。



［Happy Squeaking!!］

4. 自分を変えるオブジェクト

4. 2 インターセッション

「メタモデリング」によって作られた「メタモデル」は、世界をモデリングするための堅牢な基盤を与えます。さらに、メタモデルを自由に変更可能にすることで、世界を様々に構成し直すことが可能になります。

江戸時代から明治時代への「文明開化」による日本社会の組み替えは、私達がコントロールできるものではありませんが、ことソフトウェアに関しては、意図的にメタモデルを変更可能にすることによって、後の柔軟性、拡張性を持たせることができます。

ここで、最近はやりのXMLについてみてみましょう。XMLは、"eXtensible Markup Language"の略です。いったい何がExtensible(拡張可能)なのでしょうか。HTMLではタグの要素は「HTML3.2仕様」といった形で定められ、自由な変更は不可能でした。(ブラウザメーカによる勝手な拡張はあったにせよ)。一方でXMLは、何が「タグの要素となるのか」というメタモデルを自ら定義できる機能を持っています。これによりHTMLでは実現し得なかった、任意の意味をもった構成要素を文章に持たせることが可能になりました。

XMLでは、タグの意味を定めるためにDTD(Document Type Definition)という言語が使われており、これがメタモデルを提供する役割を果たしています。例えば、本の情報を格納するページのフォーマットを規定する際に、DTDで以下のように定義することができます。

```
<!DOCTYPE bookInfo [
<! ELEMENT bookInfo(title, author, isbn, commentList) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT isbn (#PCDATA) >
<!ELEMENT commentList (comment+) >
<!ELEMENT comment (#PCDATA) >
] >
```

DTDは、本情報の文書には、「タイトル、著者、ISBN、(コメント(複数))」が要素としてあることが定められています。このDTDにもとづき、XMLの実際の文書は、以下のようになります。

```
<? xml version=1.0" ?>
<bookInfo>
<title> Smalltalk Best Practice Patterns</title>
<author>Kent Beck</author>
```

<isbn>0-13-476904-X</isbn>
<commentList>
<comment>筆者のコンサルティング経験の中で生まれたSmalltalkのデザインパターンを平易に紹介。</comment>
</bookInfo>

HTMLはいわば世界が固定された世界です。HTML3.2ならばHTML3.2の世界で、それに準拠したタグ内に実際のデータが埋め込まれていくのみです。HTML3.2を守らず勝手なタグを付け加えても、Webブラウザ側ではそれが何を意味するかがわからない(メタモデル、およびそれを表すメタ情報がない)ので、適切な表示などの処理を行なうことができません。XMLであれば、文書を構成する要素をDTDという形式で与えることによって、文書に新たな意味を持った要素を自由に追加でき、後の検索、表示の手がかりにしていくことができるのです。

また、オブジェクト指向分析設計の際に使われるモデリング表記法として、皆さんおなじみのUMLがありますが、やはりモデルの要素を定めるメタモデルがきちんと定められています。(具体的には「クラス」や「関連」といったものがメタモデルになります)。また、新たなモデル要素の追加に対し、ステレオタイプという仕組みによって、対処することができるようになってもいます。その例として、ObjectTime社のUMLのリアルタイム拡張 (ROOM)があります。capsule、port、connectorという三つの新たなモデル要素が、ステレオタイプを使うことでUMLのメタモデルに追加され、結果としてリアルタイム分野でのモデリング記述能力を増大させています。本来リアルタイムシステム用でないUMLも、ステレオタイプという、メタモデル追加の仕組みを用意しておいたことにより、リアルタイム用に容易にカスタマイズ、発展されていくことができたのです。これもメタモデルの組み替えの有効性を示しているといえるでしょう。

このように、システムに対し、単にメタ情報を尋ねてその値を取り出すだけでなく、得られたメタ情報を利用して自身の値を書き換えたり、メタ情報を構成するメタモデルを変更していくことが可能であれば、柔軟性、拡張性において非常に強力になります。

こうした機能は、「イントロスペクション」に対して「インターセッション」と呼ばれます。インターセッションは、「調停、とりなし」といった意味です。オブジェクト指向の世界では、通常のオブジェクトとメタ情報を表現するオブジェクトが互いに関わり合い、自らの環境を変化させていけることを指しています。

「イントロスペクション」に加え、「インターセッション」がサポートされることで、オブジェクトは、真に自律的な、よりエージェント化されたものへ進化していくことができるようになります。「インターセッション」と「イントロスペクション」はまとめて「リフレクション」と呼ばれます。リフレクション機能により、システムが自らの状態を察知し、それに合せて変容していくというエージェントシステムは、まだ実験的と思われていますが、次の潮流であることは間違い有りません。

「リフレクション」がフルサポートされたオブジェクト指向言語は、まださほどないのが現状です。が、Squeakはかなりいい線までいっているといえます。以下、例をみてみることにしましょう。

オブジェクトの広場

［Happy Squeaking!!］

5. Squeak演習: インターセッション

5. 1 インターセッション初歩

インターセッションのもっとも簡単な例として、メタ情報を利用してインスタンスの状態や振る舞いを変更させることをしてみましょう。

オブジェクトの状態を変更する

今回は、以前の例にもでてきた、銀行口座クラス(Account) からインスタンスを作り、インターセッションの機能を使い属性の値を変更してみることにします。

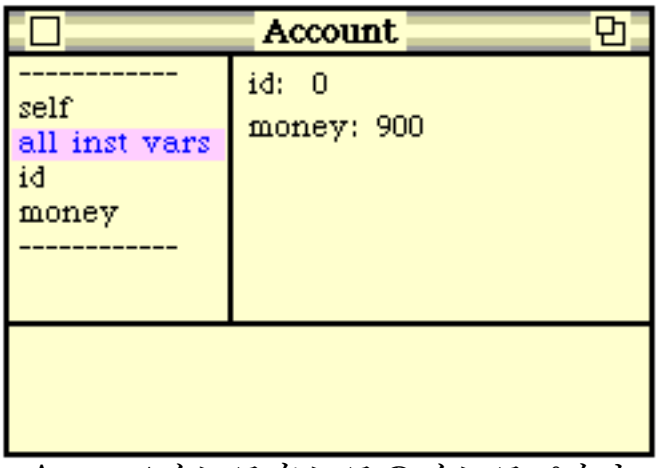
以下のリンクからソースのダウンロードをしてください。
BankApplication.st (<= CLICK NOW)

ファイルリストを使いSqueak内にロードします。やり方は第三回の「サンプルコードの読み込み」を参照してください。

まずは銀行口座のインスタンスを生成し、行儀よく値を設定します。

```
| acc |
acc := Account new.
acc initialize.
acc deposit: 1000.
acc withdraw: 100.
acc inspect
```

以下のようなインスペクタが立ち上がります。ちゃんと値が設定されています。



Accountインスタンスのインスペクト

今、Accountにはidを外から設定するためのメソッドが定義されていません。従っ

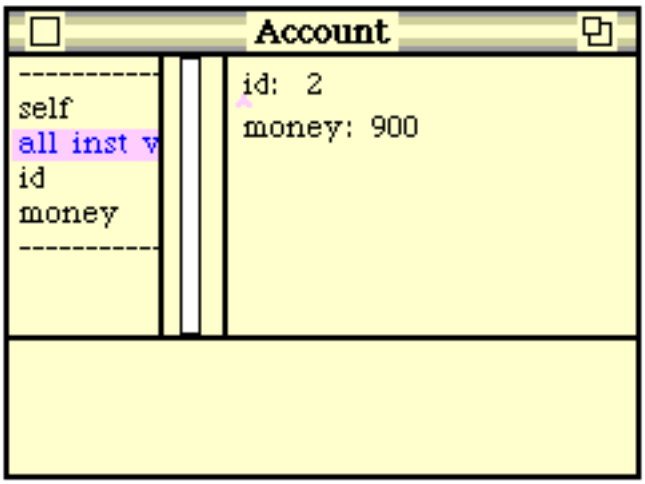
てidの値はinitialize後は常に0になっています。

しかしこのような場合でも、相手がどのような名前の変数を属性として持っているかがわかっていれば書き換えが可能です。インターセッションを利用して値を設定してみましょう。

accのクラス、Accountが、どのような変数を属性として定義しているかは、acc class instVarNames とすることでわかります。ここから'id'という変数がAccountクラスの何番目に定義されているかを知り、次にここで得たインデックスを用いて、最後に変数の値を書き換えるメッセージを送ることとします。

```
| acc  varNames index |
acc := Account new.
acc initialize.
acc deposit: 1000.
acc withdraw: 100.
varNames := acc class instVarNames. "変数のリストを得る"
index := varNames indexOf: 'id'. "'id'変数が何番目に定義されているか"
acc instVarAt: index put: 2. "値の書き換え"
acc inspect.
```

acc inspectの前に、以下を追加してもう一度"do it"してください。今度は見事にidの値が書き換わっています。



インターセッションで値を変更したAccountインスタンスのインスペクト

Squeakの場合は、instVarNamed:put: も使えますのでそれを使えば上記のコードはもっと短くてすみます。(余裕のある方は、instVarNamed:put:の実装を見てみてください)。

indexOf:についてははじめて出てきました。集合オブジェクトに投げることが出き、特定のオブジェクトが集合の何番目に含まれているかを調べるメッセージです。

SmallTip: 集合のインタンスに含まれる特定要素のインデックスを知りたい場合には、indexOf:を用いる

> instVarAt: put: を使えば、オブジェクトが内部に保持しているはずの変数を自由に書き換え可能なので、カプセル化が壊れることになります。Account側でカプセル化をどうしても守りたい場合は、instVarAt:put自体をオーバーライドして、アクセス不要にしてしまう方法があります。

オブジェクトの操作を間接的に起動する

次は、属性のアクセスではなく、操作の起動を行うことにします。

Accountクラスで定義された操作名の集合を得て、そこから使えそうな操作を選択。最後に実際の操作の起動という流れです。
以下のようになります。

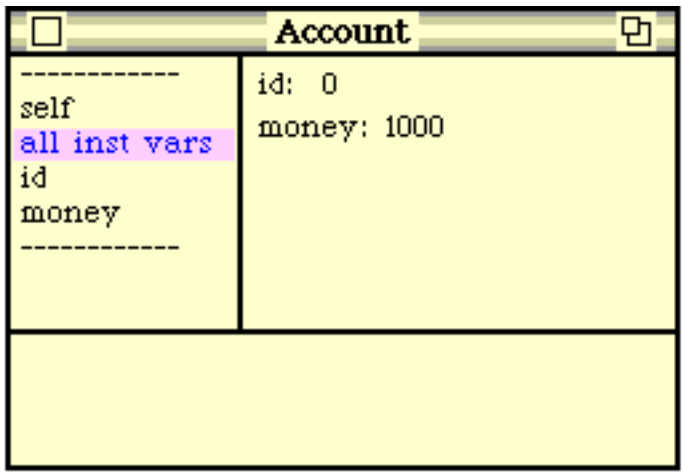
```
| acc opNames maybeOp |
acc := Account new.
acc initialize.
opNames := acc class selectors. "操作名の集合を得る"
maybeOp := #deposit:. "多分あるだろうという操作"
(opNames includes: maybeOp) "集合に含まれていた場合に"
        ifTrue:[ acc perform: maybeOp with: 1000]. "間接的に実際の起動"
acc inspect.
```

操作の集合は Set (initialize deposit: withdraw: getBalance)となります。これに、有るだろうという操作のセレクタ(#depposit:)が含まれるかをincludes:で確かめます。

> SmallTip: 集合のインスタンスに特定要素が含まれるかを確かめる場合にはincludes:を用いる

含まれていた場合には実際にメッセージを送ります。ここで、直にdeposit:1000と書いていないところが今回のポイントです。このようにすることで、maybeOp変数に代入されるセレクタの名前を変更するだけで、いろいろな場合に対処できる可能性がでてきます。(この例では、ほとんど影響はないですが、より本格的なものを作るときには有効です)。

結果は以下の通り、ちゃんとdeposit:が起動されていることが確認できます。



perform: によりdeposit操作がおこなわれたAccountのインスタンス

Smalltalkでは perform: というメッセージにより、間接的なメッセージの起動を行なうことができます。
基本的な記法は
perform: (送りたいメッセージ名) with: (送りたいメッセージの引数)
となります。

送りたいメッセージの引数に応じて、幾つかのバリエーションが存在します。

```
perform: (引数なし)
perform:with: (引数一つ)
perform:with:with: (引数二つ)
perform: withArguments: (引数を配列にして設定)
```

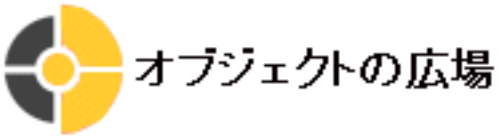
acc initialize と書かずに、acc perform: #initialize と書けるということです。
メッセージ名(セレクタ)は、単なるシンボル(書き換え不能な文字列)であり、変数を使うなどして自由に設定できます。上記の場合は、maybeOpに #deposit:を代入して利用しています。

もう少し例を示しましょう。

```
String new perform: #size.
Transcript perform: #show: with: 'Hello'.
'HelloSmalltalk' perform: #copyFrom:to: withArguments: #(1 5).
```

SmallTip: perform:を利用することで、メッセージ名による間接的なメッセージの起動ができる

Prev. Index Next



［Happy Squeaking!!］

5. Squeak演習: インターセッション

5. 2 オブジェクトの属する世界の変更

今度は、メタ情報を利用してのオブジェクトの値の単なる書き換えでなく、オブジェクトの属する世界(メタモデル)自体を変貌させることを試みます。

そのためにはSmalltalkのメタモデルの理解が不可欠になります。いままであえて避けてきた、「メタクラス」の説明に、いよいよ入っていきたいと思います。
初心者のかたには多少アドバンストな内容ですが、ここを超えれば、Smalltalkの核心に触れ、また、オブジェクト指向のより深い部分を垣間見たということになります。

> Smalltalkにおけるオブジェクト指向の世界のメタモデルがこうなっているということであり、他のやり方でのオブジェクト指向のメタモデリングもあるということには留意しておいてください。(例: UMLのメタモデルなど)

Prev. Index Next

オブジェクトの広場

［Happy Squeaking!!］

5. Squeak演習: インターセッション

5. 3 メタクラス--クラスを支えるもの

クラスのクラス

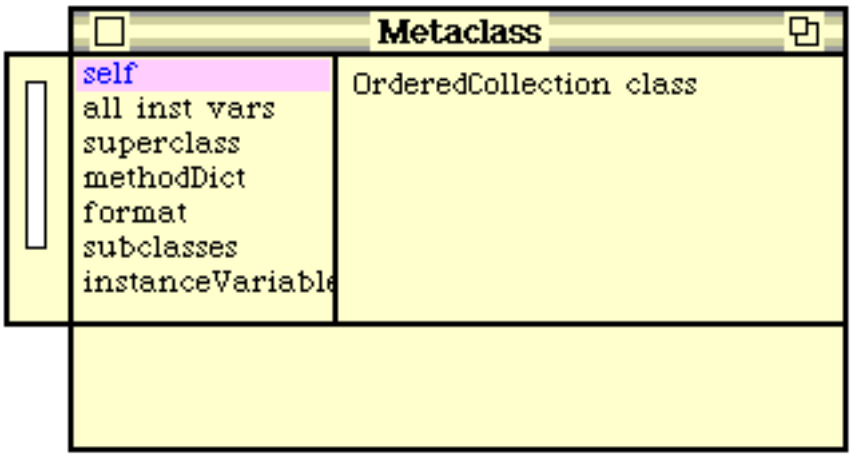
今までの演習で、インスタンスだけでなくクラスに対しても、様々なメッセージを送ってきました。インスタンスのメッセージに対する受け答えの方法は、クラスでメソッドとして定義されています。では、クラスに対するメッセージはどこで定義されているのでしょうか。それは、クラスを支えるもの、「メタクラス」においてに他なりません。

メタクラスは、クラスのクラスです。インスタンスからクラスを取り出すときにclassメッセージが使えたように、クラスに対してclassメッセージを送ることでメタクラス(ある特定クラスについてのメタクラスのインスタンス)を取り出すことができます。

以下を "do it" してみましょう。

```
OrderedCollection class inspect.
```

次のようなインスペクタが立ち上がります。



OrderedCollectionのメタクラスのインスペクト

メタクラスには、名前がありません。Smalltalkでは、便宜上「なになにクラスのメタクラス」ということを表すために、「クラス名 class」という表記を用いています。結果として、OrderedCollection class の結果はOrderedCollection classとなり、見かけ上は、何も起こっていないかのように見えますが、きちんとメタクラスは取得できています。(前者はメッセージ、後者はただのclassという表示です)。

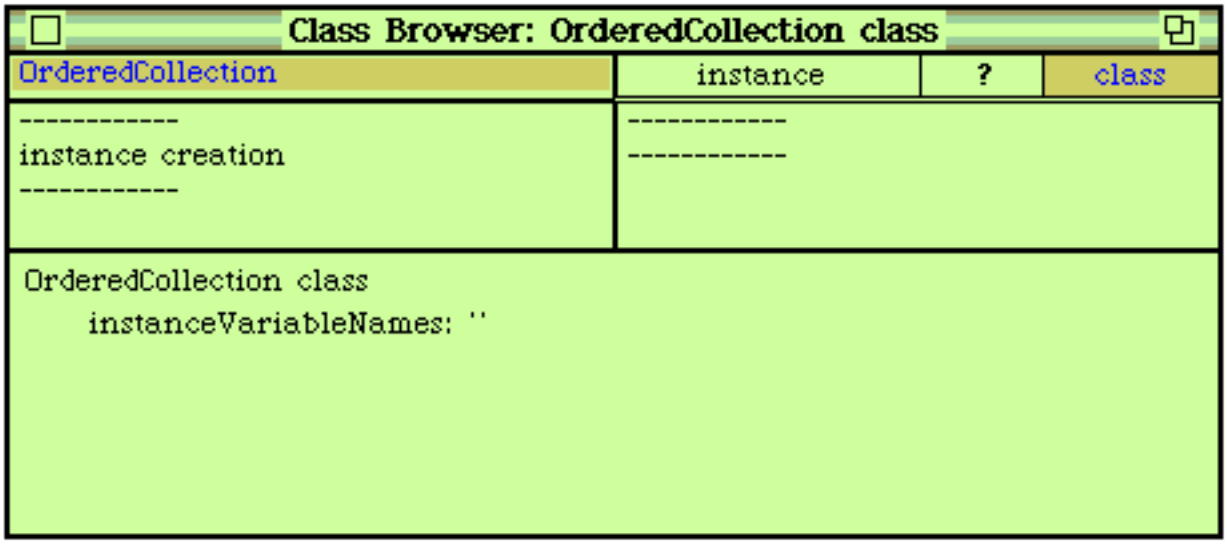
実は、もっと簡単にメタクラスを見る方法があります。

以下を、"do it"してみてください。

```
Browser newOnClass: OrderedCollection class
```

Browser newOnClass: のあとは今まではクラスを引数として渡していましたが、今回は、メタクラスを引数として渡しています。

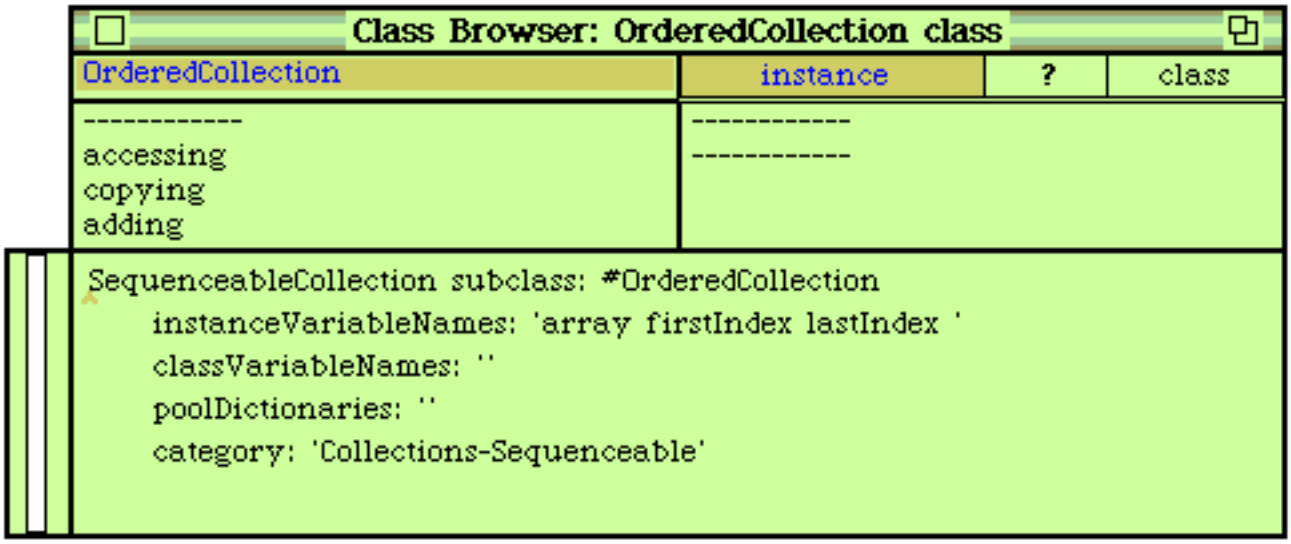
以下のようなクラスブラウザが立ち上がります。



OrderedCollection class (OrderedCollectionメタクラス)のブラウズ

ブラウザの右上の部分が"class"になっていることに注目してください。今までは"instance"サイドがハイライトされていたはずです。

ブラウザで、instanceボタンをクリックするとおなじみのOrderedCollectionクラスのブラウズに切り替わります。



OrderedCollectionのブラウズ

更につづく

オブジェクトの広場

[Happy Squeaking!!]

5. Squeak演習: インターセッション

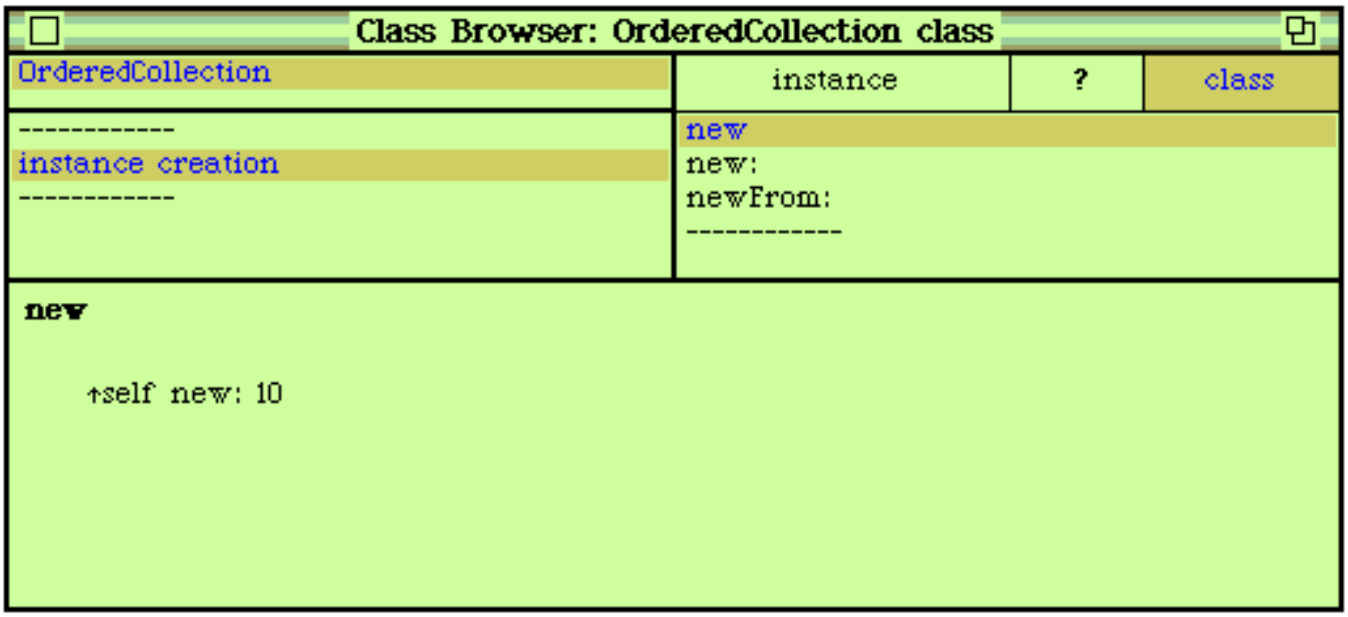
5. 3 メタクラス・クラスを支えるもの つづき

クラス自身の振る舞い-クラスメソッドの定義

ブラウザでは、クラスに操作の定義を付け加えることで、インスタンスの振る舞いを付け加えることができました。同様にメタクラスに操作の定義をつけ加えることで、クラス自身の振る舞い(クラスメソッド)を決めていくことができます。
(ブラウザの"instance"、"class"の表示は、一見紛らわしいですが、インスタンスとクラスの振る舞いの仕方と考えるとすっきりします)。

クラスの振る舞いで典型的なのは、インスタンスの生成の仕方を決めることです。今までは、あまり意識することもなく、クラスに対し、newメッセージを送ってきましたが、この振る舞いをメタクラスに定義することで変更ができるのです。

ブラウザによれば、OrderedCollectionの場合は、既に3つの生成用の操作が定義されていることが確認できます。



OrderedCollectionメタクラスに定義されたインスタンス生成操作を見る

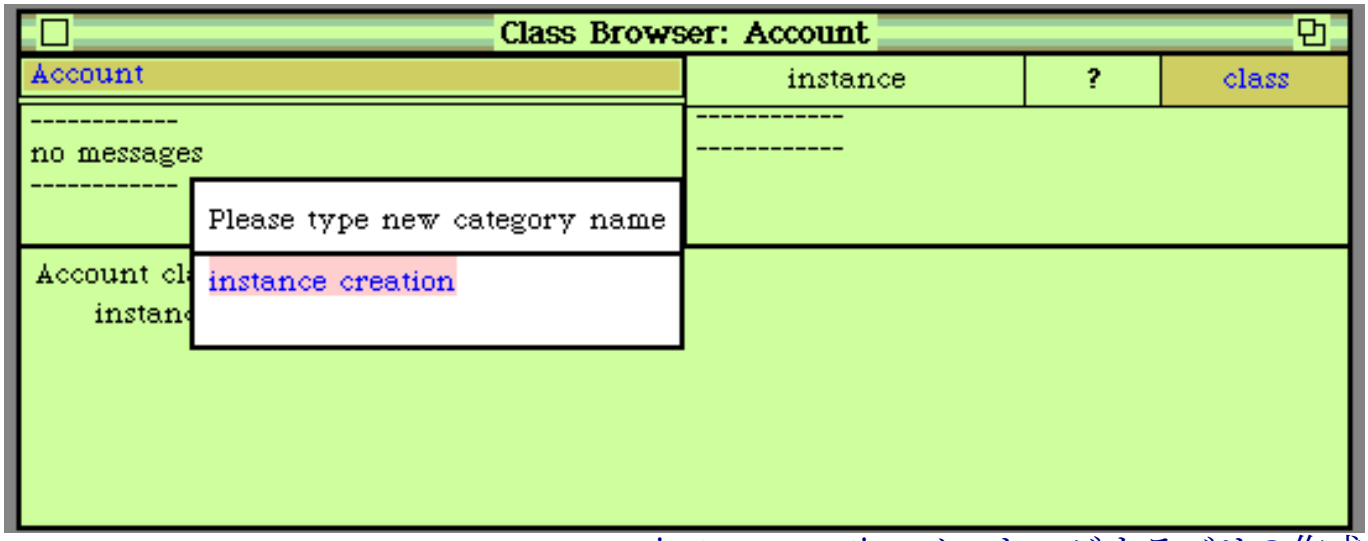
ここでは、OrderedCollectionのインスタンス生成操作の詳しい解説はしません。new: は初期サイズを指定したインスタンス生成。newFrom:は初期データを指定したインスタンス生成を提供しています。

では、演習としてAccountクラスが、Accountインスタンスを自らのやり方で生成できるようにしていきましょう。

Accountクラスのブラウザを開き、"class"側にブラウザを合せます。インスタンス生成はク

ラスが行なうことです。クラスの振る舞いの仕方(メソッド)はメタクラスに定義されるのです。

メッセージカテゴリ、"instance creation"を追加します。



instance creation メッセージカテゴリの作成

instance creationカテゴリ下に、new メソッドを以下のように定義します。

```
new
        ^super new initialize.
```

super newで上位で定義されたnewを利用し、インスタンスを生成しています。次に生成されたインスタンスに対してinitializeメッセージを送り、インスタンスを初期化して、返り値としています。

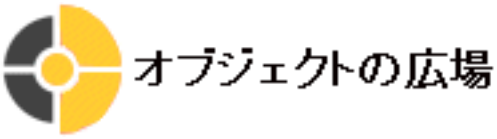
このようにインスタンス生成の操作をクラスに与えてあげることによって、Accountインスタンスの利用者は、インスタンス生成のあとでいちいちinitializeを送ってあげる必要がなくなります。

```
| acc opNames |
acc := Account new. "Accountインスタンス生成"
"acc initialize." "<<==new内部で送られるので不要"
acc deposit: 1000.
```

従って上記のように、initializeを送る部分をコメントアウトしても大丈夫になったわけです。

更につづく

Prev. Index Next



[Happy Squeaking!!]

5. Squeak演習: インターセッション

5. 3 メタクラス・クラスを支えるもの つづき

クラス自身の属性 - クラスインスタンス変数

インスタンスが属性の値を保持できるように、クラス自身も値を保持できます。クラスが持つべき属性の定義は、操作と同様やはりメタクラスで行なうことができます。
インスタンス生成の操作をクラスが担当する一方で、クラスの属性は、生成するインスタンスを管理するためによく使われます。例えば、作られたインスタンスをすべてプールしておく集合オブジェクトを持ったり、インスタンス生成の際、インスタンス固有のid番号をカウントするなどがその例です。

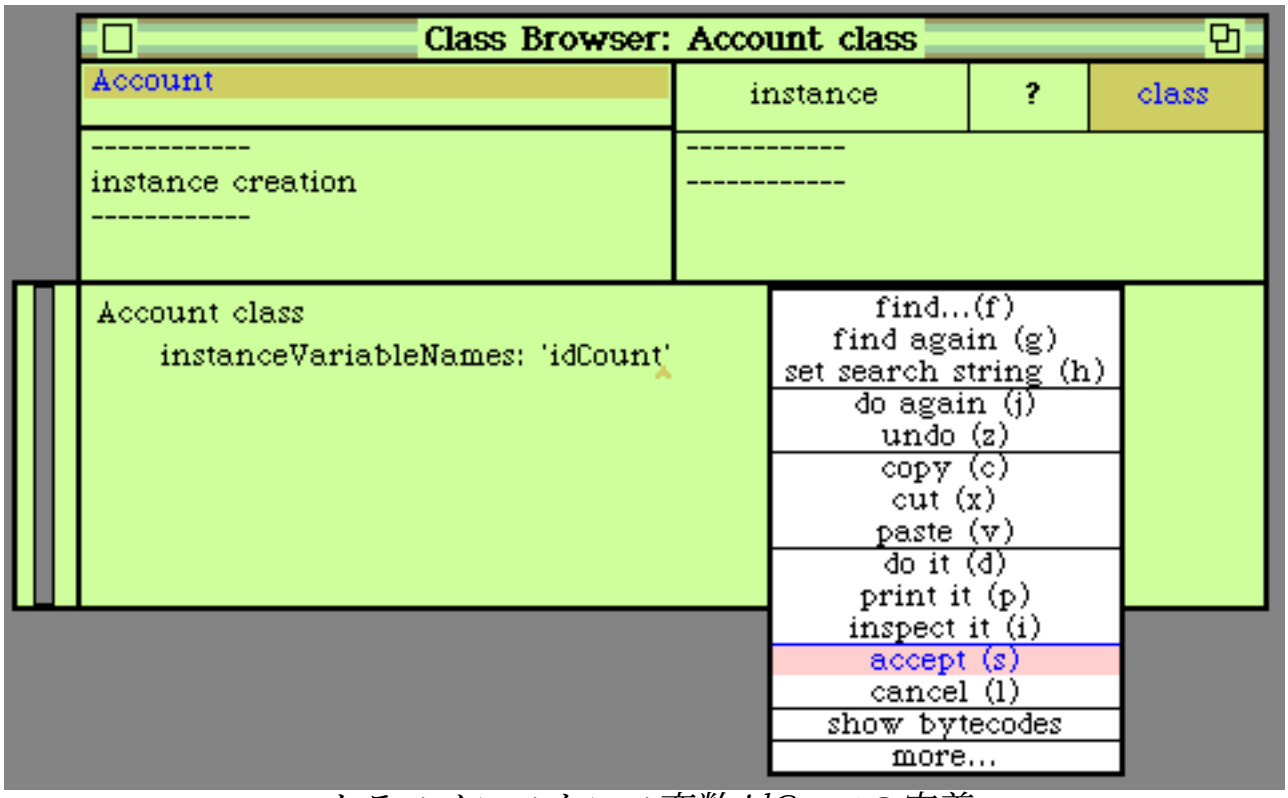
では、Accountメタクラスにdefinitionメッセージを送ってみましょう。以下のような文字列が帰ってきます。

```
Account class definition
        => 'Account class instanceVariableNames: '''''
```

これは普段見慣れているクラス定義用のテンプレート文字列とは少し異なっています。クラスにくらべ、メタクラスのほうが定義できる変数の種類の数が少なくなっているのです。しかし、クラス自身がもつ変数は、このテンプレートを埋めることで定義ができるので、とりあえずはこれで十分です。

ここでは、銀行口座が新規に作られたときに、id番号をカウントして新規インスタンスに与えられるように、idCountという変数を定義することにします。

ブラウザを開き、Account classのdefinitionを表示させます。
"の中に idCountと書いて"accept"してください。



クラスインスタンス変数 idCountの定義

これで、Accountクラスが持つことのできる属性、'idCount'が定義できました。

この「一つ一つのクラスがもつ変数」のことをSmalltalkでは、「クラスインスタンス変数」と呼んでいます。いささかややこしい名前ですが、クラスをあたかもインスタンスのように、値を保持するものとして扱っているからと考えるといいでしょう。

> Smalltalkにはまた別に「クラス変数」というものもあります。これは一種の共有変数で、「クラス」と「インスタンス(群)」が、一緒に参照、書き込みができるという性質を持っています。JavaやC++のstatic(静的)変数に、比較的近いと言えます。今回は扱いません。

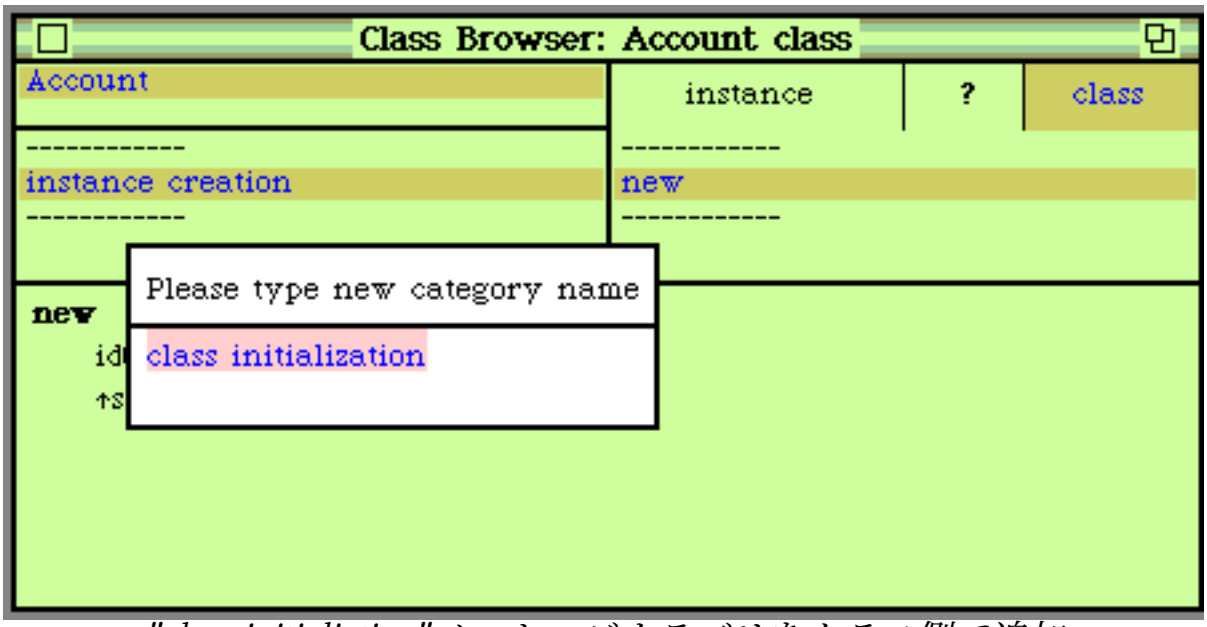
クラスインスタンス変数は、クラスが自由に扱うことのできる変数です。クラスメソッドの中でのみ利用でき、インスタンスからは触ることができないようになています。

idCountを先ほど定義したクラスメソッドnewの中で使うことにします。
以下のような実装になります。

```
new
        idCount := idCount + 1. "クラスインスタンス変数 idCountを増加"
        ^super new initiallize;
        id: idCount
        "増加したidCountを生成したインスタンスに渡す"
```

ただし、idCountはまだ初期化されていないので、これだけではエラーになります。これを防ぐため、インスタンスメソッドにinitializeを定義したのと同様、クラスメソッドにもinitializeを定義することになります。
(かたや「インスタンス」の初期化、かたや「クラス」の初期化です。いうまでもないですが、両者は全く別々のオブジェクトです。混乱しないようにしてください)。

メッセージカテゴリ"class initialization"を、ブラウザの"class"側で追加します。

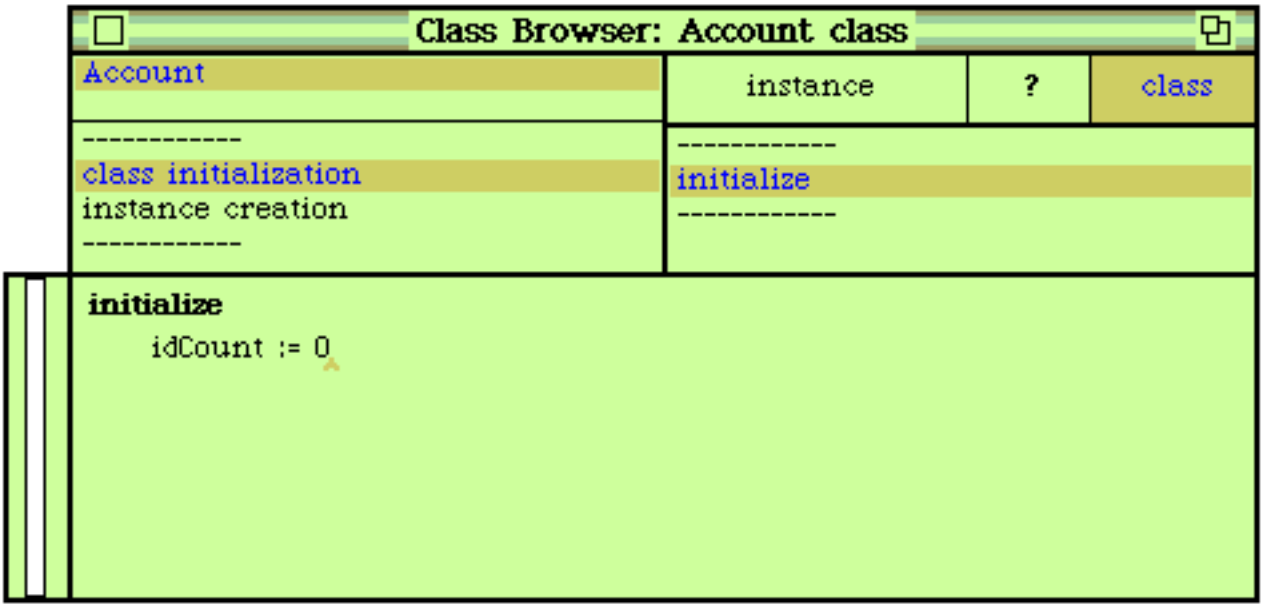


"class initialiation"メッセージカテゴリをクラス側で追加

initializeメソッドの実装は以下のようになります。

```
initialize
        idCount := 0
```

"accept"してください。これで定義されました。



クラスメソッドinitializeの定義

ワークスペースで、Account initializeとdo itしてください。これによりidCountが初期化され、クラスがこの変数を使える準備が整ったことになります。

最後に、新しく定義されたnewメソッドでは、生成したAccountインスタンスに対し、idCountによって振られたidを設定するためのid: メッセージを送っています。これはインスタンスの行なうことなので、メタクラスにではなく、クラスに操作として定義します。ブラウザを"instance"サイドに切り替えて、以下を"accept"してください。

```
Account のid:メソッド (accessingメッセージカテゴリ)
id: newId
        id := newId.
```

以下はここまでの修正を加えたAccountクラスです。

FileIn: BankApplication2.st (<=Click)

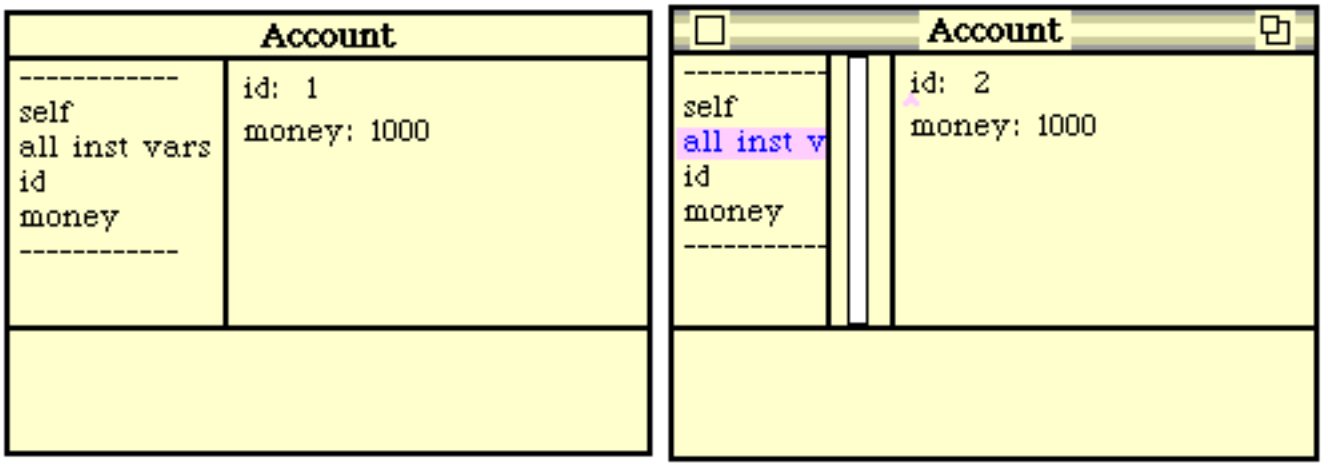
ファイルインにより、Account initializeは自動的に実行されるようになっています。そのためこのソースをそのまま用いる場合は、単にファイルインするだけで動作確認ができます。

SmallTip: ファイルインにより、ファイルインしたクラスのクラスメソッド initializeが自動的に起動される

では以下を何回か実行してみましょう。

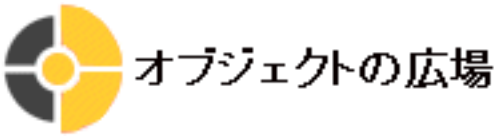
```
| acc |
acc := Account new.
acc deposit: 1000.
acc inspect.
```

一回目、二回目と、インスタンス生成ごとにid番号が振られているのが確認できます。



クラスのnew内の作用によって自動的にidが降られたAccountインスタンス

インスタンスはオンデマンドで作られるオブジェクトであるのに対し、クラスはテンプレート文字列を埋めて定義した瞬間から、Squeak環境の中に居座ることになります。従って両者のオブジェクトの保持する変数のライフサイクルは全く異なり、この例の場合では、AccountクラスのidCountは明示的にワークスペースなどでAccount initializeと実行しない限り、増加した値が保たれることになります。Accountクラスが存在する限り残る、半永続的なものになります。



［Happy Squeaking!!］

5. Squeak演習: インターセッション

5. 4 メタクラスの継承関係

Accountメタクラスでの例

「クラスに属性、操作が持てるということは…」となると次に出てくるのが継承です。実はクラスと同様に、メタクラスにも継承関係はあります。

Accountクラスの継承関係は次のようになっています。

```
"print it"
Account printHierarchy.
'
Object ()

        Account (''id'' ''money'' )'
```

次にAccoutメタクラスの継承関係をみてみましょう。

```
"print it"
Account class printHierarchy.
```

以下のような文字列が得られます。

```
'
Object ()
        Behavior (''superclass'' ''methodDict'' ''format'' ''subclasses'' )
                ClassDescription (''instanceVariables'' ''organization'' )
                        Class (''name'' ''classPool'' ''sharedPools'' )
                                Object class ()

                                                Account class (''idCount'' )'
```

非常に複雑で、驚いてしまいます。Behavior、ClassDescription、Classとは一体何なのでしょうか。

ここで視点を変えてトップダウン的に、概念的に考えてみましょう。生物の誕生、進化過程のように、Smalltalk世界が生まれ、発展していった流れを追うことにします。

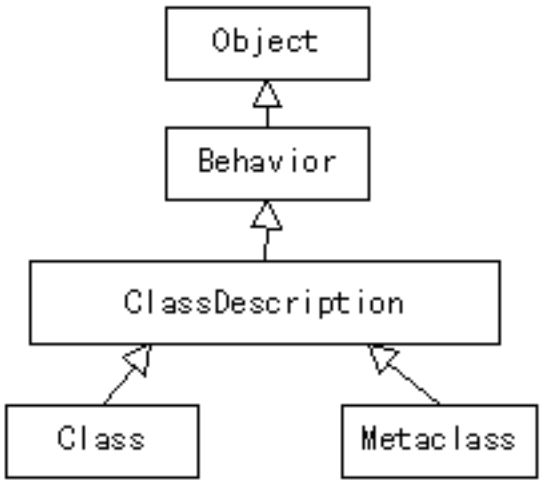
Smalltalkの世界では、Objectという存在が、すべての世界の根元になっているとしています。「最初にObjectありき」です。Objectは、いわば究極の抽象物で、もやもやとしたつかみ所のない、「もの」であったと考えてみてください。

> ちなみに，通常のオブジェクトと混同しないようにこの究極のObjectはSmalltalkコミュニティでは伝統的に「ラージオブジェクト」と呼び習わされています.

この「もの」が発展して、振る舞いを持つように拡張された「もの」がBehavior(「振舞いをもつもの」)です。さらに、属性の値が持てるように拡張したものが、ClassDescription(「クラスのようなもの」)なのです。

「Classのようなもの」は実はSmalltalkでは2種類あります。Class(「インスタンスを分類するもの」)と、Metaclass(「クラスを分類するもの」)です。

ここまでをまとめると以下のような概念図になります。



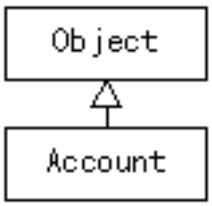
今度はボトムアップ的に考えます。

Account、OrderedCollectionなど、それぞれのクラスは、自分の存在の基盤として、自分用のメタクラス 、(Account class、OrderedCollection class)とつながっています。

メタクラスから見た場合には、Account classは、Accountの設計図を持つ工場として、Accountを生成したと考えることができます。

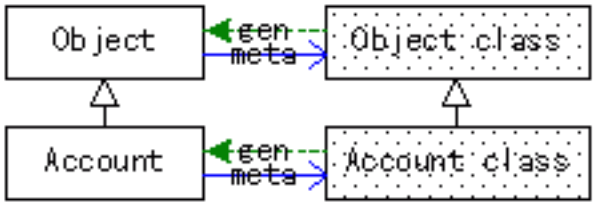
> Accountクラスが2つ以上存在しても、意味がありません(Accountクラスが2つあってもどちらを使っていいか不明)ので、通常Accountメタクラス（Account class）はたった一つのAccountクラスしか生成しません。

クラスの継承関係を辿っていくと、ルートはObjectになります。



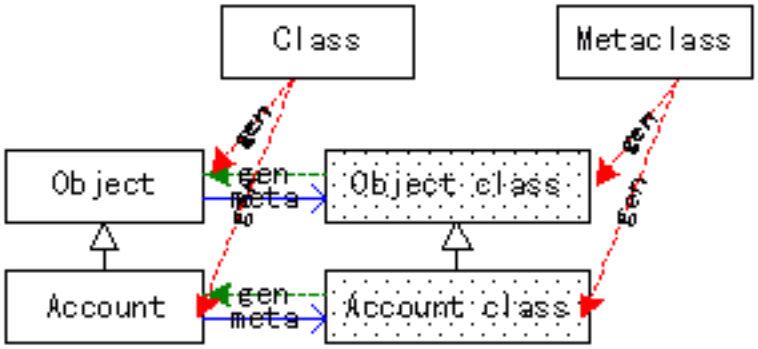
こうしたクラスを影で支えるメタクラス達はやはりパラレルな継承関係を持つこと

になります。

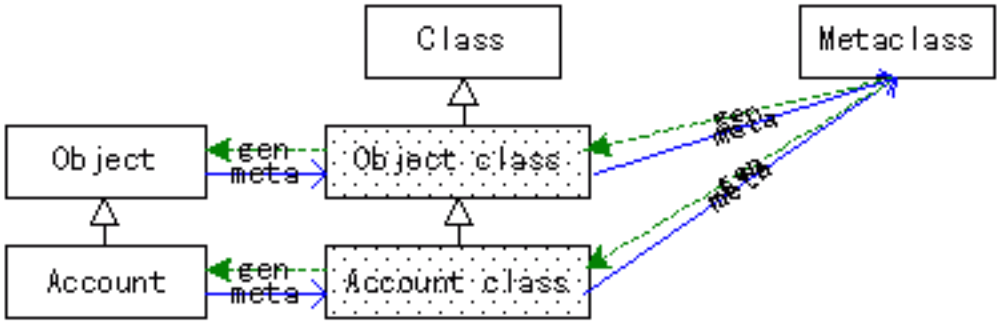


ここで先ほどのトップダウンの図とつなげてみます。

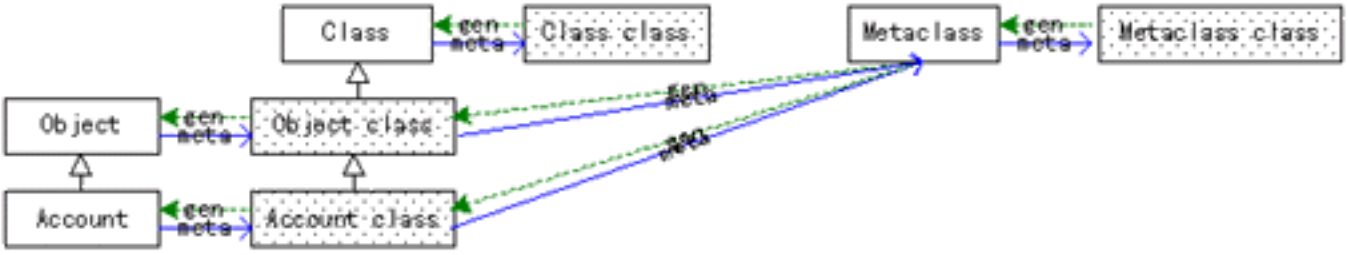
Object、Accountは、全てClassのインスタンス、
Object class、Account classなどは、全てMetaclassのインスタンス、
と、それぞれ考えることができます。



ただし、Classに関しては、Classが直接に、ObjectやAccountを生成しているわけではありません。(図において生成の矢印が重複してしまっています)。実際にはObject classやAcocunt classが、それらの役目を担っています。ということは、Classは実は個々の ～ classを取りまとめる抽象クラスだったことがわかります。

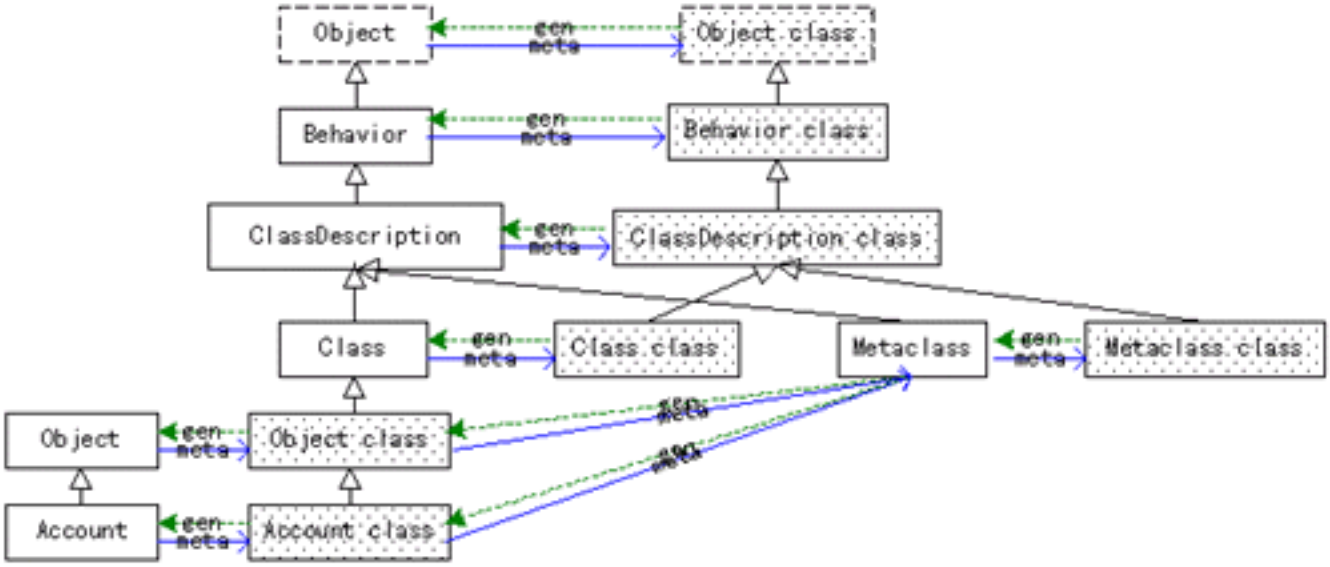


トップダウン的に考えてきたClassやMetaclassもやはりそれを支えるメタクラスがあります。



拡大図

先ほどのBehaviorやClassDescriptionも同様です。



拡大図

ようやくトップダウンの階層とボトムアップの階層とが完全につながりました。（図では一部、生成、メタの矢印が省略されています）。
Objectが二箇所で出てきますが、実際には同一のものになり、全てがObjectからはじまる世界が構築されたことになります。

これで先ほどのメタクラス階層の表示が納得できるようになったかと思います。
Object、Behavior、ClassDescription、Class、Object class、Account classと続くことになります。

Classのサブクラスに、メタクラスのインスタンス(Object class、Account classなど)が来ることに疑問を思われるかもしれません。しかし、メタクラスにクラスの属性や操作が定義できるということは、メタクラス間でも適切な継承関係が存在しなければならないことになります。newのオーバーライドができるのは、この～ classの継承関係のためなのです。メタクラスのインスタンス達は、それぞれのクラス定義(Accountなど)用にカスタマイズされたClassの一種ということになります。

> Metaclassのインスタンス(～ class)が、同時にClassのサブクラスになるという関係は実はそれほど珍しいものでもありません。このように、特定クラスAのインスタンスが、また別のクラスPのサブクラスになるという関係がある場合、そのクラスPはクラスAのパワータイプ（べき型）であるといわれます。オブジェクト指向の分析で重要な概念ですが、今回は範囲外です。

更につづく

オブジェクトの広場

[Happy Squeaking!!]

5. Squeak演習: インターセッション

5. 4 メタクラスの継承関係 つづき

メタのメタ、さらにメタ

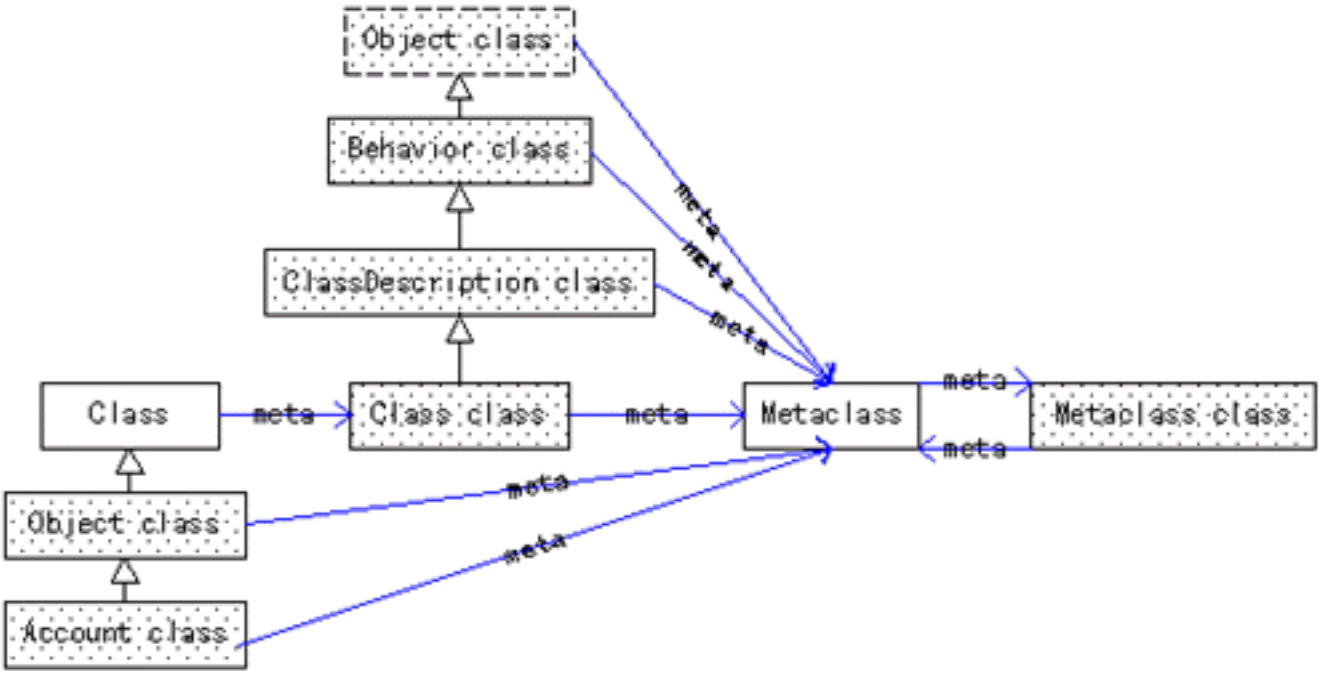
メタの概念は相対的なものであり、どこを視点とするかで、無限に続いていくという話しを最初にしました。Smalltalkでは、これが、Metaclassのメタ関係の循環となって表れています。

先ほどの図でMetaclassに注目した場合、

```
Metaclass =meta => Metaclass class
        (Account =meta=> Account class と同じ)
Metaclass =generate=> Metaclass class
        (Metaclass =generate=> Account class と同じ)
```

という関係が成り立っているのです。

メタ関係を、Metaclass classを中心に書くと以下のようになっています。



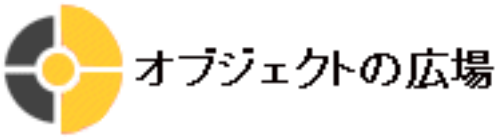
拡大図

標語風にいえば、
「MetaclassのclassはMetaclass class。Metaclass classのclassはMetaclass」
なのです。

以上がSmalltalkのメタモデルになります。難しいと思いますが、メタモデルを考えることは抽象的な思考の訓練にもなり、オブジェクト指向分析全般にも生きてきます。よくわからなかった場合は折りを見て振り返るようにしましょう。

では、演習ではこのメタモデルに手を加えていくことにします。


Prev. Index Next



［Happy Squeaking!!］

5. Squeak演習: インターセッション

5. 5 インターセッション応用

SmalltalkにInterface概念を追加

Smalltalkの主要なメタモデル要素は、ClassDescription下に定義された、ClassとMetaclassであるということがわかりました。

ここで、新たにInterfaceというものを言語内で使いたくなったとします。(Javaで言うあのInterfaceです)。通常では、言語自体の再設計ということになります。到底一般のユーザが対処できるレベルではありません。
しかしSmalltalkにはメタモデルがあります。メタモデル要素にInterfaceを付け加えることで、自らの存在する世界を変え、進化していくことができるのです。

Interfaceの分析

ではまず、Interfaceがどのようなものかを考えてみましょう。モデリングには分析が不可欠です。メタモデリングにおいてもこれは同様です。

Interfaceは、操作名の集合を表したものです。属性、及び操作の実装はInterfaceには含まれない点が特徴です。

Interfaceはクラスにより実装(implements)されます。例えば、IAccountというインターフェースが#(deposit: getBalance withdraw:)という操作名を集合として持つ場合には、IAccountをimplementsするクラスは、それらの全ての操作が理解できねばなりません。つまり自身かスーパークラスでメソッドとして実装していることになります。

クラスは複数のInterfaceをimplementsすることができます。 例えば、IPrintableが#(printString) 、IAccountが#(deposit: getBalance withdraw)を持つ場合には、それらの両方をクラスはimplementsできます。その場合クラスは、#(printString deposit: getBalance withdraw)の全てのメッセージに答えられねばなりません。

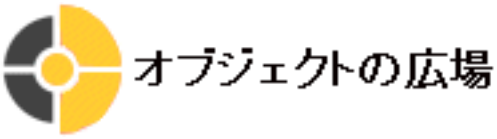
Interfaceは、他のInterfaceを拡張(extends)することができます。IReadableが#(read)、IWritableが#(write)を持つ場合に、両者をextendsするインターフェースとして#(read write)を持つIReadWriteStorableを定義できます。

操作の集合がクラスとは別のInterfaceとして定義されることで、特定のインスタンスが、特定のメッセージに答える保証をすることが可能になります。インスタンスinstのクラスが、IPrintableをimplementsしていた場合、クラスが何であったとしても、メッセージの送り側としてはprintStringを安心して送ることができます。

ざっと以上のような感じでInterfaceがどのようなものかを把握したとします。
これを元にして実際のSmalltalkメタモデルの追加を行ないます。

更につづく

Prev. Index Next



[Happy Squeaking!!]

5. *Squeak演習: インターセッション*

5. 5 インターセッション応用 つづき

Interfaceのメタモデルの追加

まずはInterfaceをSmalltalkの中に定義しなければなりません。これはClassでもMetaclassでもない、全く独立したモデル要素と考えられます。よって、ClassDescriptionの下にサブクラスとして新たに作成します。

Interfaceは名前("IPrintable"など)、操作名の集合(printString、getBalanceなど)をもつので、そのための属性をそれぞれname、operationListとして定義します。

```
ClassDescription subclass: #Interface
        instanceVariableNames: 'name operationList '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Metamodel-Interface'
```

nameは、書き換えられても困るのでシンボル文字列を使うことにします。operaiotnListは、重複する操作名が入っては都合が悪いので、操作名を示すシンボル文字列を要素としてもつSetのインスタンスとします。

これより属性アクセス用のメソッドは以下のようになります。

(Interface >> accessingカテゴリ)

```
name
        name isNil ifTrue:[ name := 'a nameless interface'].
        ^name

name: newName
        name := newName

operationList
        operationList isNil ifTrue:[ operationList := Set new].
        ^operationList

operationList: aListOfOperaionSymbol
        operationList := aListOfOperaionSymbol
```

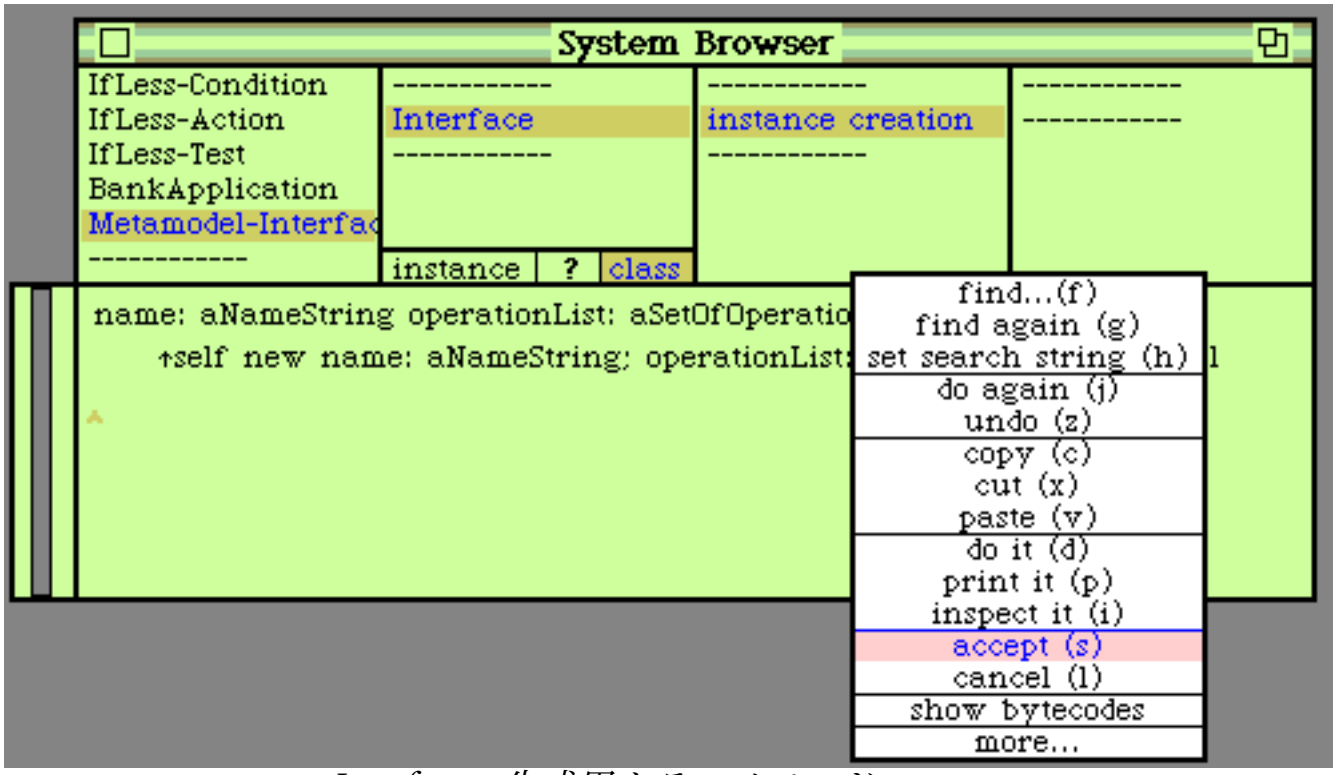
次は、Interfaceを便利に生成できるようなクラスメソッドを定義することにしましょう。

クラスメソッドの名前はnewでなければならないということはありません。Interfaceの場合は、名前と操作リストが必ず必要なのですから、生成時にそれらの情報が引数として与えら

れるようにしたほうが便利でしょう。

そこで以下のようなメソッドをブラウザの"class"側に定義します。
(Interface class >> instance creationカテゴリ)

```
name: aNameString operationList: aSetOfOperationSymbol
        ^self new name: aNameString;
                operationList: aSetOfOperationSymbol asSet
```



Interfaceの生成用クラスメソッドのaccept

これでInterface name: #IPrintable operationList: #(#printString) という形式でInterfaceのインスタンスを生成できるようになりました。

Setのインスタンスをを operationList: の後で渡さなくとも、配列で渡せば、クラスメソッドの内部で変換されるようにしています。(aSetOfOperationSymbol asSetの部分)。

配列オブジェクトに対してasSetでSetのインスタンスに変換しています。この処理がないと

```
| se |
se := Set new. "Setを生成"
se add: #printString. "要素 #printStringを追加"
interface := Interface name: #IPrintable operationList: se.
        "新たなInterfaceのインスタンス生成"
```

としなければなりません。

SmallTip: 配列をSetのインスタンスに変換するときにはasSetを用いる

さて、ここまでのソースです。
FileIn: Metamodel-Interface.st (<=Click)

更につづく

オブジェクトの広場

[Happy Squeaking!!]

5. Squeak演習: インターセッション

5. 5 インターセッション応用

nterfaceのメタモデルの追加 つづき

次に機能の実装に入ります。今回はあくまでもメタモデル追加の例ですので、あまり本格的な実装までには立ち入りません。

まずはあるインスタンスが、特定のInterfaceに対応しているかを確かめる操作を作ってみましょう。

以下のような実装になります。
(Interface >> actionsカテゴリ)

```
by: anObject
        self operationList do:[ :each | (anObject respondsTo: each) ifFalse:[^self
error:anObject name , ' does not support ', self name, ' interface']].
        ^anObject
```

新たにでてきたのはdo:とerror: です。順番に説明していきましょう。

集合オブジェクトの要素を一つ一つ取り出したいときにdo:を使うことができます。
例えば、#('this' 'is' 'Smalltalk') という配列があり、それぞれをTranscriptに改行させて表示したいとします。この場合do:を使って、

```
#('this' 'is' 'Smalltalk') do:[:each | Transcript cr; show: each].
```

と書くことができます。

> eachの中には、要素ひとつひとつが順番に入っていきます。名前はeachである必要はありませんが、ブロック内部でのみ使われる変数として、ブロックの先頭で宣言しなければなりません。例の場合は[:each | がその部分になります。

> SmallTip: 集合オブジェクトの要素を順番に取り出すときにはdo:を用いる

次は、error: です。これは単純で、エラーノーティファイアを出し、処理を中止したいときに使用されます。

> SmallTip:エラーノーティファイアを出し、処理を中止したいときに error: を用いる

では改めてコードを見ます。

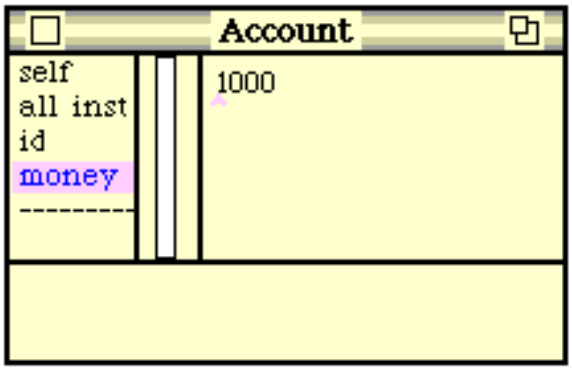
```
by: anObject
        self operationList do:[ :each | (anObject respondsTo: each) ifFalse:[^self
error: anObject name , ' does not support ', self name, ' interface']].
        ^anObject
```

Interface自身の持つ操作名の集合operationListの要素をdo:により一つ一つ取り出し、それぞれの操作が、引数として与えられた anObjectによって答えることができるかをrespondsTo:によって調べています。答えることのできない操作が有った場合には、エラーノーティファイアを出して処理を中止します。全ての操作に答えられることがわかった場合は、引数として来たanObjectをそのまま返します。

使い方は以下のようになります。

```
"inspect it"
| accIf acc |
accIf := Interface name: #IAccount operationList: #(#deposit: #getBalance
#withdraw:). "IAccount インターフェース生成"
acc := Account new. "Accountインスタンス生成"
(accIf by: acc) deposit: 1000. "AccountインスタンスをIAccountに適合して実行"
```

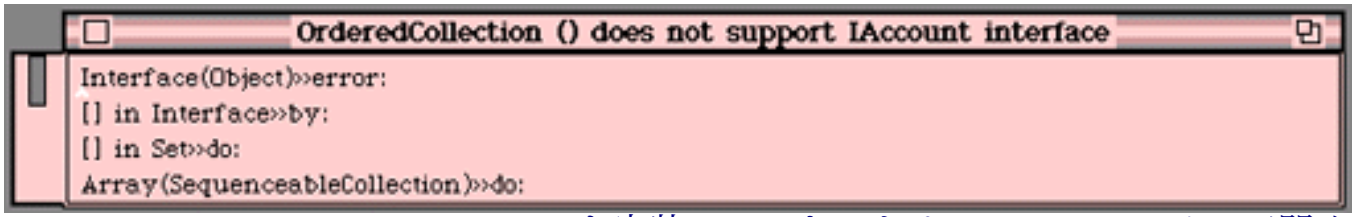
Accountインスタンスは、きちんとIAccountの操作名の集合を実装していますのでエラーは出ず、正常に実行されます。



IAccountを実装したAccountインスタンスのdeposit:の実行結果

IAccountの規定する操作を実装していないOrderedCollectionなどを使うとエラーになります。

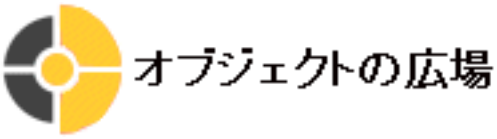
```
| accIf ord |
accIf := Interface name: #IAccount operationList: #(#deposit: #getBalance
#withdraw:).
ord := OrderedCollection new.
(accIf by: ord) deposit: 1000.
```



IAccountを実装していないためノーティファイアが開く

本来は、ブラウザ上のクラスの定義用テンプレートをカスタマイズして、instanceVariableNames: などの変数定義の他に、supportsInferfaces: と記入できるようにし、"accept"時に、そのクラス(のインスタンス)がサポートするInterfaceを調べ、実装していない場合はノーティファイアを出すといった具合にするのが理想的です。あくまでサンプルですのでこのような簡易な実装にとどめています。

更につづく



[Happy Squeaking!!]

5. Squeak演習: インターセッション

5. 5 インターセッション応用

nterfaceのメタモデルの追加 つづき 2

次はextendsの簡易版として、複数Interface間の融合(MixIn)を作ってみます。
これにより複数の異なるInterfaceから、両者の操作リストを混ぜ合わせた新たなInterfaceを作ることができるようになります。

(Interface >> actionsカテゴリ)

```
mixIn: anInterface
        | newNm newOpList newIf |
        newNm := self name, ' + ' , anInterface name.
        newOpList := Set new.
        newOpList addAll: self operationList;
                        addAll: anInterface operationList.
        newIf := self class name: newNm operationList: newOpList.
        ^newIf
```

新しく出てきたのはaddAll: です。これは、集合オブジェクトに、要素一つ一つでなく、別の集合オブジェクトの要素をまとめて追加されるのに利用されます。

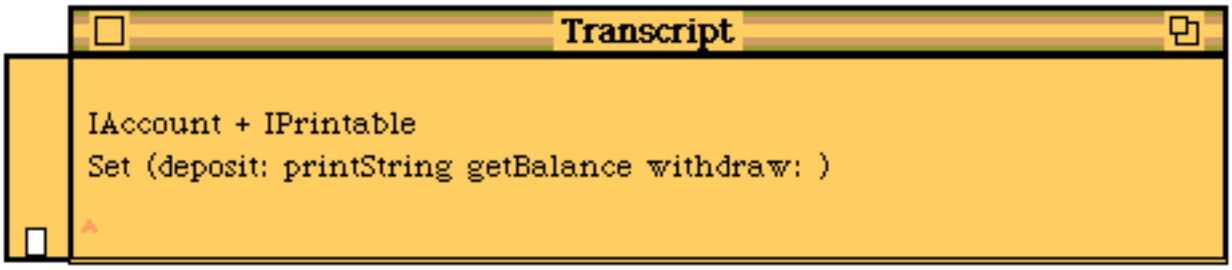
> SmallTip: 集合オブジェクトに、別の集合オブジェクトの要素をまとめて追加する場合には、addAll:を用いる

mixIn: の引数はInterfaceのインスタンスです。受取った側ではそこから名前と操作リストを取り出し、自分の名前、操作リストとそれぞれ融合を行ない、その値を設定した新たなInterfaceのインスタンスを生成し、呼び出し側に返します。

MixInの実行例は以下のようになります。

```
| accIf prtIf mixIf |
accIf := Interface name: #IAccount operationList: #(#deposit: #getBalance
#withdraw:).
prtIf := Interface name: #IPrintable operationList: #(#printString).
mixIf := accIf mixIn: prtIf. "IAccountとIPrintableのMixInの生成"
Transcript cr; show: mixIf name.
Transcript cr; show: mixIf operationList printString.
```

インターフェース名、および操作名の集合が融合されたMixIn Interfaceができたことが確認できます。

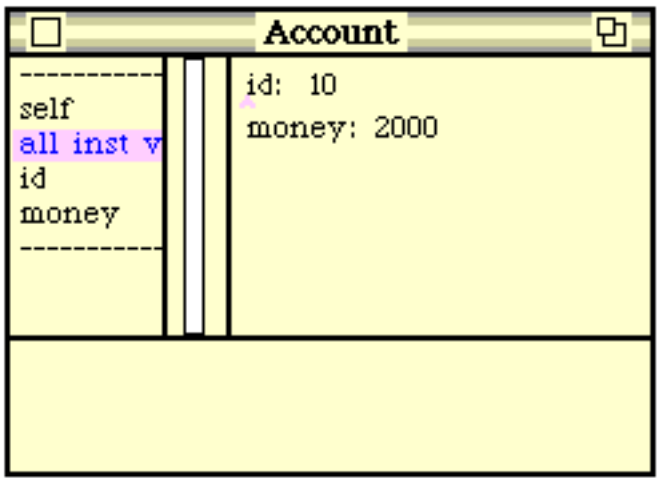


MixIn Interfaceの表示

それではMixIn Interfaceを実際に使ってみましょう。

```
"inspect it"
| accIf prtIf mixIf acc ord |
accIf := Interface name: #IAccount operationList: #(#deposit: #getBalance
#withdraw:).
prtIf := Interface name: #IPrintable operationList: #(#printString).
mixIf := accIf mixIn: prtIf.
acc := Account new.
ord := OrderedCollection new.
(prtIf by: acc) printString.
(prtIf by: ord) printString.
(accIf by: acc) deposit: 1000.
(mixIf by: acc) deposit: 1000.
```

Acccountは、printStringメッセージにも答えられる(Objectで定義されている)ので、問題なく実行されます。



MixInに適合したAccountインスタンスのインスペクト

最終的なソースコードです。
FileIn: Metamodel-Interface2.st (<=Click)

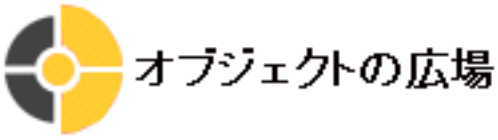
わずか2kにも満たないコードを追加しただけで、Smalltalkの世界の変更ができてしまうメタモデルの強力さ、味わっていただけたでしょうか。

本格的なInterfaceとしてはまだまだ改良の余地があります。例えば定義されたInterfaceがクラスやメタクラスと同様にブラウザで見えるようにしたり、クラスに対して、サポートできるInterfaceを答えられるようにするなどが考えられます。余力がある人はぜひチャレンジしてみてください。

###

さて、今回は、多少高度な内容でした。初心者の方はオブジェクト指向の懐の深さが味わえれば十分です。次回は初心に返り、最近何かと話題の多いデザインパターンについて、MVCを題材にしつつ解説していきます。ご期待ください。

Prev. Index Next



［Happy Squeaking!!］

6. 参考文献

●和書

「Smalltalkイディオム」 ソフトリサーチセンター 1997

日本のSmalltalker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。

"Smalltalk - an introduction to application development using VisualWorks" Prentice Hall 1995

商用のSmalltalk、VisualWorksを教材としている。メタクラスについての説明が詳しい。

● Web

NeoClasstalk Home Page

Smalltalkのメタモデル拡張版、NeoClasstalkについて、論文や実装がダウンロードできる
http://www.emn.fr/cs/research/teams/object/tools/neoclasstalk/neoclasstalk.html

Let's "do it"!!

Prev. Index