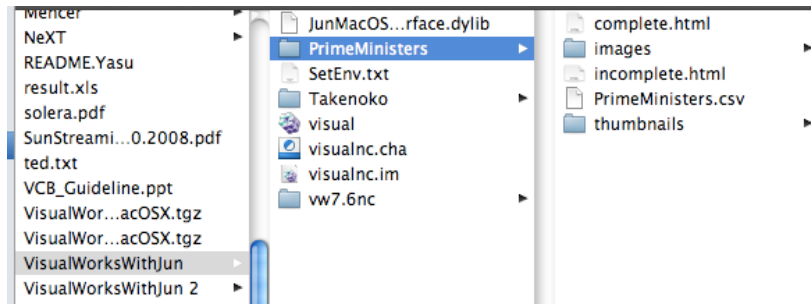


文法は終わったのでプログラミングに突入。まあファイル入出力からやるのが良からう、ということ。



こんな感じで VisualWorks のあるところにデータフォルダを置く。

なお VisualWorks 7.7 から日本語が扱えるようになった（というか Unicode 対応になった）。レベルとしては日本語対応ではなく多言語対応だった。が、国際対応、というわけではない。残念。（ここでいうところの国際対応、というのはドキュメントで言語を切り替えることができる、という程度の意味。）

なお ANSI で Smalltalk の規格が決まっているらしい。で、VisualWorks はそれに近づいていっているらしい。なんか変なの。

```
| aDirectory |
aDirectory := Filename defaultDirectory.
^aDirectory
```

とすると、aDirectory には visual.app を起動したカレントディレクトリが入る。  
Doltではなく Inspect it するとわかるかしら。  
結果は a MacOSXFilename('/Users/yasuda/Desktop/VisualWorksWithJun') となる。

```
aDirectory := Filename defaultDirectory construct: 'PrimeMinisters'.
とするとファイル名にデリミタを付けてつないでくれる。
a MacOSXFilename('/Users/yasuda/Desktop/VisualWorksWithJun/PrimeMinisters')
```

aDirectory exists とすると、存在確認ができる。（戻り値は Boolean で true となるはず）

```
| aDirectory |
aDirectory := Filename defaultDirectory construct: 'PrimeMinisters'.
aFilename := aDirectory construct: 'PrimeMinisters.csv'.
^aFilename exists ifFalse: [^nil].
```

とやって、PrimeMinisters/PrimeMinisters.csv ファイルの存在確認をして、だめなら止まる。

# 青木さんが言うには「実行システムに依存するデリミタなどをコードに入れてはイケない」。

ファイルを読み込むためのスケルトンそのいち。

```
| aDirectory |
aDirectory := Filename defaultDirectory construct: 'PrimeMinisters'.
aFilename := aDirectory construct: 'PrimeMinisters.csv'.
^aFilename exists ifFalse: [^nil].
JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do: []
    ensure: [aStream close]
```

ここで do: [] ブロックを詰めるようにする。

```
JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do: [[aStream atEnd not] whileTrue: [ aStream next ]]
    ensure: [aStream close]
```

これは中身を一字ずつ読み出す、というすけるとん。次はそれを Transcript に出す、だけ。

```
JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do:
        [[aStream atEnd not]
         whileTrue: [
```

```

        aCharacter := aStream next.
        Transcript nextPut: aCharacter; flush
    ]
]
ensure: [aStream close]

```

JunControlUtility は、単に assert [] をやって、do: [] をやって、ensure: [] をやる。  
ただしエラーがあってもちゃんと ensure: [] をやる、という制御構造。  
標準的な制御構造で書くと以下のようになる。つまり [] whileTrue: [] ensure: [] だ。

```

aStream := (aFilename withEncoding: #UTF_8) readStream.
[[aStream atEnd not]
    whileTrue: [
        aCharacter := aStream next
        Transcript nextPut: aCharacter; flush
    ]
] ensure: [aStream close]

```

次は一行ずつ読む、というもの。  
JunStringUtility は行末文字がどないなもんでもそれなりに対処する、というもの。

```

JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do:
        [[aStream atEnd not]
            whileTrue: [
                aString := JunStringUtility getLine: aStream.
                Transcript nextPutAll: aString; flush
            ]
        ]
    ensure: [aStream close]

```

次は一ファイル丸ごと読む、というもの。

```

JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do:
        [[aStream atEnd not]
            whileTrue: [
                aString := aStream contents.
                Transcript nextPutAll: aString; flush
            ]
        ]
    ensure: [aStream close]

```

で、いよいよ。  
一行ずつ読み出すプログラムを加工して、その一行の文字列を特定のデリミタに基づいて分割して Collection とするもの。

```

JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do:
        [[aStream atEnd not]
            whileTrue: [
                aString := JunStringUtility getLine: aStream.
                aCollection := JunStringUtility separate: aString dividers: (String with: $, with: Character cr).
                Transcript nextPutAll: aCollection printString; cr; flush
            ]
        ]
    ensure: [aStream close]

```

なお format するとこうなる。

```

| aDirectory aFilename aStream aString aCollection |
aDirectory := Filename defaultDirectory construct: 'PrimeMinisters'.
aFilename := aDirectory construct: 'PrimeMinisters.csv'.
aFilename exists iffFalse: [^nil].
JunControlUtility
    assert: [aStream := (aFilename withEncoding: #UTF_8) readStream]
    do:
        [[aStream atEnd not]
            whileTrue:
                [aString := JunStringUtility getLine: aStream.
                aCollection := JunStringUtility
                    separate: aString

```

```

dividers:
  (String
    with: $,
    with: Character cr).
Transcript
  nextPutAll: aCollection printString;
  cr;
  flush]]
ensure: [aStream close]

```

さてプログラムを workspace でなくファイルに書き残していこうかなあ。

□ サンプルプログラムの読み込み

'CSV2HTML4PrimeMinisters.st' asFilename fileIn

とやるというよと言われているのだけれど、そのとおりしても何も起きない。  
 実際にはこれはプログラムコードをコンパイラに渡すものなのだ。  
 どこにそれができあがるのだ？

ファイルの読み込みは FileBrowser を操作して .st ファイルを選択し、コンテキストメニューから fileIn を選べばよい。

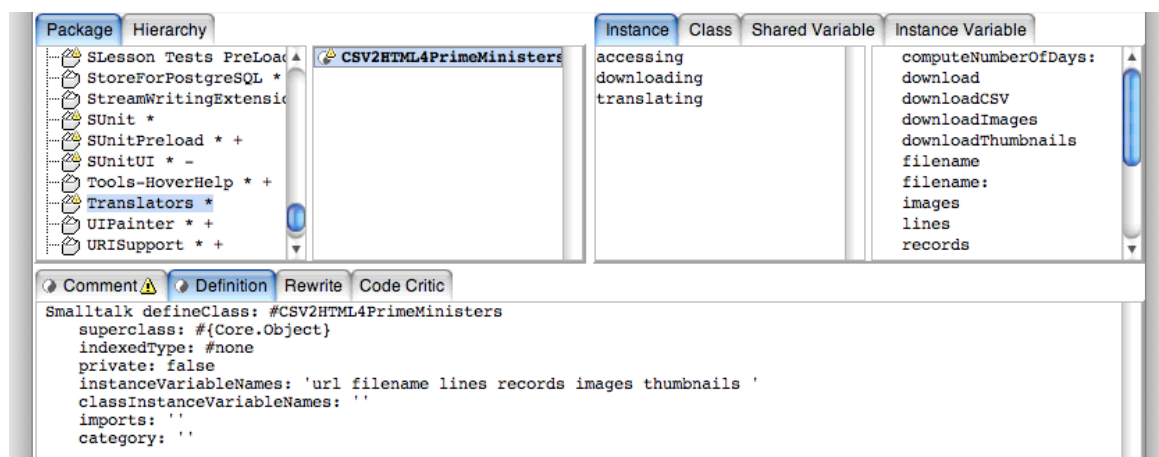
```

| aDirectory aFilename |
aDirectory := Filename defaultDirectory.
aFilename := aDirectory construct: 'CSV2HTML4PrimeMinisters.st'.
aFilename exists ifFalse: ['nil'].
aFilename fileIn

```

でもできるよん。

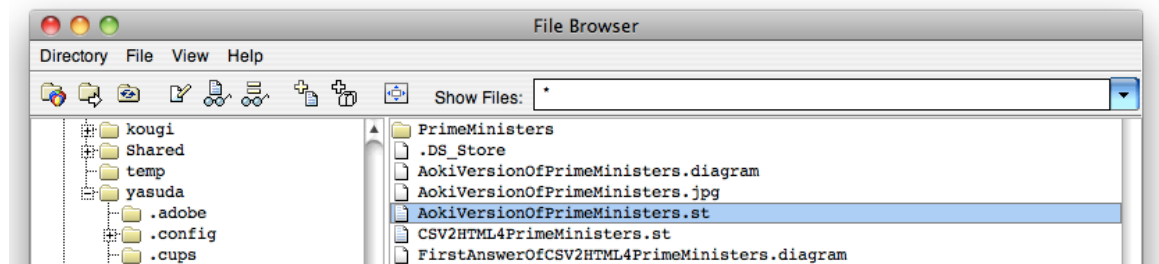
実際これで出来上がるのは CSV2HTML4PrimeMinisters というクラスができています。



このとおり。

で、これを読解して、実際に動くようにしてね、と。

ファイルブラウザでファイルを見て、そのコンテキストメニューにも FileIn がある。



==== 宿題と格闘

・ Array の要素を取り出す方法が判らない

Array を作る例はあったが、取り出す例が無かったぢやないか。(@\_@!)  
 anArray at: で取れる。

・部分文字列を取り出す

aString copyFrom: 10 to: 12 など。

aString findFrist: [ each: xxxx ]

=====

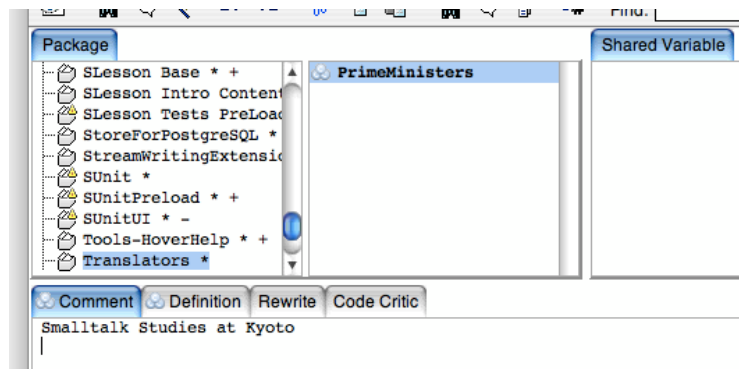
取り込んだパッケージを消すのはコンテキストメニューで Remove (Unload) というのがある。

===== パッケージの作成（ネームスペースの作成）

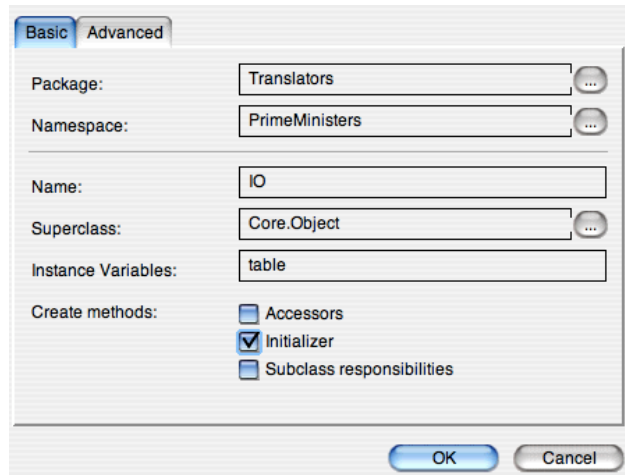
Package メニューの New Packages.. を実行して、  
Translators  
とか名前を決めてやる。するとパッケージメニューに一つ入る。

これで出来たとしても、コメントタブに警告がついてると思われる。（黄色の△！マーク）  
そこでコメントを入れて（"" で囲む必要は無し）Accept してやると消える。

ここで Class メニューの New... を選択して、その先の Namespaces を実行する。  
ここでネームスペースの名前を指定して実行。  
できるんだけど、何故かココのコメントがなくてもコメントタブに警告がつかないじゃないか。こら。



この時点ではネームスペースしかない。IOクラスをこの先に作る。Class メニューの New Class を実行。

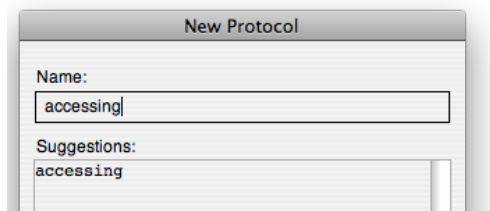


インスタンス変数 table を指定している。複数ある場合は空白区切り。  
イニシャライザ、を指定すると  
クラスメソッドの instance creation カテゴリに new メソッド、  
インスタンスメソッドの initialize-release カテゴリに initialize メソッド  
のひな形を作成される。  
アクセサ、を指定すると、インスタンス変数に対応したぶんの getter, setter が用意される。

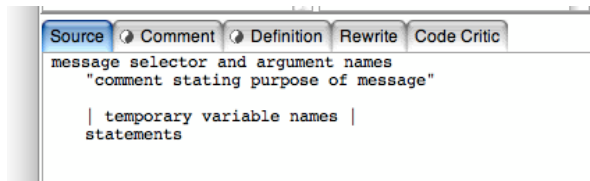
次に IO クラスのサブクラスに Reader を作るとうとする場合、ブラウザの IO クラスを選択して New Class にする。

# # ここで IO Class の new を new して initialize するものに書き換えて、IO Instance の initialize を書き換える。

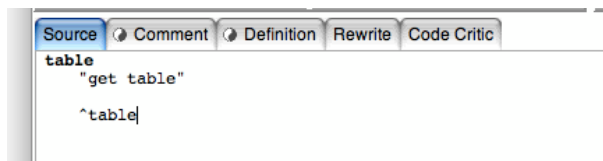
次に table に対する accessing プロトコルを作る。これは Protocol メニューがある。



次に table メソッドをこの accessing プロトコル中に入れるには、例によって method メニューには New Method がなく、ブラウザのこの accessing プロトコルのソースタブにあるコメント的な形記述を書き換えて、Accept する。

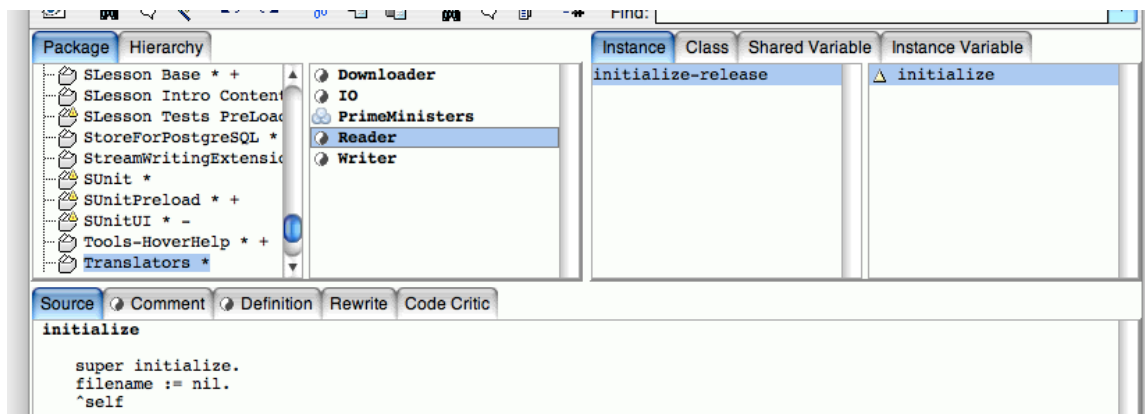


をこんなふうにするわけね

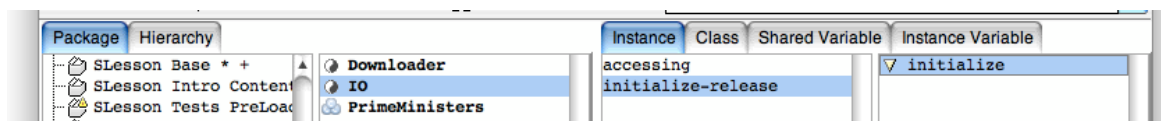


ひでえなあ。メニューでやらしてくださいよ。

ところで Reader の initialize をしようと思ってブラウザを見ると、initialize メソッドに上△がついている。



これは上位クラス IO に initialize メソッドがあるよ（上書きしているよ）と言っている。IO クラスのメソッドを見ると、



というわけ。

===== PrimeMinister 設計

僕が言うていた COBOL の例は

- ・入力データフォーマットと出力配置は独立のもので、間にメモリを置いて中立性をもたせる

というアイデアだった。

つまり青木さんが出しているようなテーブルというものを使って IO クラスが受け渡す、というのが一つの方法か。

しかしそれを追うのもつまん。100 万行を 1 万ページに生成する処理とのひらきもおおきそう。

これは濱崎さんも言うていたけれど、

- ・CSVデータ最初の一行は他とは異なる
- ・HTMLテーブル最初の一行は他とは異なる

という問題がある。テーブルの処理、というものと、ここがうまく一致すればいいのだが、なんとなくしっくり来ない。

そこで、

- ・これはテーブルというものの処理とは見なさず、
- ・複数行を相手にするバッファ（FIFO的なもの）と、
- ・一行ぶんに対応するレコード（フィールド（キー）と値のペアの集合体）を相手にするもの

があれば良い、というように考えようか。

つまりこれは行処理なのだ、と。行で見る限り手続き的なのだと。

テーブルであればカラム処理がありそうだけれど、それが今回無いじゃないの。

（それより「テーブル」というオブジェクトを無限大に取ることができないのに、処理は無限に大きくなることに問題がないように見えるじゃないの。つまり「テーブル」を「メモリ」に乗せるオブジェクトとして考えることによって処理に制約が出てきているのではないの。）

というような方針であれば、

```
read line
set line to headLine
loop until EOF:
  read line
  add line to buffer
end of loop

write headerOfHTML
write headLine as headline
loop all:
  get line from buffer
  write line as dateline
end of loop
write footerOfHTML
```

と書いて、手続き的な処理がよく見えるではないの。

これならデータが 100 万行あっても 10 行ずつに分割して処理することができるし、その場合の修正が見通しよく簡潔に出来る。つまりこれは処理そのものが手続き的だから、ではないか。

言い換えると、データをオブジェクトにして機能をそこに含ませようとする、データのサイズが制約になるが、機能をオブジェクトにしてデータをそれに扱わせる分には、データのサイズは制約にならない、ということか。

これはある種のオブジェクト指向モデル（データエンキャプスレーション）の限界（制約）なのか。

```
read line
set line to headLine
loop until EOF:
  read line
  add line to buffer
  if buffer is full then call purgeBuffer
end of loop
```

```
purgeBuffer:
write headerOfHTML
write headLine as headline
loop all:
  get line from buffer
  write line as dateline
end of loop
write footerOfHTML
```

登場人物（青木表現）は Buffer クラス(Queue?)と Record クラス(Tuple?)か。

で、思い切って Record クラス（の一部のカテゴリ）はレコードフォーマット依存の何かを入れるか？

あるいは key に対して汎用において、key のindexを発見するようにするか？

しかし後述（★）の理由から、できるだけ indexOfKana といったメソッド名を有効に使いたい。

だから Record クラスに indexOfKana とかする方が良いか？だとすると Record クラスのインスタンス変数そのものにフィールド名を入れて、それに一対一対応するgetter method として indexOfKana を設けるか。

つまり、

```
Record クラス
  インスタンス変数 : kana, dateStart, dateStop...
  インスタンスメソッド : getKana getDateStart getDateStop
                        setKana setDateStart setDateStop
```

ということか。

つまり普通の key value の get, set があれば良い、というようにはしない

record key:

record setAsKey: value:

はない、というようにするか。

（なおこの場合は

record key: (super Class) indexOfKana

のようにして使うことになる。イマイチカッコ悪い？）

★ method 名はハッシュ化される固有名詞（定値）であり、検索されない（高速にリダイレクトされる）

Smalltalk では indexOfKana のようなメソッド名を固定的に書くことを（局所化は心がけるにしても）避けてはいけない、ということを思った。

つまりメソッド名はコンパイラが定値に直してしまう、高速化の手段なのだ、と。

プログラムとデータの分離をそこまで厳しくしない方が良い、ということか。

□ #12 回 2009.11.4 プログラミング CSV2HTML

クラスを分けて作る。

Translators クラスの Attributes オブジェクトにある initialize メソッドには シングルトンを返すように書かれている。  
new メソッドには ^self singleton と書かれており、これが呼ばれる。  
singleton メソッドは「クラス変数 singleton が ifNil であれば、super new してそれに initialize を投げて yorself を返せ」となっている。  
なお初期値は super new （つまりインスタンスを作れ=インスタンスメソッドの new を動かす）ところから放り込まれる。

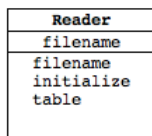
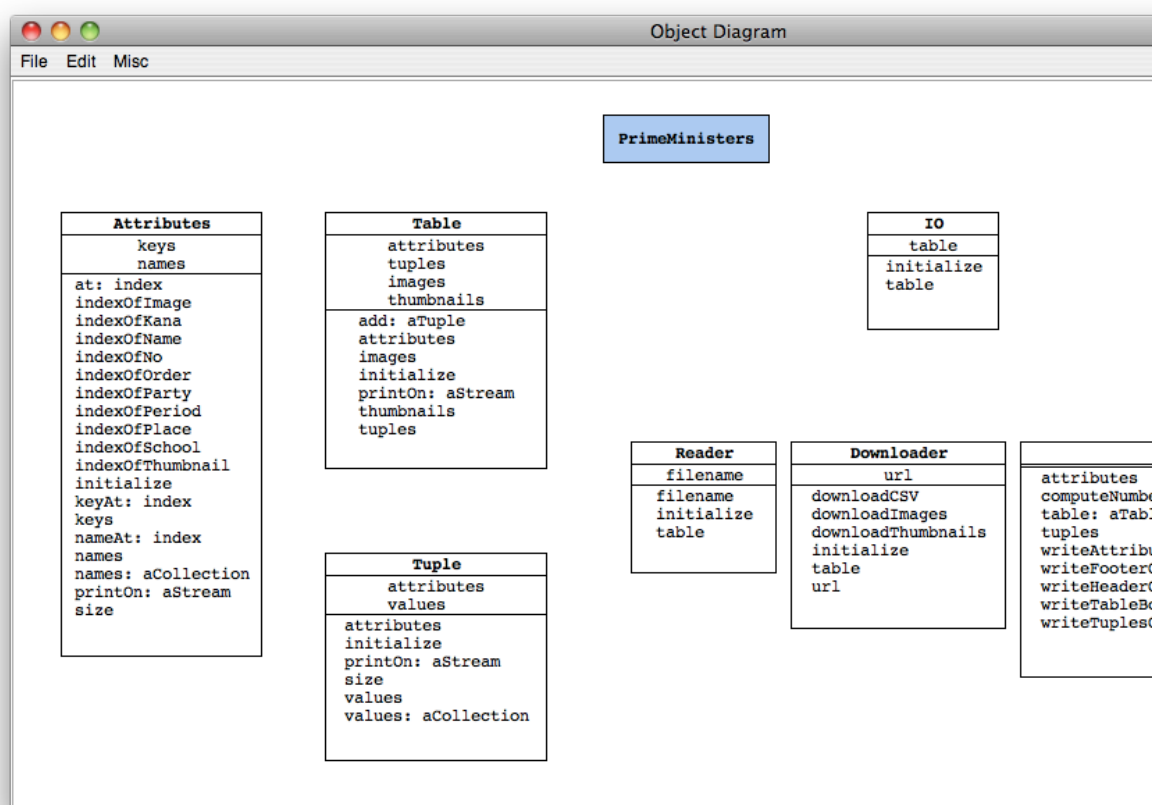
ちょっと実験。  
firstArray := Array new.  
secondArray := Array new.  
firstArray == secondArray  
を printit すると false になる。違うオブジェクトだから。しかし firstAttributes := PrimeMinister.attributes new. などとしてやると、これは true になる。

なお filein するとクラスメソッドの initialize が呼ばれる。つまり initialize メソッドに nil 化のコードなどが含まれているのはそれが理由。

（おまけ：むかーしは singleton のことを SoleInstance というようになった。孤独なインスタンスね。）

===== ダイアグラムファイルを開ける。

JunRoughObjectDiagram new; open.



は Reader クラスに filename クラス変数があり、filename, initialize, table メソッドがある、という表示。

上のダイアグラムには関係性が書かれていない。（表示されない？バグ？）

singleton にする、意味はわからない。僕には。

=== 設計

ちと自分でやってみるか。

プログラム全体構造としては、「テーブル渡しをめざす」ので、

```
aTable := PrimeMinisters.Reader table.  
bTable := PrimeMinisters.Translator aTable.  
cTable := PrimeMinisters.Writer bTable.  
^cTable.
```

このような感じになる。か。ただし青木版では Writer が HTML ファイルを作って、副作用的にブラウザが開いて表示までしちゃうので、Writer が変換系である (table というメソッド名は table をなんとかせえ、と言っている) のに合わないと言える。

僕は当初 cTable に HTML データが添付されて戻されるのか、と思ったのだけれど。(その方が上の書き方には適合しているはず。ただしその場合は最後に cTable PrimeMinisters.Display と書くことになるうなあ。

最後の戻り値に table を返すのをやめれば、

```
aTable := PrimeMinisters.Reader table.  
bTable := PrimeMinisters.Translator aTable.  
PrimeMinisters.Writer bTable.
```

という書き方もあり。(この場合 PrimeMinisters.Writer のインスタンスが戻り値になるかなと思う)

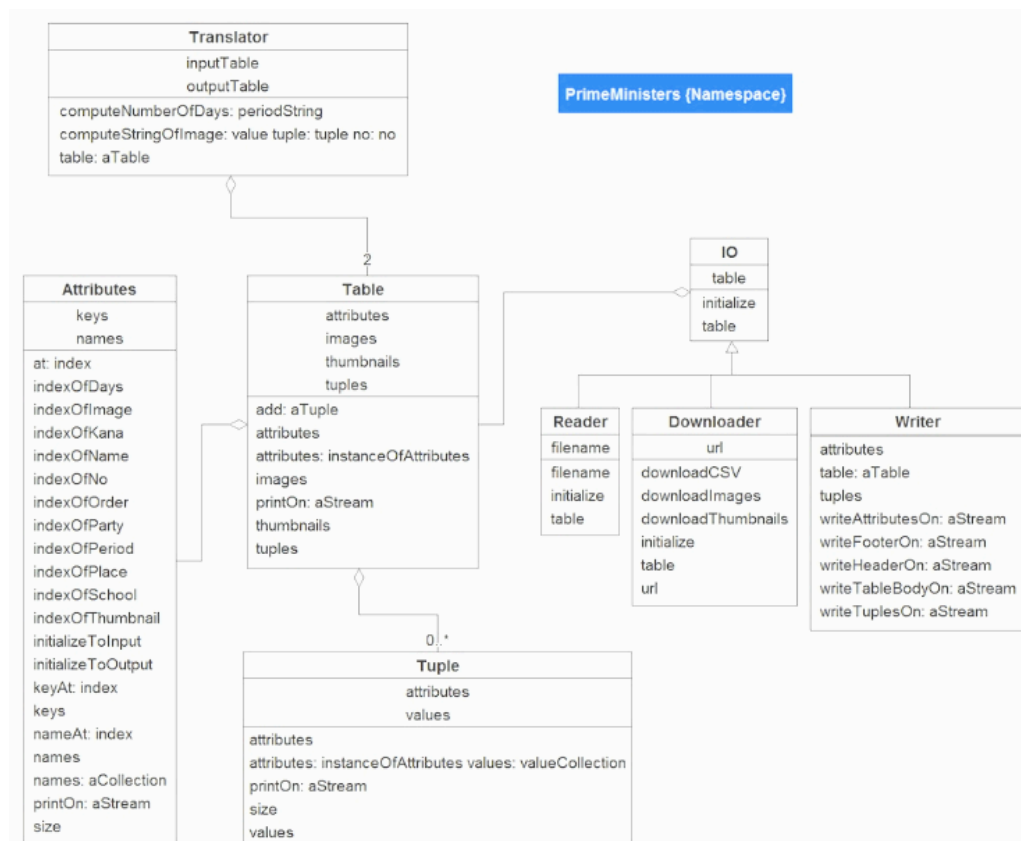
クラスメソッド呼び出し中心でテーブルオブジェクトを引数として渡すのではなく、出てきたテーブルに対して仕事をさせようとする以下のように書けるだろうなあ。

```
aTable := PrimeMinisters.Reader table.  
bTable := aTable translate.  
bTable write.
```

こんな感じ。

まず最初に use case を考えてから構成を作るなり書くなりしよう、と。つまり example を書いてから作り始める、ということか。

僕としてはこの方法かな。



この設計の面白いところは Tuple に独立性をもたせるために、Table と Tuple と両方に attribute 情報を持たせている。(インスタンス変数 attributes を持っていることに注目)

これが理由で恐らく Attributes クラスを独立させている。(Tuple だけでも幾らか仕事が出来るように機能独立させている。だけど Table は attributes を知らないわけにはいかないので両方に持たせている。)

僕がやるなら Table クラスに read, translate, write も全部持たせてしまいそうだけれど、これってたぶん reader/writer/translator の単体テストができないだろうなあ。

つまり作る人にとってはそのくらいでないと難しい、ということか。



```

## CSV2HTMLTranslator [ csvTable, processedTable, htmlTable ]

example1
"CSV2HTMLTranslator example1"
aTranslator := CSV2HTMLTranslator new.
aTranslator csvReadFromFile: 'sample.csv'.
aTranslator initProcessTable.
aTranslator translate.
aTranslator htmlWriteToFile: 'sample.html'.

--- initialize
csvTable := nil.
processTable := nil.
htmlTable := nil.

--- csvReadFromFile: filename
csvTable := CSVTable new.
csvTable readFromFile: filename. これはある意味ラッパーにしかになっていないが、これ以外では引数で csvTable を渡さねばならぬ。それがイヤ。

--- initProcessTable.   マッピング用の情報をセットする
processTable := ProcessTable new. これで attribute が生まれる
processTable initialize. ここでハードコードしてアトリビュートの中を埋める

--- translate   二つのテーブル間のマッピングをする
loop:
    csvTuple := csvTable tupleOfIndex:num.
    processTuple := processTable newTuple.
    processTuple mapFrom: csvTuple.   これは Tuple クラスを見よ
    processTable addTuple: processTuple.

--- htmlWriteToFile: filename
self writeHeader
loop:
    self writeBody
self writeFooter

## CSVTable [ anAttribute, aTable ]

--- setupFromFile: filename
anAttribute := CSVreader readLine. (anAttribute は Array)
aTable := Array new.   (aTable は Tuple の Array)
loop:
    anArray := CSVreader readLine.
    aTuple := Tuple newWithAttribute: anAttribute withArray: anArray.
    aTable addTuple: aTuple.

?? Table にも Tuple にも アトリビュートを埋めるべきか（青木式？）
?? Tuple はアトリビュート無しとすべきか

--- tupleOfIndex:num
指定された要素を返すのみ   （これを Tuple 化して返す）
anArray := aTable indexOf: num.
aTuple := Tuple newWithData: anArray.
^aTuple.

## processTable [ anAttribute, aTable ]

--- initialize
anAttribute := { 'name' 'dateFrom' 'dateEnd' ..... }
aTable := Array new.

--- newTuple
aTuple := Tuple newWithAttribute: anAttribute.
^aTuple.

--- addTuple: aTuple
aTuple から Attribute をはがして Array にして addItem する

## Tuple [ anAttribute, aData ]

--- newWithAttribute: anAttribute   新しく空データの Tuple を指定のアトリビュート(Array)で作る

```

```
anAttribute := anAttribute (引数)
^{ anAttribute, aData }
```

```
--- newWithData: aData      新しくTuple を指定のデータ(Array)と既存のアトリビュートで作る
^{ anAttribute, aData }
```

===== どりゃー！！！！！！

- ・ CSV2HTMLと言うが、実際には  
CSV -> 中間テーブル -> HTML じゃないか
- ・ つまり table を二つ作って、それを結ぶ Translator があればいいということだ
- ・ CSV to Table と、  
Init Table があって、これで作った二者を使って、  
Translate from table1 to table 2 して、  
その後、Table to HTML の変換があるんだ  
これをごちゃまぜにしようとするからおかしくなる。
- ・ なお Translate table1 to 2 は、  
loop:  
    csvTuple := csvTable tupleOfIndex:num. 一行取り出して、  
    processTuple := processTable newTuple. 空の受け皿を作って、  
    processTuple mapFrom: csvTuple. アトリビュートつき Tuple to Tuple の変換をする (Mapper だね)  
    processTable addTuple: processTuple. Map して完成した Tuple を追加

このアイデアでいいはず。

```
# この一つ上で僕が考えていたのは、CSV -> HTML に Translation をするオブジェクトがいて、
# その内部データとして CSV からのものとターゲットを用意して、Map してから HTML にすればいい、
# と思っていたのだけれど、なんだか命名や役割分担がはっきりせず、どうしても青木方式に引数でテーブルを渡したくなる
# その理由が分からなかったのだけれど、つまりこのアプリは全く異なる仕事を、データを介してつないだ (データを受け渡した) だけなのだ
```

つまり全体としては、

```
sourceTable := Reader readFromFile: 'sample.csv'.
mapTable := Translator initialize.
targetTable := mapTable translateWith: sourceTable.
targetTable writeToFile: 'sample.html'.
```

か。あれ？ いっしょちゃん？？？

あかん。どっちつかず。

===== もう一回、今度はデータ構造から

```
Tuple
    items - Array {'Yutaka Yasuda' 40 ....}
```

```
ATuple アトリビュートつき Tuple
    attribute - Tuple
    data - Tuple
```

```
Table アトリビュートと Tuple の Array
    attribute - Tuple
    body - Array of Tuple
```

で、全体としては丸ごとクラスを作って、

```
Translator 二つのテーブルを変換するクラス (で、HTML 出力は Table の機能として用意する)
    srcTable - Table
    targetTable - Table
```

```
主たる使い方としては、こんな感じ
srcTable := Table new.
srcTable readFromCSV: .....
targetTable := Table new.
targetTable setAttribute: MapInfo targetAttribute. これは単に情報を得るだけのもの
self translate.
targetTable writeToHTML: .....
```

でいいか。

```
srcTable := Table readFromFile: .....
targetTable := Table initWithAttribute: MapInfo targetAttribute. これは単に情報を得るだけのもの
self translate.
targetTable writeToFile: .....
```

つまり Table クラスに対して、CSV read と HTML write 機能を付ける感じ。  
translate は実際には Translator クラスの中で、

```
loop:
  srcATuple := srcTable indexOfATuple: num.    一行取り出して、
  targetATuple := targetTable newATuple.  空の受け皿を作って、
  targetATuple mapFrom: srcATuple.  アトリビュートつき Tuple to Tuple の変換をする (Mapper だね)
  targetTable addATuple: targetATuple.  Map して完成した Tuple を追加
```

とする。

Table クラスのメソッドは、

Tuple アクセスについてはこんなの。

```
indexOfTuple: num    指定の行を取り出して A なし Tuple として返す
anArray := body indexOf: num. 単にアレイを返す
```

```
indexOfATuple: num    指定の行を取り出して A つき Tuple として返す
anArray := self indexOfTuple: num.
anATuple := ATuple new.
anATuple setAttribute: attribute.
anATuple setData: anArray.
^anATuple.
```

```
newATuple          空データの A つき Tuple を返す
anATuple := ATuple new.
anATuple setAttribute: attribute.
^anATuple.
```

```
addATuple: anATuple  A つき Tuple のデータ部分を足す
aTuple := anATuple tuple.
self addTuple: aTuple.
```

```
addTuple: aTuple    A なし Tuple のデータ部分を足す
body addItem: aTuple    これって Array のための function (メッセージ)が使えるはずよね
```

テーブルアクセスについてはこんなの。

```
readFromCSV: filename    ファイル(CSV)から読む
attribute := CSVreader readLine. (anAttribute は Array)
loop:
  anArray := CSVreader readLine.
  self addTuple: anArray.
```

```
writeToHTML: filename    ファイル(HTML) に書く
self writeHeader
loop:
  self writeBody
self writeFooter
```

```
setAttribute: anAttribute    アトリビュートだけを設定する
```

ATuple クラスのメソッドは、

```
mapFrom: anATuple  A つき Tuple どうしてマッピング変換をする
loop:
  ... なんて書くんだろう
```

□ 12/2

浜崎さんからまず紹介。

CSV になく HTML にしかない項目をどうやって合成するか、あるいは特定のフィールドを出さない方法について。  
(方法というか、元のプログラムがそこをハードコードしていたので、それを分離したい、ということだろうか?)

CSVRecord  
CSVString  
CSVTable

HTMLObject  
HTMLContainer  
HTMLGenericTable  
HTMLTable

```
HTMLGenericTableItem
HTMLTableItem
... みたいに構造化してクラスを用意
```

```
HTMLCssStyle
HTMLDocument
HTMLHorizontalLine
HTMLHyperLinkToImage
HTMLPageTitle
HTMLSignature
HTMLText
```

DateUtility は日付から日数を得るような、そういうもの。  
NumberUtility はカンマ区切り表示に変えてくれるだけ。

primeMinisterTable

```
csvRecordItemKeys
^#(#no: #order: #name: #kana: #.... )
```

という感じで、配列としてキーを返すようにしておく。結局これがマップテーブルになる。(マッピング用のインデックス名並びということか)

ちうかこの #no: みたいなのは、最終的に perform 関数でメッセージセクタを動的に指定するのにそのまま使われる、みたい。  
そういう事が出来るのか。ふむ。

結局各項目のマッピングは、

・ CSV の入力 (Read)

は、第一行目に項目名が埋まっているので、これをキーとして覚えておく。  
この項目名が後に内部テーブルへマップされるときに使われる。

・ CSV -> primeMinister の内部テーブル

は読み込んだ CSV 項目の名前と、内部テーブルの名前のマッピングを示す Array を用いる。  
というかそういう Array がある。

```
^#(#no: #order: #name: #kana: #.... )
```

と

```
^#(#'番号' #'順位' #'名前' #'カナ' #.... )
```

みたいな感じ。  
この二つの配列は同数の要素を持ち、前から順に対応させる。

・ primeMinister の内部テーブル -> HTML 出力

ここで在位日数のような、元のデータになかったはずの項目などを出力するように指定する。というかやはり、  
そういう Array がある。

```
^#(#htmlNo: #htmlOrder: #htmlNissuu: #kana: #.... )
```

と

```
^#(#'番号' #'順位' #'在位日数' #'カナ' #.... )
```

みたいな感じ。  
この在位日数を出そうとしたときに、primeMinister クラスの htmlNissuu: メッセージは、自分が持つ nissuu プロパティを出すのではなく(他の項目はそう  
なってる)、就任、退任の日から計算する手続きが埋められている。

==ふうむ。  
この各フィールドの結びつきを Array の位置で、その要素としてフィールドの変換処理を実現するセクタで書く、というのは良いかなと思われる。  
が、フィールドの名前 (id) に処理の内容(これは各フィールドごとにユニークではない)を重ねるのはいまいち気持ち悪い。  
つまり出力側の変換指示は例えばこんな感じで書けないか？

```
( ('no' #translateCopy:)
  ('order' #translateCopy:)
  ('nil' #translateNissuu:)
  ('kana' #translateCopy:)
  ('image' #translateImage:) )
```

こう書かない方がむしろ良いのか? (@\_@!)?  
つまり要素の独自性や要素を性質をどこに埋めるか、ということか。  
うう、これをやるとしかしてきたテーブルのカラム名を指定する場所がない、困った、、  
み、三つ組み？

==

ちやうよ、こんなのハードコードすりゃええじゃん。結局上記のテーブルだってハードコードするんじゃないか。  
self translateCopyFrom: #no to: #no.  
self translateCopyFrom: #order to: #order.

```

self translateNissuuWithStart: #from withStop: #end to: #nissuu.
self translateCopyFrom: #kana to: #kana.

```

でええやん。透過だし、各項目の関係性がよく見える。加工も楽だ。  
もし上記の ('no' #translateCopy:) をデータ読みする、というのであれば、

```

self translateCopyFrom: 'no' to: 'no'.

```

で動くように作ればよい。# を使うのは結局コンパイラに静的に処理させて探索せず参照したい一心だから。

(もちろんこうするとエラーチェックコードを作る事が出来ないのはちょっといやだけれど、、、)

```

self tuples
do:
[ :tuple | outputTable add:..... ]

```

という感じ。

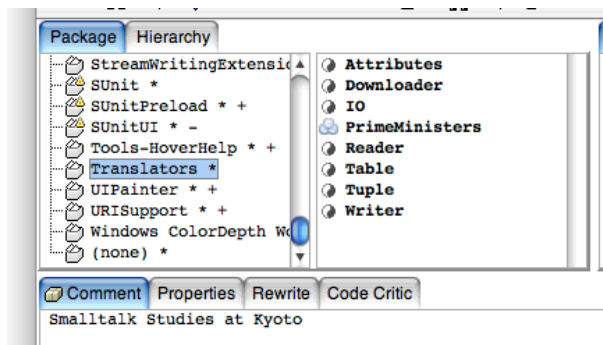
===== 12/19 Smalltalk 勉強会 in 名古屋 ハッカソンの部

■ とりあえず作る。

□ step 1.

とりあえずテストランするための Tuple 構造を既存プログラムをひな形に作る。

ファイルの読み込みは FileBrowser を操作して .st ファイルを選択し、コンテキストメニューから filein を選べばよい。  
Filing in from:  
/Users/yasuda/Desktop/Visualworks/CSV2HTML/20091007.st



クラスブラウザに Translators パッケージが読み込まれた、けど、これ Filein したあと、ブラウザをあげ直さないと出てこなかったよ。

Table クラスの Class method に example があるので、例えばこのあたりで試しに実行できる。

```

"PrimeMinisters.Table translate."

```

が、これの中身は、

```

aTable := PrimeMinisters.Table translate.

```

なので、translate メッセージを追うと、

```

aTable := PrimeMinisters.Reader table.

```

```

PrimeMinisters.Writer table: aTable.

```

となっている。とりあえず Reader から追うか。驚いた事にこれは単項メッセージで table というものになってる。へええ。

と、おもったら今日はここまで。まあ直下のマッピング処理のコードアイディアを聞けたから良しとしよう。

□ 問題のマッピング処理

```

self translateCopyFrom: #id to: #id.
self translateCopyFrom: #no to: #no.
self translateNissuuWithStart: #from withStop: #end to: #nissuu.

```

みたいな事をしようと思ったら、受け取った #id を Tuple の Label にある #つき名前から Data の Array の番号にマップしないといけない。  
で、どうするかというと、Dictionary というのをを使うと良いよというのが山宮さんの意見。

```

| aDict |
aDict := Dictionary new.
aDict add: #id -> 1.
aDict add: #no -> 2.
aDict add: #name -> 3.

```

こんな感じで作って、こんな感じで取り出す。  
aDict at: #name.

「->」というのは Association というものだそうで、#id -> 1 などそのまま Inspect it すればそこに key, value が見える。

(Dictionary の add: が引数を Association だと思っている点に注意)

つまり二つの値を key と value のペアにするための手続き。これを add: する、というのが上の記述。なるほどなれた人には素直に見えそう。他にもうちょっと一般的な at: put: も使える。(これは Association とは違って、普通の Array などでも普通に使える汎用性の高い記述と言える) つまりこんな感じ。

```
| aDict |
aDict := Dictionary new.
aDict at:#id put:1.
aDict at:#no put:2.
aDict at:#name put:3.
```

これを使うと恐らく、translateCopyFrom: srcKey to: destKey は、

```
srcIndex := srcDict at: srcKey.
destIndex := destDict at: destKey.
srcValue := srcArray at: srcIndex.
destArray at: destIndex put: srcValue.
```

という感じ？実は一発で書けるはずなので、

```
destArray
  at: (destDict at: destKey)
  put: (srcArray at: (srcDict at: srcKey))
```

かな。

コード内部ではいずれもハッシュ探索と間接ポインタ参照だけで作れているので恐らく速度的にもマシだと思う。

(が、青木モデル(全行 Tuple で label, data の両方を内包する)でも、実際には label は共通のリテラル(シンボル)なので、間接参照たった一発だけなのよ。つまりメモリ占有量が増えるわけでもない。(いやまあ各行 32 バイトくらいは増えるだろうけど))

□ トランスレータの全体設計

Reader が aTable を返す

このとき先頭一行については srcDict を構築するために特別扱いする。(このタイミングで srcDict を作る) 残り行についてはインスタンス変数 recordArray (二次元配列)に入れる。

Translator クラスのインスタンス変数に

- ・ srcDict (CSVのラベル並び)
- ・ destDict (HTMLの列ラベル並び)

を与えて、引数に srcArray を与える。すると destArray を返す。

つまり

```
aTranslator := Translator newWithSrcDict: srcDict withDestDict: destDict.
destArray := aTranslator translateFrom: srcArray.
で変換する。
```

Writer クラスは Table 相手にテーブル形式で HTML を吐くだけ。

```
writeHeaderOn:
writeBodyOn:
writeFooterOn:
```

くらいか？

全体には recordArray から each で取り出してループで呼び出す  
それはまあ Table クラスの中に書くべきなのかな？つまり

```
csvTable := Reader loadFromFile: 'sample.csv'.
srcDict := Mapper makeDictFrom: (csvTable label).
destDict := Mapper makeDestDict.
aTranslator := Translator newWithSrcDict: srcDict withDestDict: destDict.
newTable := csvTable translateWith: aTranslator.
Writer writeHTMLwith: newTable.
```

とかなんとか、でも何か面倒？でもないか。

Mapper クラスに少なくとも HTML 出力側のラベルの並びはハードコードせないかん。

csv にどういうものが要求されるかも書いとんといかんのか。

結局 Table クラスの中は aTable インスタンス (インスタンス変数 labels と columns の Array をもつだけ) を作ったりするだけ？

んー？？？

- ・ mapping table はしかし srcTable -> destTable の変換で、用意しないといけないのは

```
src - #(#id #no #name)
dest - #(#id #no #name)
```

といったもので、あれば例えば Mapper class の instance として定値として書けば良い？

(あるいは srcLabels メッセージの返り値に ^ #(#id #no #name) とハードコードするなどして。

しかし csv データ行目の値 'id' と #id のマッピングをわざわざ作らなければならないのはおかしくないか？

で、あれば dest 側だけでよい。

- ・ しかし csv 以外のデータ形式からテーブルを作ったとするとどうする？やはりあった方が楽では？

ちょっと堂々巡りか。

げ、CSV ファイル最初の一行は日本語ラベルやったわ。

人目,代,氏名,ふりがな,在位期間,出身校,政党,出身地,画像,縮小画像