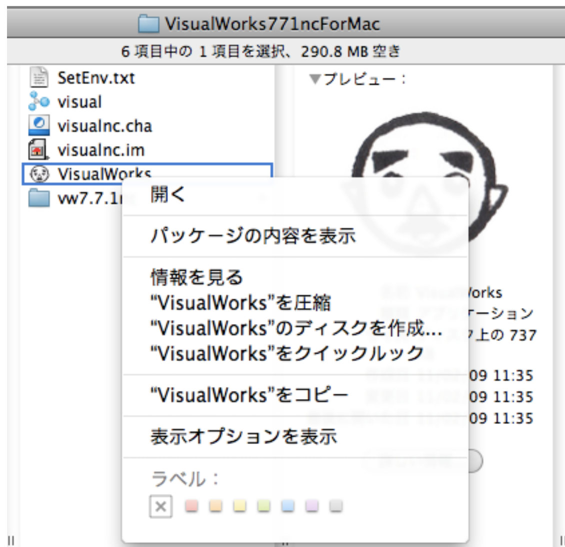
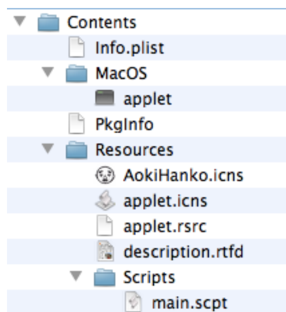
□ 2011.3.2

本日は最初に Macintosh 用の簡単起動アイコン（AppleScript 製）の紹介。

今回 dmg 形式で VisualWorks 771 が配られたが、そこに AppleScript で書いた起動用スクリプトがついている。



VisualWorks （青木）アイコンを対象にそのコンテクストメニューから「パッケージの内容を表示」を選択すると中身が見える。NeXTSTEP 由来の古い構造。



この main.scpt ファイルが実行される。ダブルクリックすると AppleScript Editor が開いてくれる。
コード内容概説。

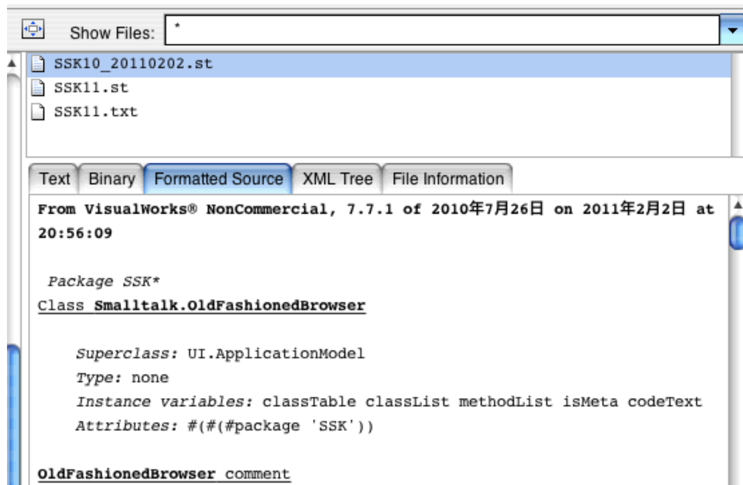
```
tell application "Finder"  Finderに向かって投げる
	set aPath to get (path to me) Mac形式のパスを得る
	set aFolder to get (folder of aPath as Unicode text) ダブルクリックされた自分のディレクトリを得る
	set aDirectory to get POSIX path of aFolder  これを Unix 的な / 区切りのものに替える
end tell
set aShellScript to "(cd " & aDirectory & "; ./visual.app/Contents/MacOS/visual ./visualnc.im 2>/dev/null)" 後述
do shell script aShellScript
```

最後の一行で shell script を生成して実行する。中身は：
cd カレントディレクトリして ./..... /visual ./visualnc.im つまり VisualWorks を image つきで起動するだけ。

・気を取り直して前回の続き

復習から。

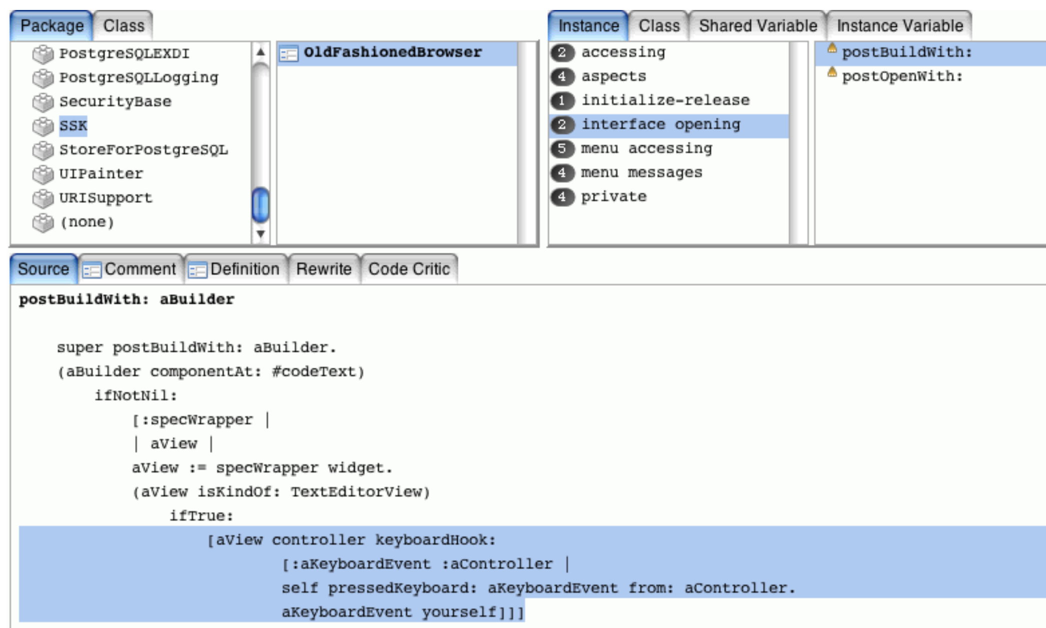
ファイルブラウザを開いて、前回の資料 SSK10_20110202.st の中身を確認。
XML 形式のデータなので、Formatted Source タブを指定することで直接中身を見ることが出来る。



今日のテーマは reflection / reflective だ、と宣言あり。
Alan Kay はすべてをメモリの上でやる、Unix 的なファイルベースでものごとをやらない、という設計方針を建てたのだ、と。

＃ファイルからメモリに移して、またファイルに保存する構造が広まりすぎたのは確かだなあ。
＃何故メモリの中にある構造化されたデータをファイルI/O のためにシリアライズせねばならんのか、という話は昔からある。
＃僕らは構造化されたデータをそのまま扱えて良いはずなのに。

ともあれ SSK11.st を file in する。
今回は postBuildWith: が前回と少し違う。



前回はハイライト部分にキーボードのフックコードを延々と書いたが、今回は pressedKeyboard: メソッドを呼び出すようにした。
pressedKeyboard: は private の instance メソッドとして用意した。

```
pressedKeyboard: aKeyboardEvent from: aController

    | aMenu anItem |
    aMenu := self menuAt: #codeTextMenu ifAbsent: [^nil].
    anItem := aMenu menuItemWithValue: #acceptCodeText:from:.
    self selectedClass isNil ifTrue: [anItem disable] ifFalse: [anItem enable].
    anItem := aMenu menuItemWithValue: #formatCodeText:from:.
    anItem enable
```

次、postOpenWith:

```
postOpenWith: aBuilder

    super postOpenWith: aBuilder.
    self updateMenuIndication
```

ここから呼び出される updateMenuIndication を見ると、

```
updateMenuIndication

    self updateMenuIndicationInClassList.
    self updateMenuIndicationInMethodList.
    self updateMenuIndicationInCodeText
```

メニューの各部分をそれぞれ更新する。
さてこの状態で実行すると、以下のようにコンテクストメニューは現れるがすべて disable されている。

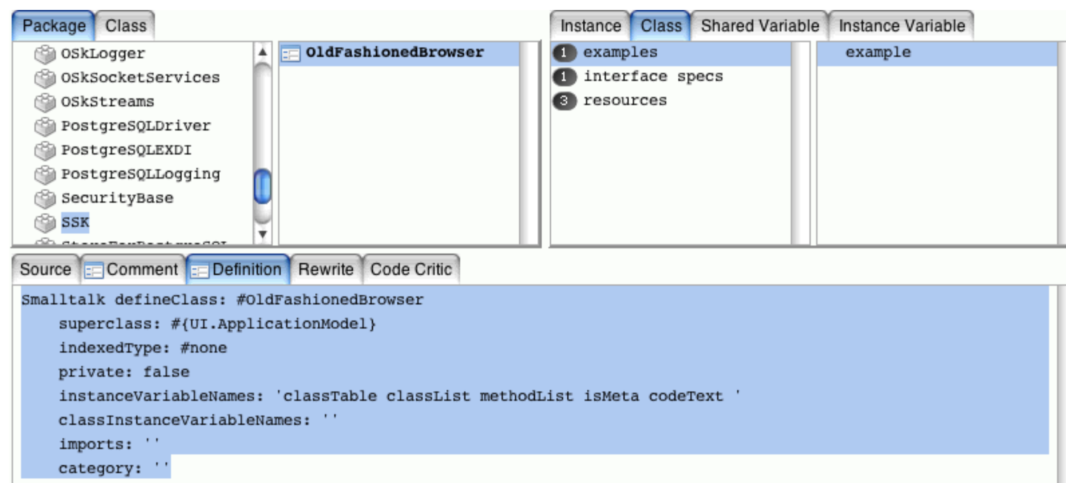
```
AbstractBrowserEnvironment in {Refactory.Browser}
AbstractBrowserNavigator in {Refactory.Browser}
AbstractChangeList
        Accept
        Format
```

キーを叩いて何か入力があると、これらは enable になる。

今回はこの accept および format の機能を実装しよう。
それもこの自作ブラウザを使ってブラウザ自身を改造する、つもりでやろう。（実際にやった）

まずワークスペースをあけて実験。

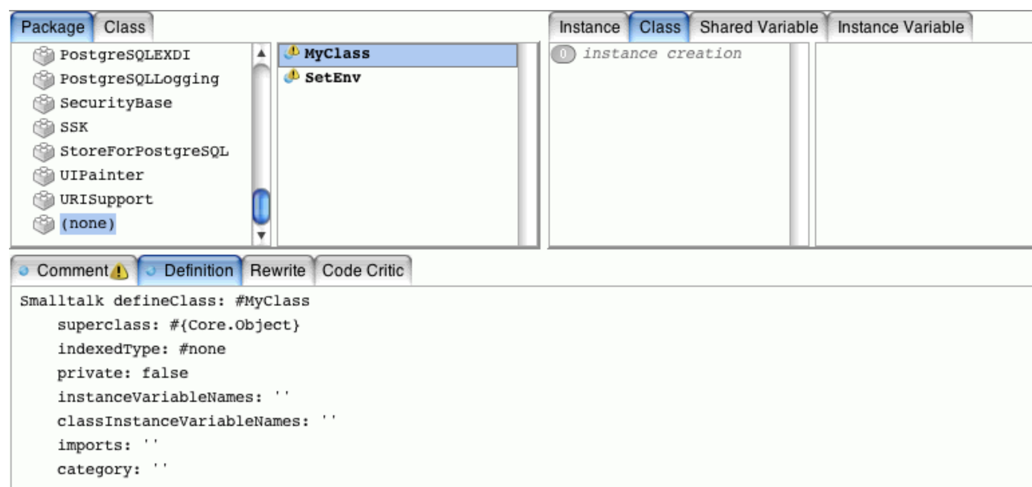
OldFashinedBrowser の definition をコピーしてそのままワークスペースに貼り付け。



Workspace で下のように変更。つまり典型的なクラスに対する定義を自分で作ってしまおうということ。

```
Smalltalk defineClass: #MyClass
	superclass: #{Core.Object}
	indexedType: #none
	private: false
	instanceVariableNames: "
	classInstanceVariableNames: "
	imports: "
	category: ".
```

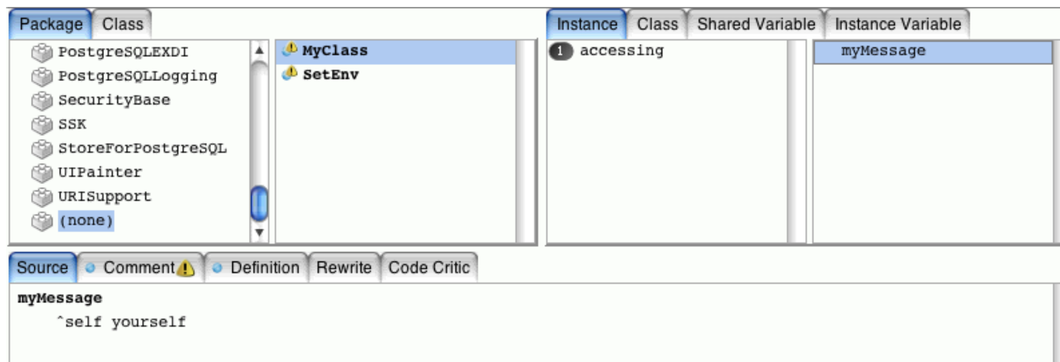
要するに MyClass という名前の、Object 直下のクラスを意味するように。
これを do it すれば MyClass クラスが出来る。
システムブラウザで確認。出来た。（パッケージが none になっているのが面白い）



続いて Workspace に下記のように書いて do it。

```
MyClass
	compile: 'myMessage
	^self yourself'
	classified: #'accessing'
```

出来た。



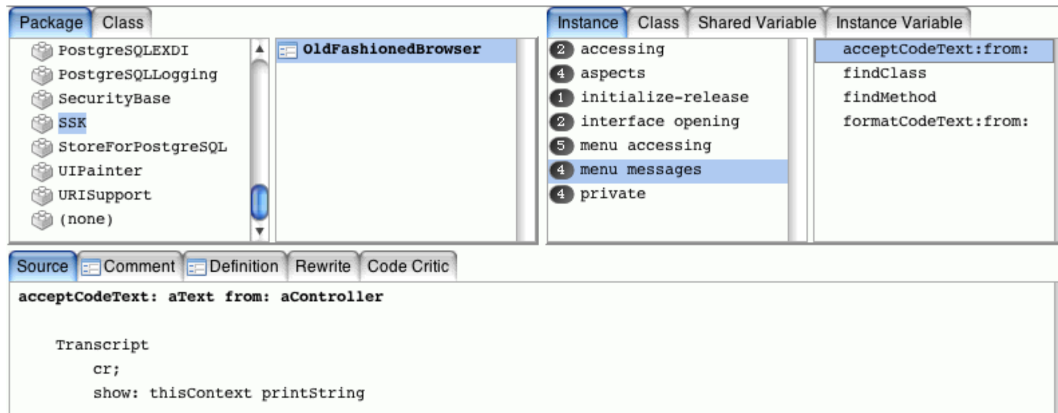
＃ところでこの時点でおかしな事に気がついた。
＃僕は最初誤って myClass （先頭を小文字）としてクラスを生成したはずなのに、実際のクラス名は MyClass になっていた。
＃勝手に直すとは、すごいことをするなあ。

というわけでこの機能を使えば（つまり Class にめがけて compile: classified: などと言ってやれば）ブラウザの Accept が作れるではないか。

なお、削除は下記の通り。
MyClass removeFromSystem

さて実装。

acceptCodeText: を直す。以下が初期状態（前回までの状態）。単に Transcript にコンテクストを出すだけだった。



ここを修正。

＃以下、段階的な修正が行われたので典型的な途中経過を残す。

一段階め。とりあえず全体的な動作について書く。

acceptCodeText: aText from: aController

```
| aString compilerClass |
aText isEmpty ifTrue: [^nil].  コードフィールドが空だったら何もしなくて良いよね
self selectedClass ifNil: [^nil].  クラスが選ばれてなかったら何もできないよね
aCode := aText asString. コードを取ってきて準備
self selectedMethod メソッドフィールドについて、、、
    ifNil:    空なら（メソッドフィールドで何も選択されていなかった＝クラスしか選択されていなかった＝この場合はクラスの definition が出る）
        [compilerClass := Kernel.Compiler.  コンパイラを呼び出して
        compilerClass evaluate: aCode]  コード部分を evaluate せえ（するとクラスが生成されたりするはず）
    ifNotNil: [:aMethod | ]  （メソッドが）指定（選択）されていたら、、、、どないしょう？
```

この段階ではまだメソッドが選択されている時に accept すると何が起きるかは書いていない。

二段階目。
コンパイル（本来の accept 相当）するように処理を追記した。
メソッドが選択されていた場合は、そのメソッドの中身（コード）が表示される。
が、もしメソッドが選択されていない場合はちょっと違うものが出る。

まずインスタンスラジオボタンが押されていたら、そのクラスの定義が出る。

```
ParseTreeSearcher in {Refactory.Browser}
ParseTreeTransformationRule in {Refactory.Browser}
Part in {Tools.Trippy}
PartialURL
PartListAbstractInspector in {Tools.Trippy}
PartPort
```

instance　class

```
Smalltalk.OS defineClass: #PartialURL
    superclass: #{OS.URLwithPath}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
```

逆にクラスラジオボタンが押されていたら、インスタンス変数だけが出ている。

```
ParseTreeSearcher in {Refactory.Browser}
ParseTreeTransformationRule in {Refactory.Browser}
Part in {Tools.Trippy}
PartialURL
PartListAbstractInspector in {Tools.Trippy}
PartPort
```

instance　class

```
OS.PartialURL class
    instanceVariableNames: ''
```

が、インスタンス変数自身はインスタンス側表示でも出ている。というわけでこちらについては accept しても何もしないことにする。
つまり処理フローとしては以下のようになる。

```
if　メソッドが選択されていない then
    if クラス側 then
        なにもしない
    else // インスタンス側
        code text フィールドの中身をクラス定義だと解釈して「メッセージとして実行」する
    end
else // メソッドが選択されている
    メッセージを code だと解釈して「当該クラスに登録」する
end
```

以下、この段階の実際のコード

acceptCodeText: aText from: aController

```
    | aCode |
    aText isEmpty ifTrue: [^nil].
    self selectedClass ifNil: [^nil].
    aCode := aText asString.
    self selectedMethod
        ifNil:
            [| compilerClass |
            self isMeta value ifTrue: [^nil].  クラス側であれば何もしない
            compilerClass := Kernel.Compiler. コンパイラーを用意
            compilerClass evaluate: aCode]   フィールドのテキストをコンパイラに evaluate させる
        ifNotNil:
            [:aMethod | 引数で method を受け取り、
            | aClass aProtocol |
            aClass := self selectedClass.  選択されているクラスを調べ
            aProtocol := aClass organization categoryOfElement: aMethod selector.  （後の *1 参照）
            Transcript
                cr;
                show: aProtocol printString]  確認のために出してみるか
```

*1 ちょっと読めなかったところ
(aClass organization) categoryOfElement: (aMethod selector).
つまり aMethod のセレクタ名を得て、それが当該クラスの保持しているどのカテゴリに属しているかを調べる。

青木さんはこれで実行していたが、ちょっと怖いので実験用のクラスを作ることにする。以下を workspace で do it。

```
Smalltalk defineClass: #AAAClass
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: ''.

AAAClass
    compile: 'myMessage
    ^self yourself'
  classified: #accessing.
```

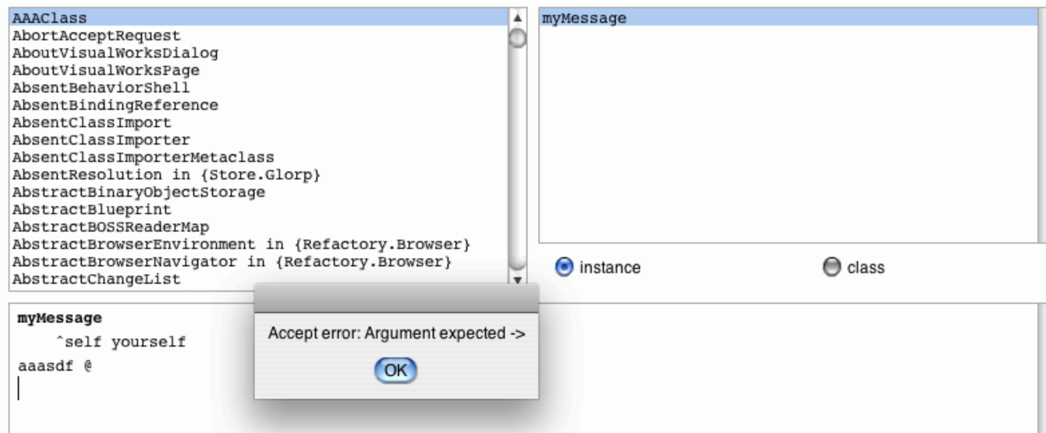
これで上の acceptCodeText を実行させてみると、Transcript には正しく #accessing と出た。
動作を確認できたので、Transcript すぐ後に二行追加（青文字箇所）。

```
		aClass := self selectedClass.  選択されているクラスを調べ
		aProtocol := aClass organization categoryOfElement: aMethod selector.  （後の *1 参照）
		Transcript
			cr;
			show: aProtocol printString.  確認のために出してみるか
		aClass compile: aCode classified: aProtocol.   得られたクラスに対してコードとプロトコルを与えて登録処理
		self codeTextBlock value]   もう一度結果を取り出しておく（のかな？これは何のため？）
```

三段階目。
危ない処理（コンパイルを含む）については block にしてエラーハンドラを入れるようにする。（そういう作法なのである）
青文字部分が追加箇所。

```
acceptCodeText: aText from: aController
	| aCode aBlock |
	aText isEmpty ifTrue: [^nil].
	self selectedClass ifNil: [^nil].
	aCode := aText asString.
	aBlock :=
		[self selectedMethod
			ifNil:
				[| compilerClass |
				self isMeta value ifTrue: [^nil].
				compilerClass := Kernel.Compiler.
				compilerClass evaluate: aCode]
			ifNotNil:
				[:aMethod |
				| aClass aProtocol |
				aClass := self selectedClass.
				aProtocol := aClass organization categoryOfElement: aMethod selector.
				Transcript
					cr;
					show: aProtocol printString.
				aClass compile: aCode classified: aProtocol.
				self codeTextBlock value]].
	aBlock on: Object errorSignal
		do: [:anException | ^Dialog warn: 'Accept error: ' , anException messageText asString]
```

これで例えばコンパイルエラーを起こすようなものがあった場合、何かしらのエラーが見えるはずである。試しに起こしてみる。



acceptCodeText についてはこれで終了。

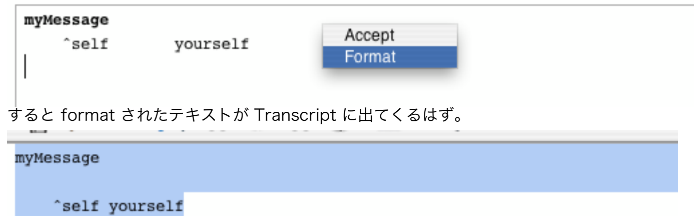
つづいて formatCodeText の方も作る。

一段階目。全体構造をとりあえず書く。

```
formatCodeText: aText from: aController
	| aCode parserClass |
	aText isEmpty ifTrue: [^nil].   コードフィールドが空だったら何もしなくて良いよね
	aCode := aText asString.   コードを取ってきて準備
	parserClass := Refactory.Browser.RBParser.
	self selectedClass   クラスについて、、、
		ifNil: []   クラスが選ばれてなかったら何もできないよね
		ifNotNil:   何か選ばれていたら
			[:aClass |   そのクラスを引数にとって
			self selectedMethod   メソッドフィールドについて見て、、
```

```
                    ifNil: []   空なら（メソッドフィールドで何も選択されていなかった＝クラスについてだけ選択＝クラスの definition が出る）
                    ifNotNil:   何か選ばれていたら
                        [:aMethod |   引数で method を受け取り、
                        aCode := (parserClass parseMethod: aCode) formattedCode asText.   aCode を整形してまた aCode にバインド
                        Transcript   まあとりあえず出してみるか
                            clear;
                            show: aCode asString]]
```

この状態で実行した。



すると format されたテキストが Transcript に出てくるはず。

```
myMessage

    ^self yourself
```

出た。（一行間が開けられたり、無駄な空白が取られていたり）
それが確認できたので、Transcript は取り除いて、以下のように最後の一行（青文字）を追加。

```
                    ifNotNil:
                        [:aMethod |
                        aCode := (parserClass parseMethod: aCode) formattedCode asText.
                        self codeText value: aCode]]
```

つまり今開いている codeText の value: を整形した aCode に入れなおす、と。
これで format, accept ができるブラウザが出来た。

二段階目。
先ほどと同じ思考で、危ないものはブロック化してエラーハンドルする、と。

```
formatCodeText: aText from: aController
    | aCode parserClass aBlock |
    aText isEmpty ifTrue: [^nil].
    aCode := aText asString.
    parserClass := Refactory.Browser.RBParser.
    aBlock :=
            [self selectedClass
                ifNil: []
                ifNotNil:
                    [:aClass |
                    self selectedMethod
                        ifNil: []
                        ifNotNil:
                            [:aMethod |
                            aCode := (parserClass parseMethod: aCode) formattedCode asText.
                            self codeText value: aCode]]].
    aBlock on: Object errorSignal
        do: [:anException | ^Dialog warn: 'Format error: ' , anException messageText asString]
```

わざとコンパイルエラーになるようなものを指定してエラーを出して確認。

三段階目。
次は、ifNil になっているところ、つまり
1. クラスが選ばれていない状態
2. メソッドが選択されていない状態（definition が出ているはず）
でどうするか。まず前者(1)はおいて、後者(2)についてはメッセージの形をしているので、メッセージとしてパースして format してやればいいか。

```
    ifNil: [aCode := (parserClass parseExpression aCode) formattedCode asText]
```

とすれば良い。以下、当該部分に設置。（青文字部分）

```
    aBlock :=
            [self selectedClass
                ifNil: []
                ifNotNil:
                    [:aClass |
                    self selectedMethod
                        ifNil: [aCode := (parserClass parseExpression aCode) formattedCode asText]
```

```
                        ifNotNil:
                            [:aMethod |
                            aCode := (parserClass parseMethod: aCode) formattedCode asText.
                            Transcript
                                clear;
                                show: aCode asString].
                    self codeText value: aCode]].
```

なお全体の format ではなく、選択部分だけ format する、といった要求もあるだろう（workspace などに見られる）
この場合、selection だけ取り出すには aController selection で行う。（aController はこのメソッド自体の引数であった。用意の良い事よ！）
    Transcript cr; show: aController selection printString.
のようにして確認することが出来る。

さて前者(1)の ifNil についても同じく parseExpression で処理すればいいか？（青文字部分）

```
formatCodeText: aText from: aController
    | aCode parserClass aBlock |
    aText isEmpty ifTrue: [^nil].
    aCode := aText asString.
    parserClass := Refactory.Browser.RBParser.
    aBlock :=
            [self selectedClass
                ifNil: [aCode := (parserClass parseExpression: aCode) formattedCode asText]
                ifNotNil:
                    [:aClass |
                    self selectedMethod
                        ifNil: [aCode := (parserClass parseExpression: aCode) formattedCode asText]
                        ifNotNil:
                            [:aMethod |
                            aCode := (parserClass parseMethod: aCode) formattedCode asText.
                            Transcript
                                clear;
                                show: aCode asString].
                    self codeText value: aCode]].
    aBlock on: Object errorSignal
        do: [:anException | ^Dialog warn: 'Format error: ' , anException messageText asString]
```

四段階目。
今のままだと aCode := (parserClass parseExpression: aCode) formattedCode asText あたりがそっくりさん３連発なので、
これをまとめて処理することに。
今回たまたまセレクタが異なっていて、それ以外の書式については同一で済むので、そのあたりを狙ってまとめる。（青文字部分）

```
    aBlock :=
            [self selectedClass
                ifNil: [aSelector := #parseExpression:]
                ifNotNil:
                    [:aClass |
                    self selectedMethod
                        ifNil: [aSelector := #parseExpression:]
                        ifNotNil:
                            [:aMethod |
                            aSelector := #parseMethod:]].
            aCode := (parserClass perform: aSelector with: aCode) formattedCode asText.
            self codeText value: aCode].
```

およ。意外と読めるではないか。#つきでセレクタを目立つように書けるのが効いているのか？
はたまたプログラマが # がついたら（大抵）セレクタだ、と思い込んで読んでしまうのが良いのか。

五段階目。
ところで aClass も aMethod も format 側では要らない。（これは accept 側のものを引っ張ってきたため）
かつ、上のコードは単にセレクタを得ることしかしていないので、ちょっと位置関係を変えてもう少し整理する。（青文字）

```
formatCodeText: aText from: aController
    | aCode parserClass aBlock aSelector |
    aText isEmpty ifTrue: [^nil].
    aCode := aText asString.
    parserClass := Refactory.Browser.RBParser.
    aBlock :=
            [aSelector := self selectedClass
                        ifNil: [#parseExpression:]
                        ifNotNil:
                            [self selectedMethod ifNil: [#parseExpression:] ifNotNil: [#parseMethod:]].
            aCode := (parserClass perform: aSelector with: aCode) formattedCode asText.
            self codeText value: aCode].
    aBlock on: Object errorSignal
        do: [:anException | ^Dialog warn: 'Format error: ' , anException messageText asString]
```

構造的には、
　　self selectedClass
から始まる条件分岐（三つに分かれる）に対して、それぞれのブランチの実行箇所で aSlector := ... とやって、条件分岐を終えて収束した
ところで副作用的に aSelector を設定する、というスタイルから、
aSelector := に対して何をバインドすべきか、ということを三つの選択肢から選んで（値として答える）条件式、というスタイルへ変更したことになる。

これ、if 文のブランチが「値を返すもの」になっているのが効いてるんだなあ。
濱崎さんが言うように Smalltalk では結局すべて式（値をもつ）からなあ。
C だとブランチには確かに実行式が書かれているが、if 文そのものはそこで得たはずの実行式の値を捨ててしまうなあ。

以上、これらのコードの修正を自作のブラウザを立ち上げっぱなしの状態で、自分で修正して、自分で動作を確かめていた。
つまりプログラム（今起動している OldFashinedBrowser ）はディスクからメモリの中にコピーをされて、独自の（孤立した）空間の中で
動作しているのではなく、常に（唯一の、生きた）オブジェクトの空間で相互作用しながら動いているのだなあ。
冒頭にあった reflective / reflection のひとつの姿を見ることができるなあ。

＃それにしてもUnix 的なファイルからメモリへコピーし、それらはプロセスとして別空間で孤立して動作する、というモデルとは全く反対側だ。。。