Hamming's problem

- h is an "Ordered Set"
- $1 \in h$
- $x \in h \Rightarrow 2*x \in h, 3*x \in h, 5*x \in h.$
- generate all elements of $h < limit$

Let’s solve it

1. Write a test
2. make the test run green
3. clean up the code
   - ➡ remove any duplication
4. repeat until done

CS410/510 Advanced Programming
Lecture 7:

Regular Expressions in Smalltalk

Just Like Haskell

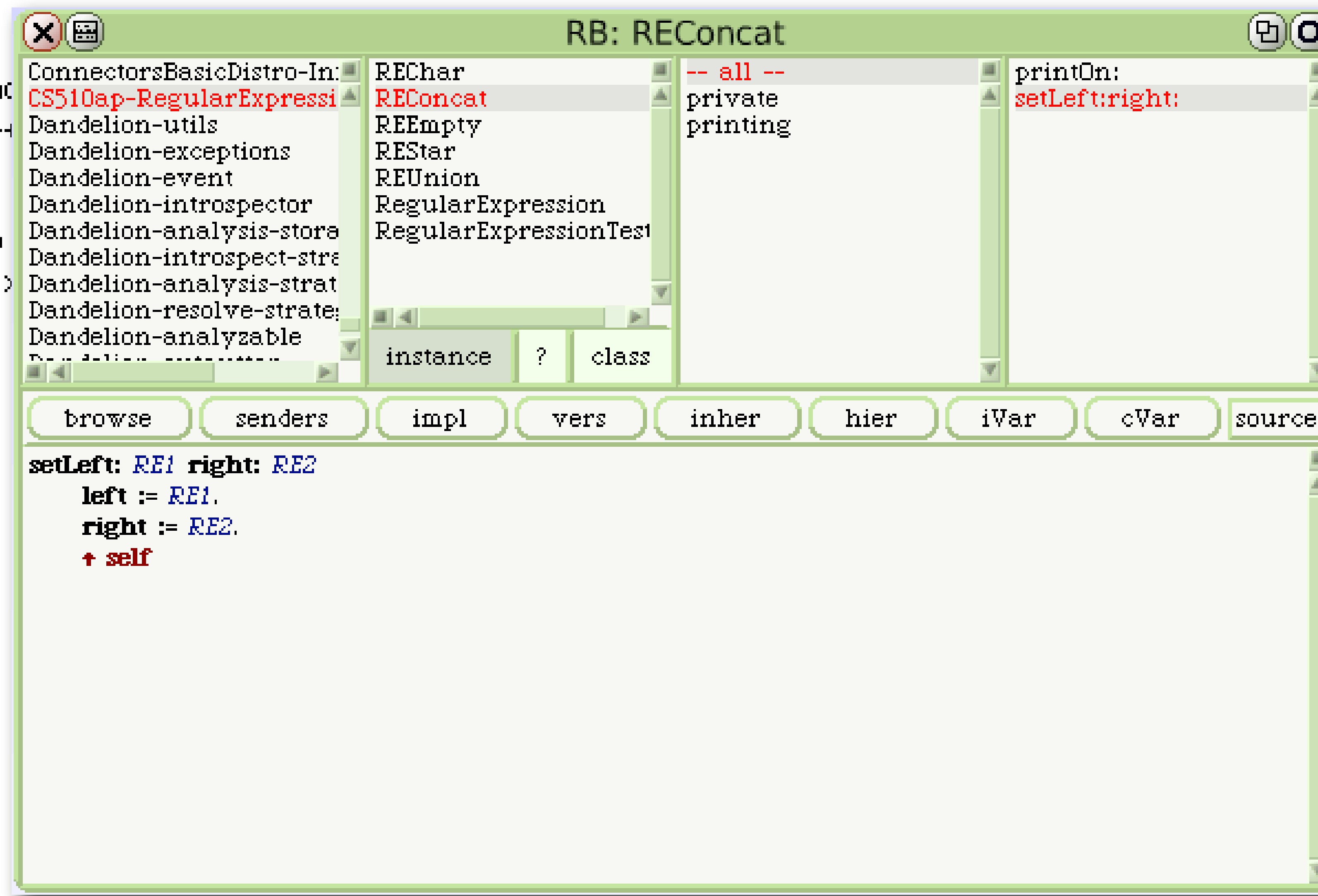
```
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "("++showU x++"+"++showU y++")"
    where showU (Union x y) = show x++"+"++showU y
          showU x = show x
  show (Concat x y) = show x++show y
  show (Star (x@(Concat _ _))) = "("++show x++")*"
  show (Star (x@(Union _ _))) = "("++show x++")*"
  show (Star x) = show x++"*"
```

Just Like Haskell

```
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "("++showU x++"+"++sho
    where showU (Union x y) = show x++"+"++
          showU x = show x
  show (Concat x y) = show x++show y
  show (Star (x@(Concat _ _))) = "("++show
  show (Star (x@(Union _ _))) = "("++show x
  show (Star x) = show x++"*"
```

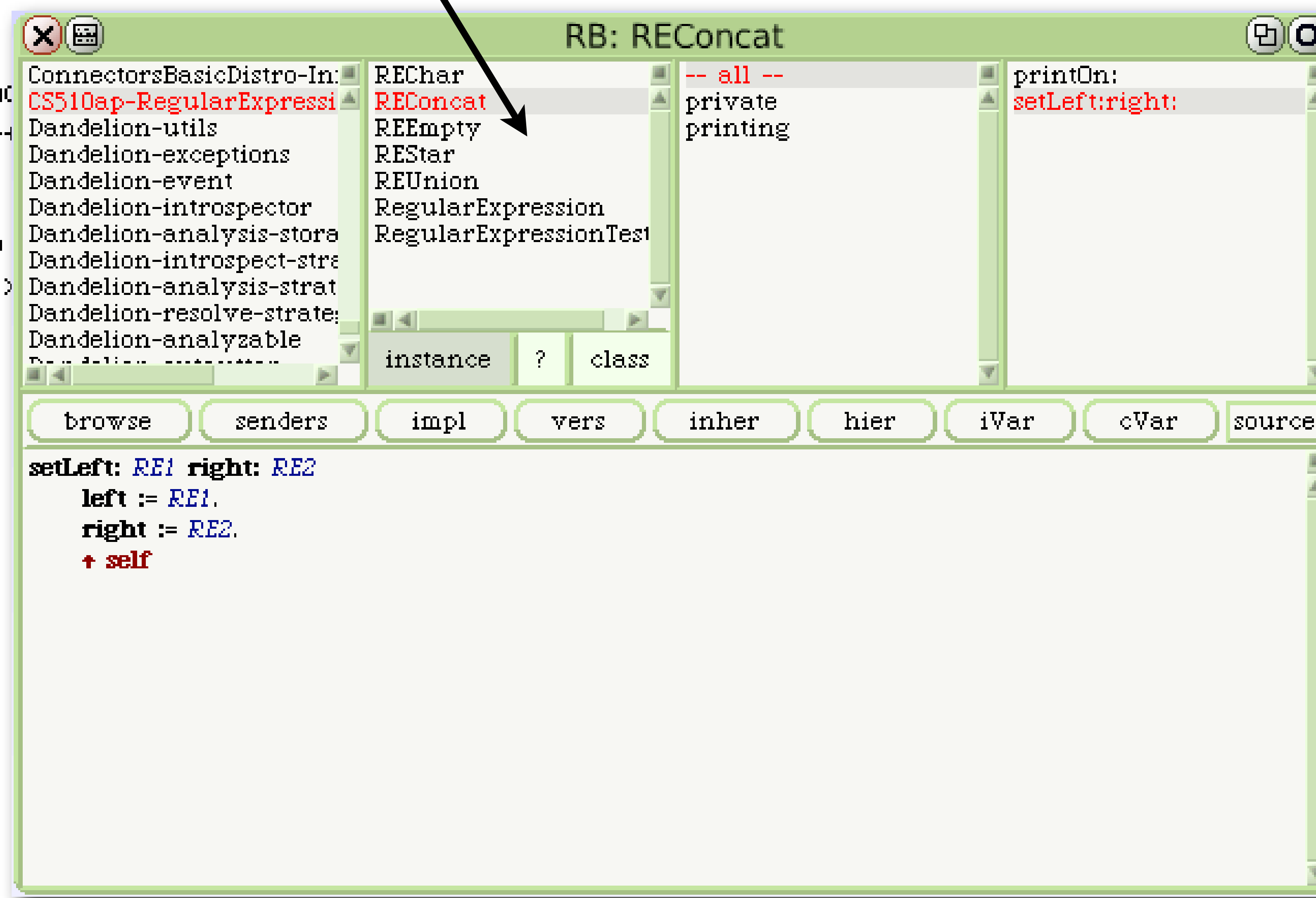


Just Like Haskell

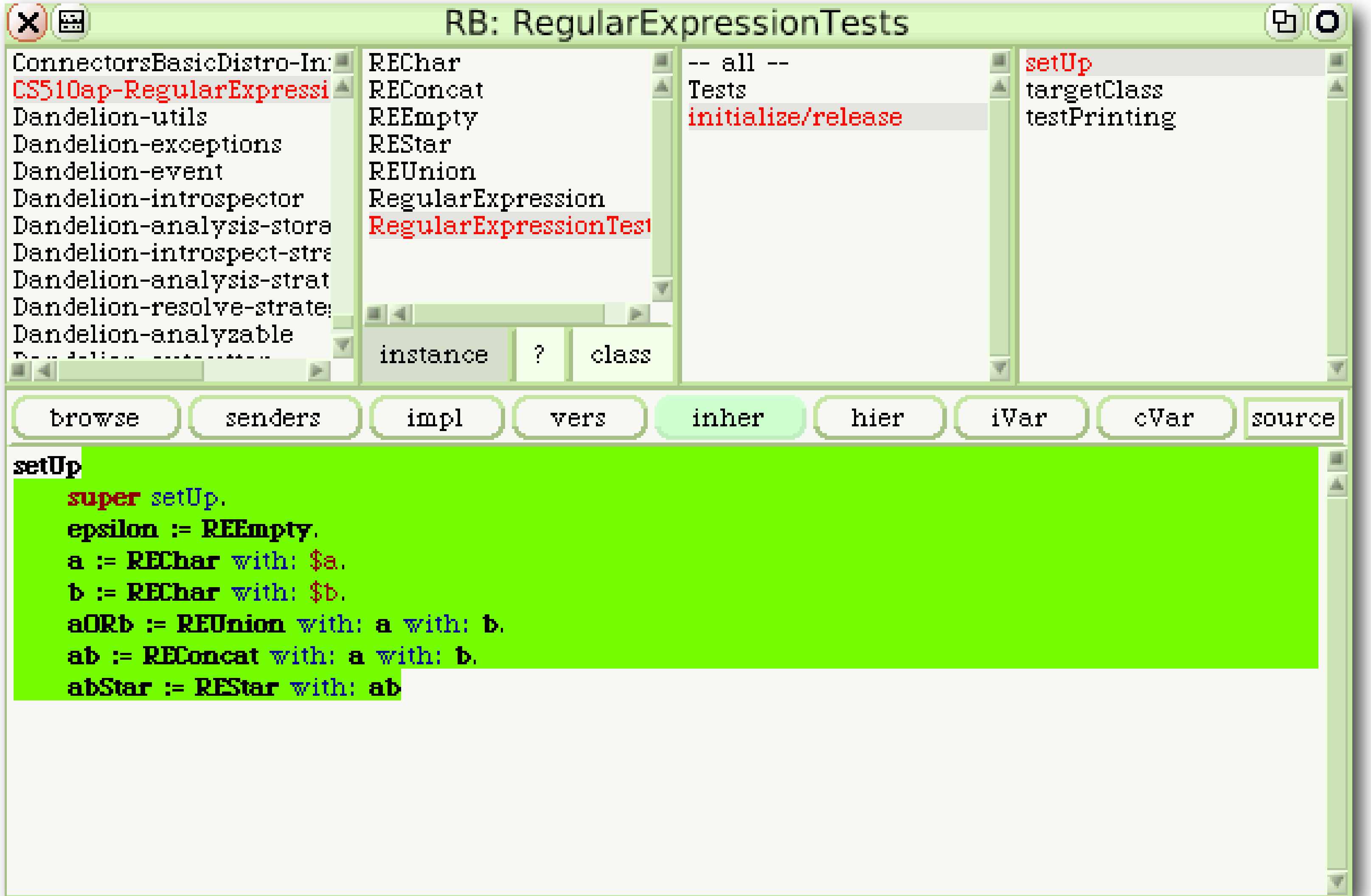
```
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "("++showU x++"+"++sho
    where showU (Union x y) = show x++"+"++
          showU x = show x
  show (Concat x y) = show x++show y
  show (Star (x@(Concat _ _))) = "("++show
  show (Star (x@(Union _ _))) = "("++show x
  show (Star x) = show x++"*"
```

One subclass for each alternative representation

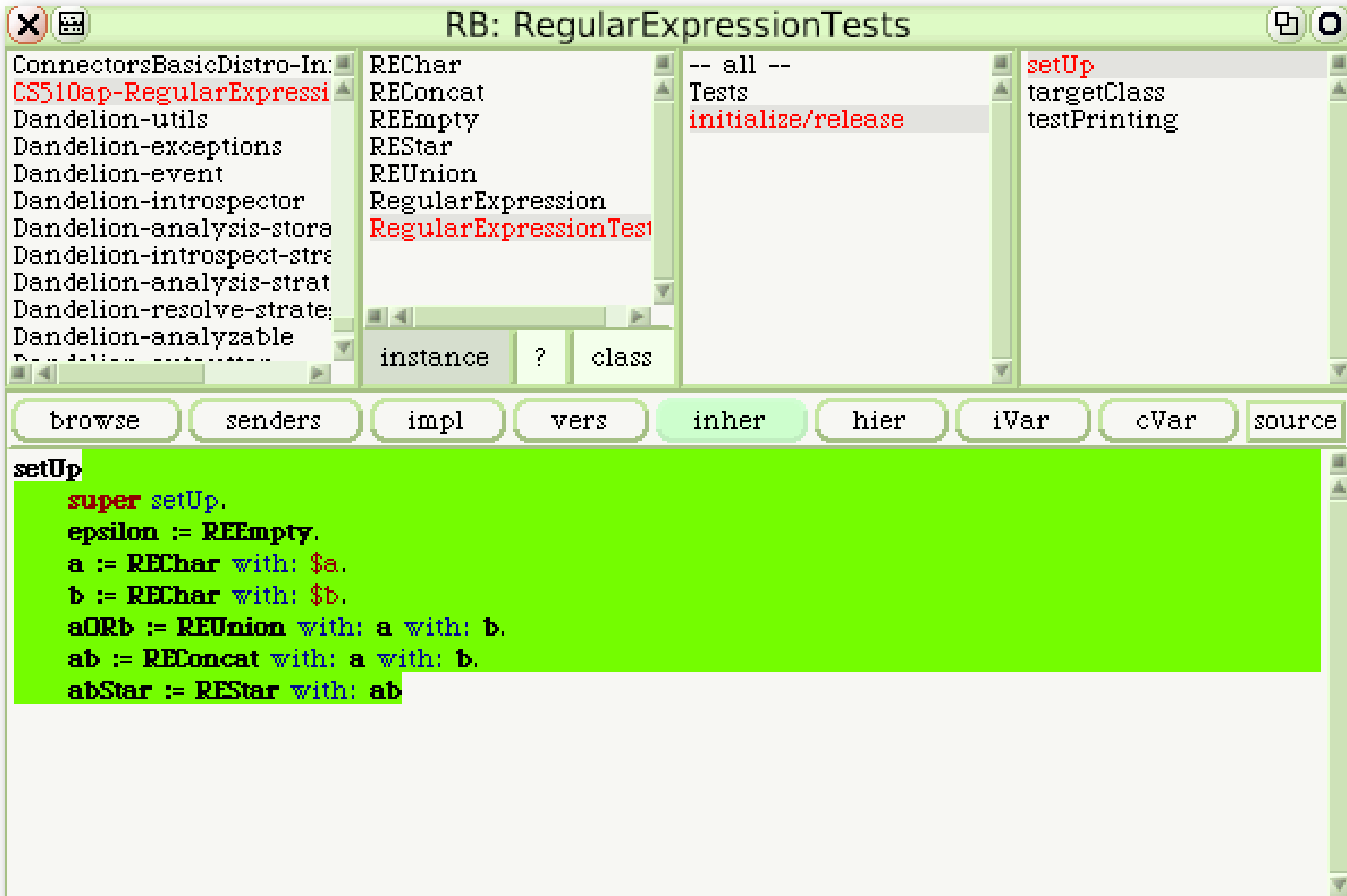


Write Tests



PORTLAND STATE UNIVERSITY

Write Tests



1. Run tests

2. get *message not understood*

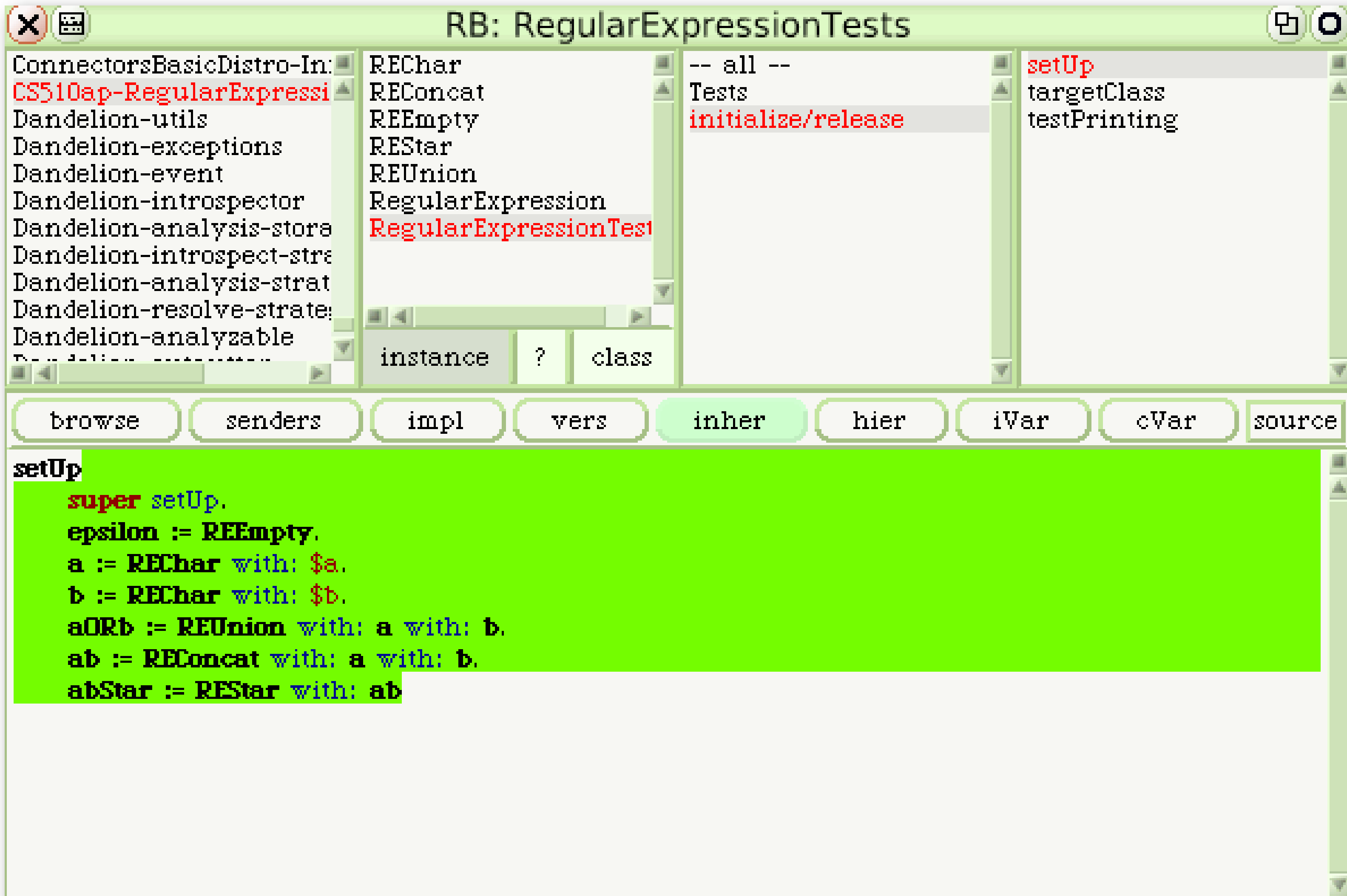
3. define method

4. repeat from 1

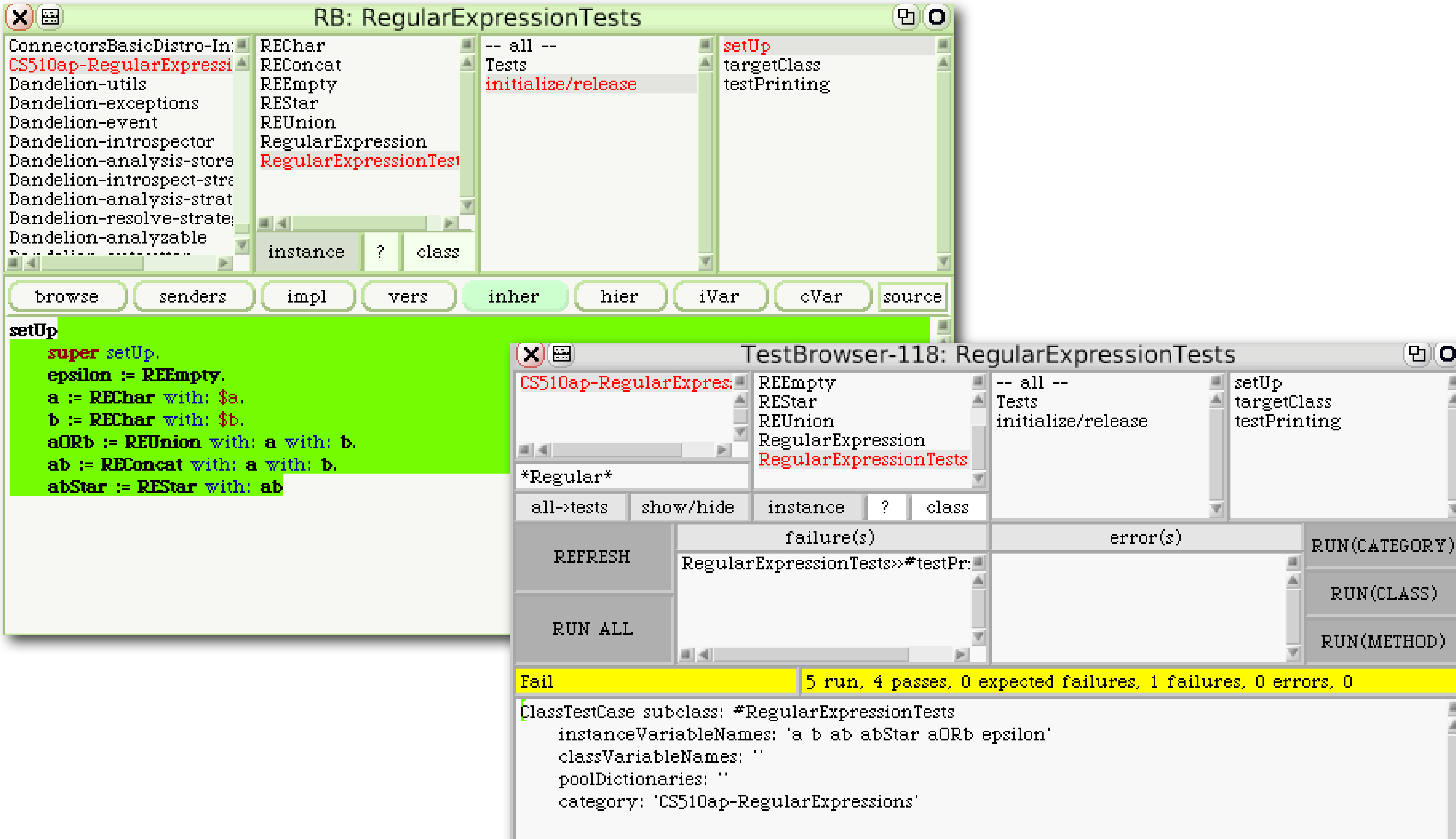
…

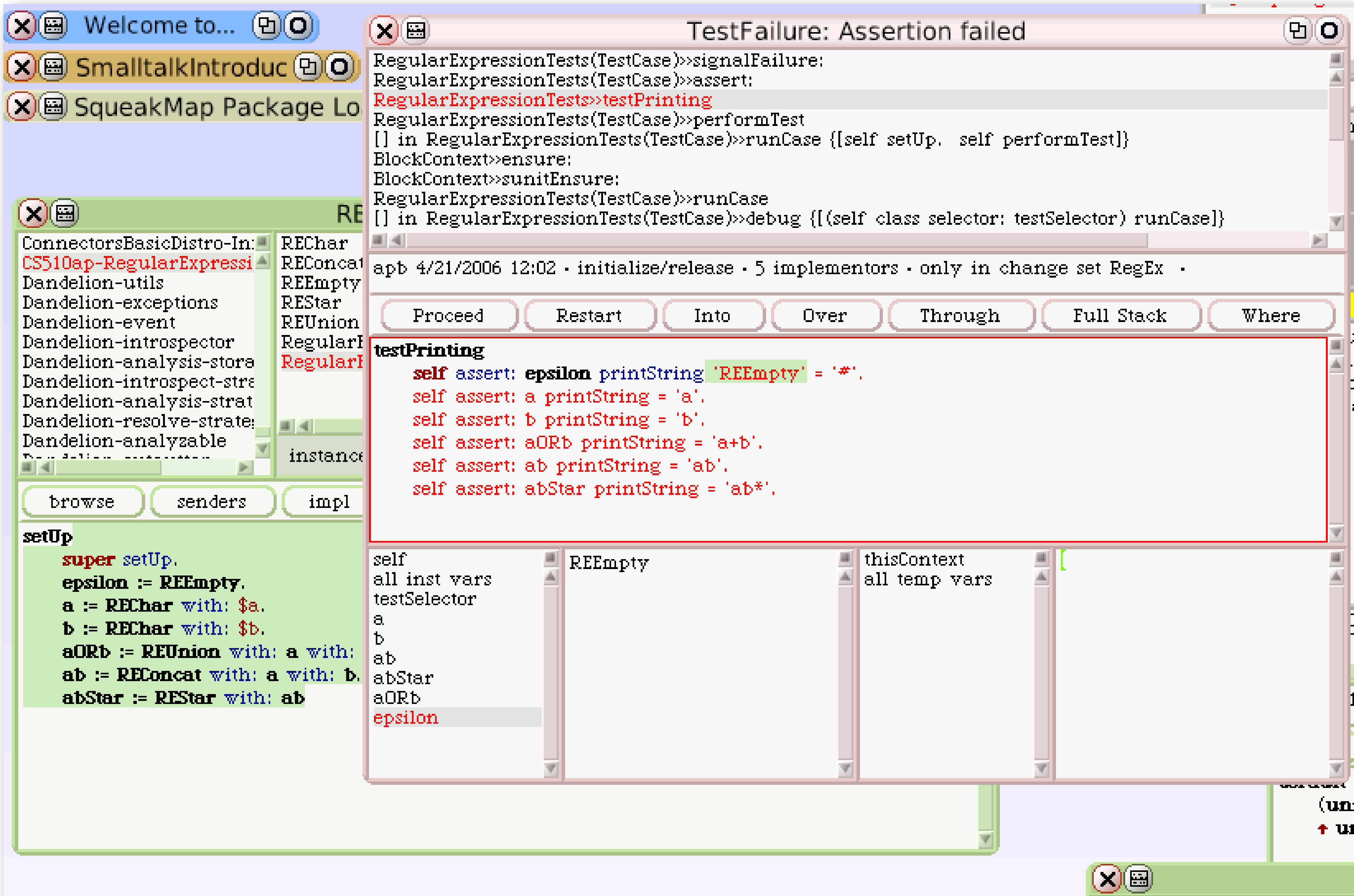
19. get real failure

Write Tests


RB: RegularExpressionTests
ConnectorsBasicDistro-In:
CS510ap-RegularExpressi
Dandelion-utils
Dandelion-exceptions
Dandelion-event
Dandelion-introspector
Dandelion-analysis-store
Dandelion-introspect-stra
Dandelion-analysis-strat
Dandelion-resolve-strate
Dandelion-analyzable
REChar
REConcat
REEmpty
REStar
REUnion
RegularExpression
RegularExpressionTest
instance
?
class
-- all --
Tests
initialize/release
setUp
targetClass
testPrinting
browse
senders
impl
vers
inher
hier
iVar
cVar
source
setUp
super setUp.
epsilon := REEmpty.
a := REChar with: $a.
b := REChar with: $b.
aORb := REUnion with: a with: b.
ab := REConcat with: a with: b.
abStar := REStar with: ab

Write Tests


RB: RegularExpressionTests
ConnectorsBasicDistro-In:
CS510ap-RegularExpressi
Dandelion-utils
Dandelion-exceptions
Dandelion-event
Dandelion-introspector
Dandelion-analysis-store
Dandelion-introspect-stra
Dandelion-analysis-strat
Dandelion-resolve-strate;
Dandelion-analyzable
REChar
REConcat
REEmpty
REStar
REUnion
RegularExpression
RegularExpressionTest
instance
?
class
-- all --
Tests
initialize/release
setUp
targetClass
testPrinting
browse
senders
impl
vers
inher
hier
iVar
cVar
source
setUp
super setUp.
epsilon := REEmpty.
a := REChar with: $a.
b := REChar with: $b.
aORb := REUnion with: a with: b.
ab := REConcat with: a with: b.
abStar := REStar with: ab
TestBrowser-118: RegularExpressionTests
CS510ap-RegularExpres:
REEmpty
REStar
REUnion
RegularExpression
RegularExpressionTests
-- all --
Tests
initialize/release
setUp
targetClass
testPrinting
Regular
all->tests
show/hide
instance
?
class
REFRESH
RUN ALL
failure(s)
RegularExpressionTests>>#testPr:
error(s)
RUN(CATEGORY)
RUN(CLASS)
RUN(METHOD)
Fail
5 run, 4 passes, 0 expected failures, 1 failures, 0 errors, 0
ClassTestCase subclass: #RegularExpressionTests
instanceVariableNames: 'a b ab abStar aORb epsilon'
classVariableNames: ''
poolDictionaries: ''
category: 'CS510ap-RegularExpressions'



What's the problem?

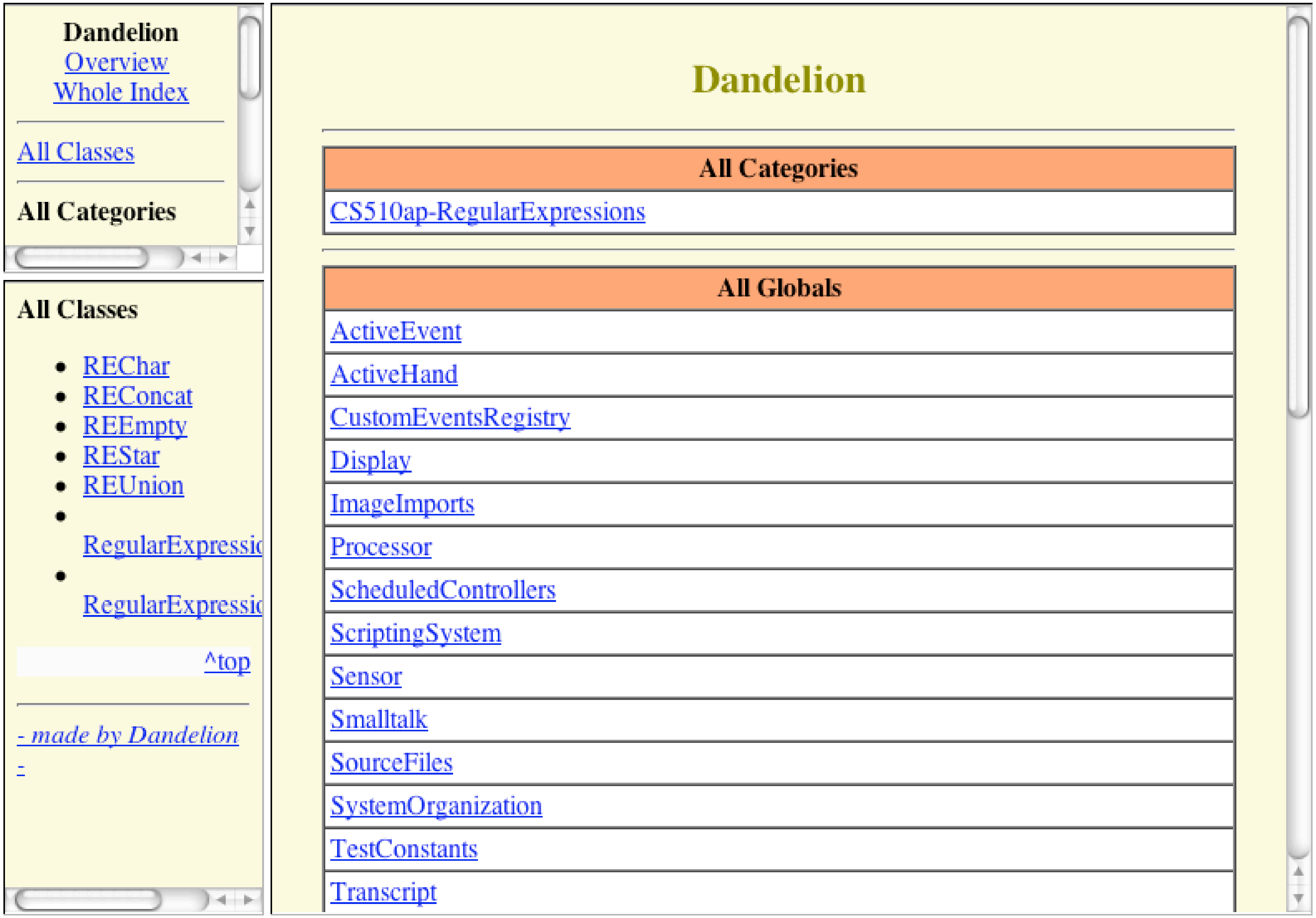
I need an instance, not the class

- But there need be only one instance of REEmpty
- Enter: the Singleton pattern.
  - make a class instance-variable called uniqueInstance
  - make a class-side method named default

```
default
    uniqueInstance ifNil: [uniqueInstance := self basicNew].
    ↑ uniqueInstance
```

  - override new to be an error

What do we have so far?



Convenience Operations

```
alpha = Union (C 'a')
              (Union (C 'b') (C 'c'))
digit = Union (C '0')
              (Union (C '1') (C '2'))
key = Union (string "if")
            (Union (string "then")
            (string "else"))
punc = (C ',')
ident = Concat alpha
              (Star (Union alpha digit))
number = Concat digit (Star digit)
lexer = Union ident (Union number (Union key punc))

val re1 = Concat(Union (C '+')(Union (C '-')Empty))
                (Concat (C 'D')(Star (C 'D')))

string :: String -> RE
string [] = Empty
string [c] = C c
string (c:cs) = Concat (C c) (string cs)
```

- Write tests:

  self assert: $a asRE printString = 'a'

  self assert: (a + b) printString = 'a+b'

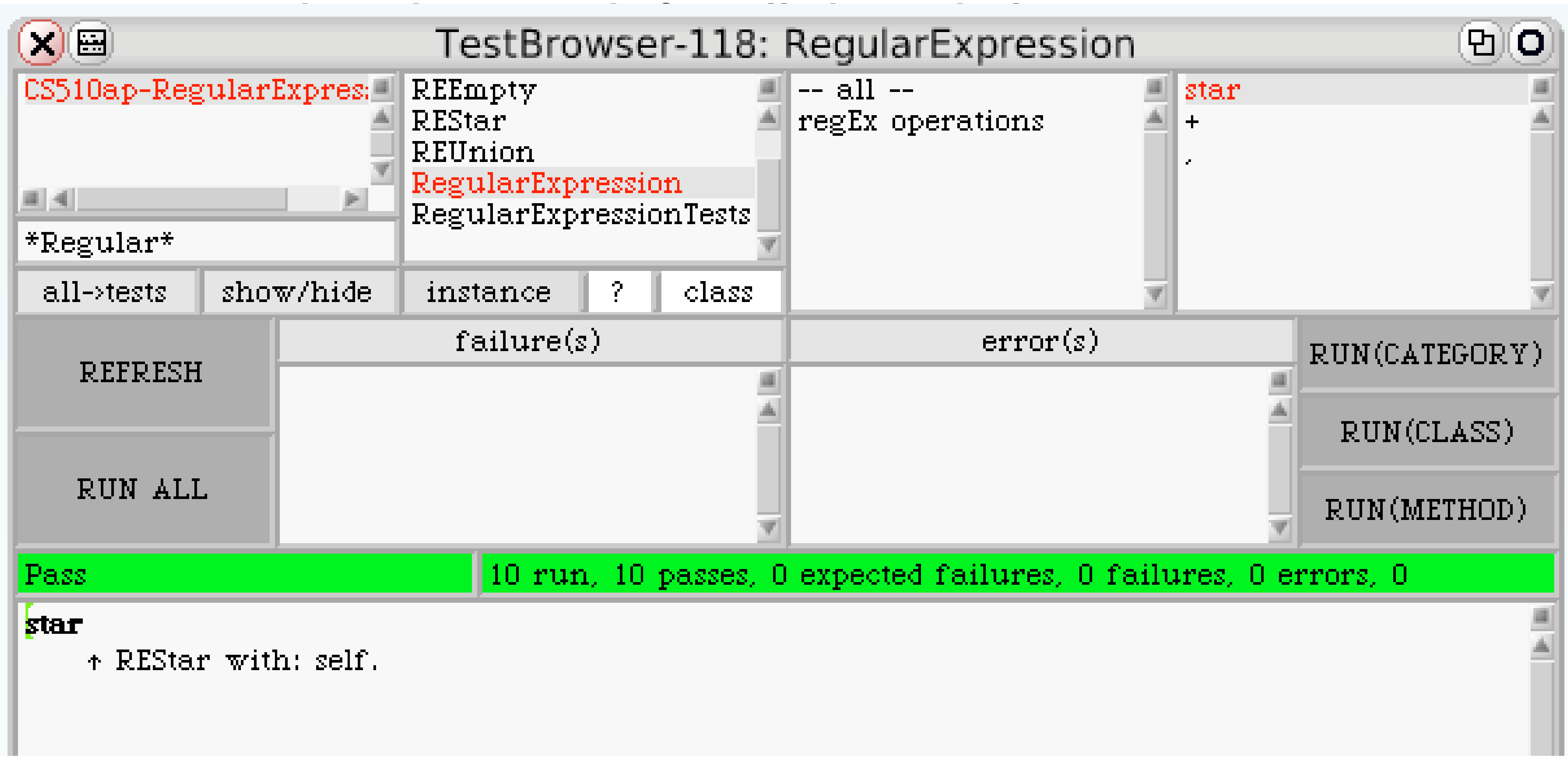
- Why compare *printStrings*?

Where do the operation methods go?

- In the abstract superclass RegularExpression
  - so that they work for all the subclasses

Where do the operation methods go?

- In the abstract superclass RegularExpression

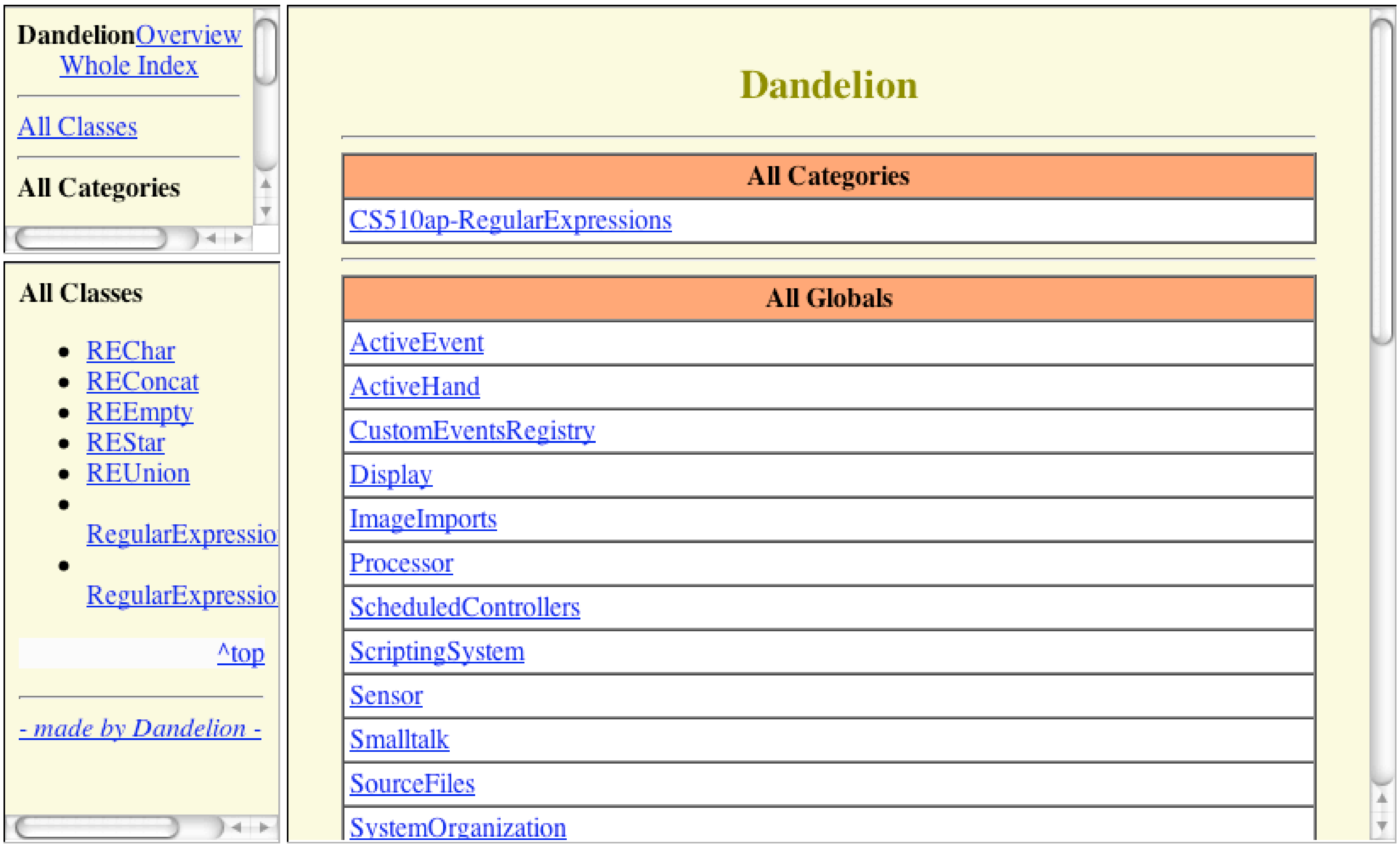


Refactor *tests* to remove duplication

```
testPrinting
    self assert: epsilon printsAs: '#'.
    self assert: a printsAs: 'a'.
    self assert: b printsAs: 'b'.
    self assert: aORb printsAs: 'a+b'.
    self assert: ab printsAs: 'ab'.
    self assert: abStar printsAs: 'ab*'.
```

```
assert: anExpression printsAs: aprintString
    self assert: anExpression printString = aprintString
```

which brings us to…



meaning1: sets of strings

- Code very similar to Tim's Haskell version
- Only tricky part is star
  - Haskell version:

```
meaning1 (Star r) = norm(zero ++ one ++ two ++ three)
  where zero = [""]
        one = meaning1 r
        two = [x++y | x <- one, y <- one]
        three = [x++y | x <- one, y <- two]
```

Smalltalk

RESTar

```
meaning1
	| zero one two three |
	zero := ''.
	one := base meaning1.
	two := self anyOf: one followedByAnyOf: one.
	three := self anyOf: one followedByAnyOf: two.
	↑ (Set with: zero) addAll: one;
		addAll: two;
		addAll: three;
		yourself
```

RegularExpression

```
anyOf: ml followedByAnyOf: mr
	| result |
	result := Set new.
	ml do: [:l | mr do: [:r | result add: l , r]].
	↑result
```

- Complicated enough to need a helper method
- Is there a simpler way to calculate * ?

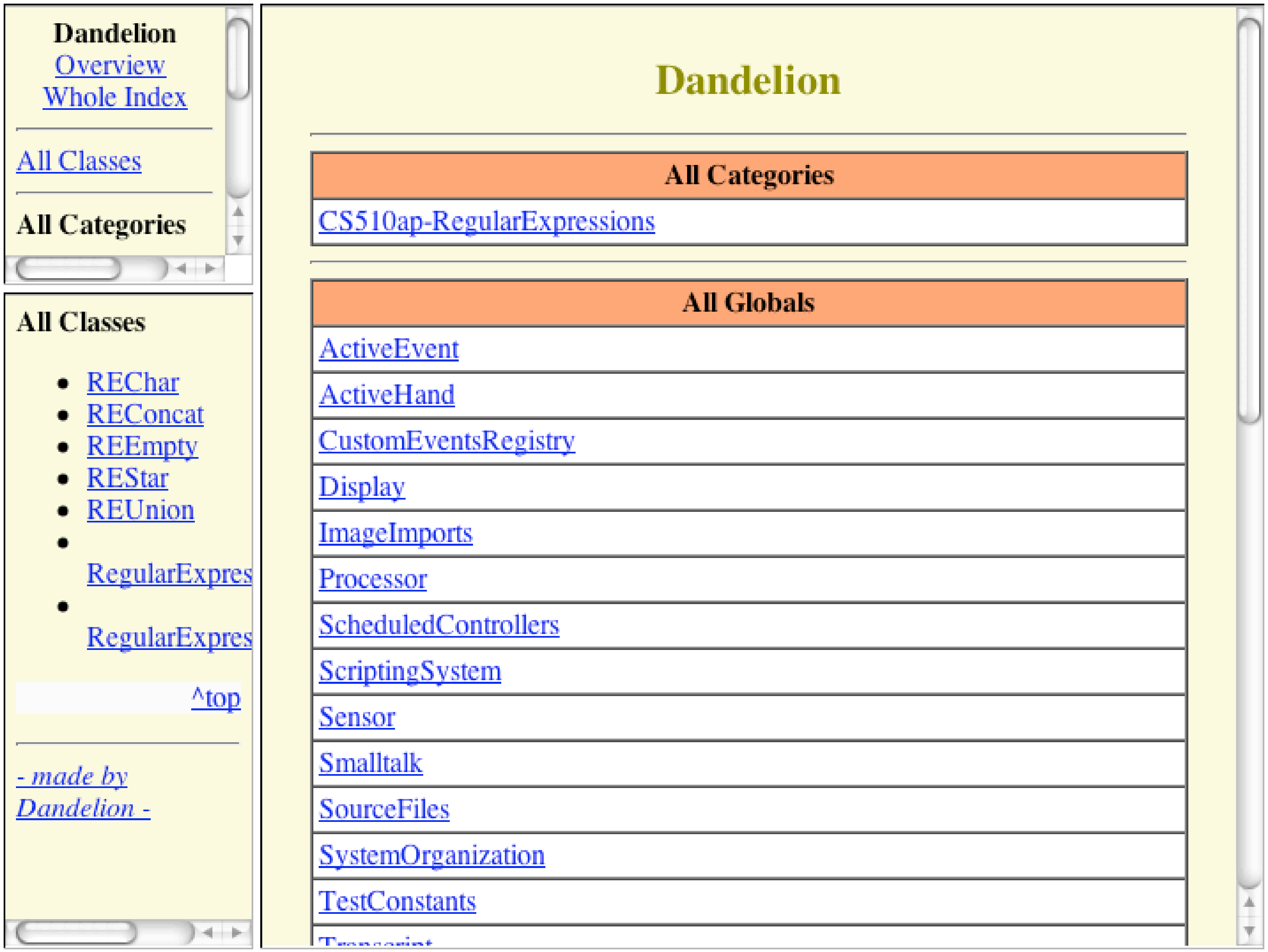
Cross tests

```
Pass    17 run, 17 passes, 0 expected failures, 0 failures, 0 errors

testMeaning1AgainstMeaning2
    self instanceVariableValues select: [ :each | each respondsTo: #meaning1 ] thenDo:
        [ :re | re meaning1 do: [ :str | self assert: (re meaning2: str) ] ]
```

- introspect on the instance variables of the test case
  - select those that respond to the meaning1 message
  - check that for every string str in re meaning1
    - re meaning2: str is true

Now RE's pass the tests



Finite State Machines

FINITE AUTOMATA AND REGULAR GRAMMARS

3.1 THE FINITE AUTOMATON

In Chapter 2, we were introduced to a generating scheme—the grammar. Grammars are finite specifications for languages. In this chapter we shall see another method of finitely specifying infinite languages—the recognizer. We shall consider what is undoubtedly the simplest recognizer, called a finite automaton. The finite automaton (fa) cannot define all languages defined by grammars, but we shall show that the languages defined are exactly the type 3 languages. In later chapters, the reader will be introduced to recognizers for type 0, 1, and 2 languages. Here we shall define a finite automaton as a formal system, then give the physical meaning of the definition.

A *finite automaton* M over an alphabet Σ is a system $(K, \Sigma, \delta, q_0, F)$, where K is a finite, nonempty set of *states*, Σ is a finite *input alphabet*, δ is a mapping of $K \times \Sigma$ into K, q_0 in K is the *initial state*, and $F \subseteq K$ is the set of *final states*.

Our model in Fig. 3.1 represents a finite control which reads symbols from a linear input tape in a sequential manner from left to right. The set of states K consists of the states of the finite control. Initially, the finite control is in state q_0 and is scanning the leftmost symbol of a string of symbols in Σ which appear on the input tape. The interpretation of $\delta(q, a) = p$, for q

The code with NFSM

Dandelion
Overview
Whole Index

All Classes

All Categories

All Classes

- FSMState
- FSMStateSet
- NFSM
- NFSMTests
- REChar
- REConcat
- REEmpty
- REStar
- REUnion
- RegularExpressi
- RegularExpressi

^top

- made by Dandelion -

Dandelion

All Categories
CS510ap-RegularExpressions

All Globals
ActiveEvent
ActiveHand
CustomEventsRegistry
Display
ImageImports
Processor
ScheduledControllers
ScriptingSystem
Sensor
Smalltalk
SourceFiles
SystemOrganization
TestConstants
Transcript
Undeclared