UNIVERSITÄT
BERN

6. Debugging

Birds-eye view

It can be easier to talk to objects than to read classes

— The system is alive. Talk to it.

The debugger can be your best friend. Don't be afraid of it.

Roadmap

> Common syntactic errors
> Common semantic errors
> Encapsulation errors
> Class/instance errors
> Debugging patterns

Selected material based on Klimas, et al., *Smalltalk with Style.*
Selected material courtesy Stéphane Ducasse.

Roadmap

> **Common syntactic errors**
> Common semantic errors
> Encapsulation errors
> Class/instance errors
> Debugging patterns

Does not understand self

> The error message “does not understand self” usually means that you have forgotten the period at the end of a statement

```
SnakesAndLaddersTest>>testExample
   self assert: eg currentPlayer = jack.
   loadedDie roll: 1.
   eg playOneMove
   self assert: jack position = 6.
   self assert: eg currentPlayer = jill.
```

Klimas, et al., *Smalltalk with Style*

Use parentheses in expressions with multiple keyword messages

> Do not forget to use parentheses when sending multiple keyword messages in one expression

```
self assert: players includes: aPlayer.
```

```
self assert: (players includes: aPlayer).
```

Klimas, et al., *Smalltalk with Style*

True vs true

> `true` is the boolean value, `True` its class.

```
Book>>initialize
  inLibrary := True
```

```
Book>>initialize
  inLibrary := true
```

nil is not a Boolean

> `nil` is not an acceptable receiver for ifTrue:

whileTrue

> The receiver of `whileTrue:` and `whileTrue` must be a *block*

```
(x<y) whileTrue: [x := x + 3]
```

```
[x<y] whileTrue: [x := x + 3]
```

Commenting comments

> Be careful when commenting out code that contains comments
  — You may activate some other code that was commented out!

```
MyClass>>doit
    self doStuff.
    self doMoreStuff.
    "self suicide."
    self finishUp.
```

```
MyClass>>doit
    self doStuff.
"
    self doMoreStuff.
    "self suicide."
    self finishUp.
"
```

Forgetting to return the result

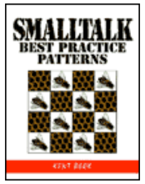
> In a method `self` is returned by default.
  — Do not forget ^ to return something else!

```
BoardSquare>>isLastSquare
   position = board lastPosition
```

Returns self (a Board`Square`), not a `Boolean`!

Interesting Return Value

When do you explicitly return a value at the end of a method?



- Return a value only when you intend for the sender to use the value.
  - Return self explicitly only if the client is expected to use it!

```
BoardSquare>>destination
    ^ self
```

Even though self is returned by default we make explicit that this is the value returned.

Method arguments are read-only

> Do not try to assign a value to a method argument.
  — Arguments are read only!

```
MyClass>>setName: aString
   aString := aString, 'Device'.
   name := aString
```

Won't compile!

self and super are read-only

> Do not try to modify `self` or `super`

Roadmap

> Common syntactic errors
> **Common semantic errors**
> Encapsulation errors
> Class/instance errors
> Debugging patterns

Do not override basic methods

> Never redefine basic-methods
>
> — `==`, `basicNew`, `basicNew:`, `basicAt:`, `basicAt:Put:` ...

> Never redefine the method `class`

hash and =

- Redefine hash whenever you redefine =
  - Ensure that if `a = b` then `a hash = b hash`
  - Otherwise Sets and Dictionaries may behave incorrectly!

```
Book>>=aBook
   ^ (self title = aBook title)
     and: [self author = aBook author]

Book>>hash
   ^self title hash bitXor: self author hash
```

add: returns the argument

> `add:` returns the argument and not the receiver
  — Use `yourself` to get the collection back.

```
OrderedCollection new add: 5; add: 6
```

```
6
```

```
OrderedCollection new add: 5; add: 6; yourself
```

```
an OrderedCollection(5 6)
```

Don’t iterate over a collection and modify it

> Never iterate over a collection which the iteration somehow modifies.

```
[:range | range do: [:aNumber | aNumber isPrime
          ifFalse: [ range remove: aNumber ]]. range
] value: ((2 to: 20) asOrderedCollection)
```

```
an OrderedCollection(2 3 5 7 9 11 13 15 17 19)
```

First *copy* the collection

```
[:range | range copy do: [:aNumber | aNumber isPrime
          ifFalse: [ range remove: aNumber ]]. range
] value: ((2 to: 20) asOrderedCollection)
```

```
an OrderedCollection(2 3 5 7 11 13 17 19)
```

Take care, since the iteration can involve various methods and modifications which may not be obvious!

Roadmap

> Common syntactic errors
> Common semantic errors
> **Encapsulation errors**
> Class/instance errors
> Debugging patterns

Use of Accessors: Protect your Clients

> The literature says:
  - "Access instance variables using methods"
    - *I.e., getters and setters*

```
SnakesAndLadders>>initialize
    …
    self squares: OrderedCollection new.
    …

SnakesAndLadders>>squares
    ^ squares
```

> However, accessor methods should be *private* by default.
  - Put them in the *private* protocol

> A client could use a public accessor to modify our state
  - If we change the representation of squares, client code could break!
  - Instead provide *dedicated methods* to modify private state

Copy a collection if you do not want it modified

- Answer a copy of a collection if you do not want it modified
  - Law of Demeter: never modify a returned collection!

```
SnakesAndLadders>>squares
    ^ squares
```

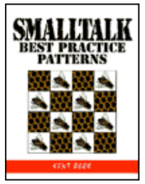
```
NastyClient>>break: aSnakesAndLadders
    aSnakesAndLadders squares removeFirst
```

```
SnakesAndLadders>>squares
    ^ squares copy
```

Klimas, et al., *Smalltalk with Style*

Collection Accessor method

How do you provide access to an instance variable that holds a collection?

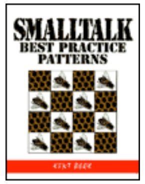


> Provide methods that are implemented with delegation to the collection.
  — To name the methods, (possibly) add the name of the collection to the collection messages

```
SnakesAndLadders>>at: position
   ^ squares at: position

SnakesAndLadders>>currentPlayer
   ^ players at: turn
```

Enumeration Method

How do you provide safe, general access to collection elements?

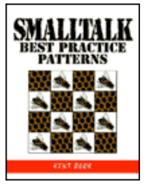


> Implement a method that executes a Block for each element of the collection
  - Name the method by concatenating the name of the collection and Do:

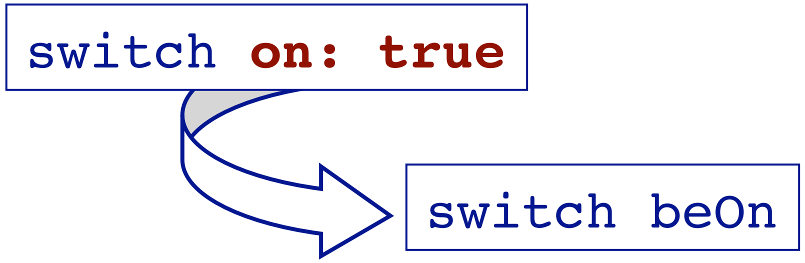
```
SnakesAndLadders>>squaresDo: aBlock
   squares do: aBlock

SnakesAndLadders>>playersDo: aBlock
   players do: aBlock
```

Boolean Property Setting Method

How do you set a boolean property?



> Create two methods beginning with "be".
  - One has the property name, the other the negation.
  - Add "toggle" if the client doesn't want to know about the current state.



```
switch on: true
```

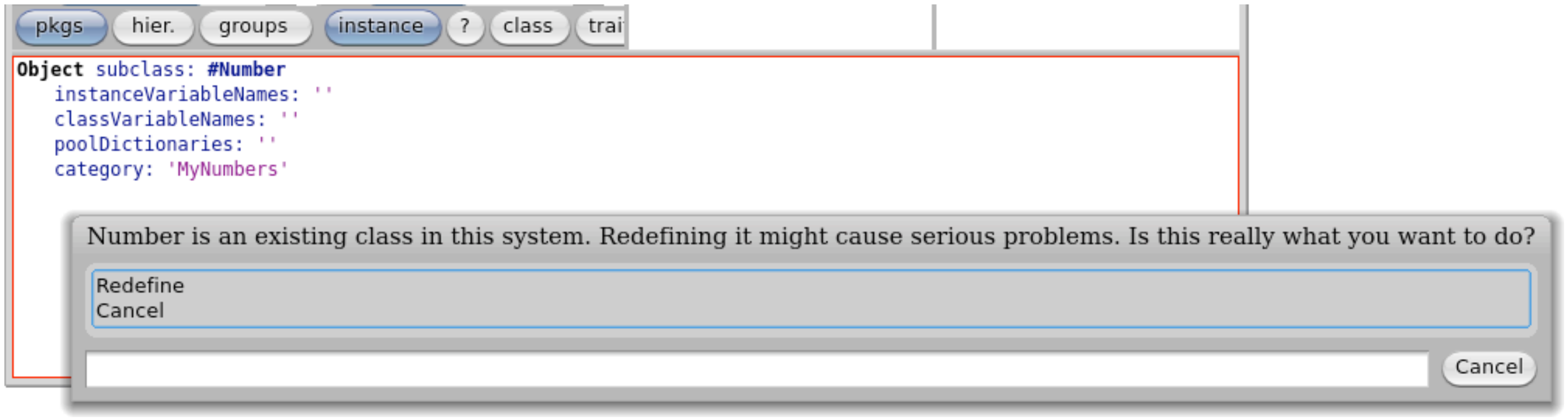
```
switch beOn
```

Roadmap

> Common syntactic errors
> Common semantic errors
> Encapsulation errors
> **Class/instance errors**
> Debugging patterns

(Re-)Defining classes

> Redefining a class:
  - Before creating a class, check if it already exists. This is (sigh) a weakness of the system
  - VisualWorks 7.0 has namespaces so less likely to redefine a class



```
Object subclass: #Number
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyNumbers'
```

Number is an existing class in this system. Redefining it might cause serious problems. Is this really what you want to do?

Redefine
Cancel

Pharo checks this for critical classes.

Class methods cannot access instance variables

> Do not try to access instance variables to initialize them in a class method.
  - It is impossible!
  - A class method can only access class instance variables and classVariables.
    - *Define and invoke an `initialize` method on instances.*
    - *Or define a Constructor Parameter Method*

```
SnakesAndLadders>>initialize
   …
   die := Die new.
   squares := …
```

```
GamePlayer class>>named: aName
   ^ self new setName: aName
```

Do not reference class names

> Do not explicitly reference the class name to create new instances of the receiver
  - *This will break subclassing*
  - Reference `self` instead

```
Object subclass: #VeebleFetzer
   instanceVariableNames: 'name'
   …

VeebleFetzer>>name: aName
   name := aName

VeebleFetzer class>>named: aName
   ^ VeebleFetzer new name: aName
```

```
VeebleFetzer named: 'mine'
```

```
a VeebleFetzer
```

```
VeebleFetzer subclass: #FeebleVetzer
   instanceVariableNames: ''
   …
```

```
FeebleVetzer named: 'mineToo'
```

```
a VeebleFetzer
```

Klimas, et al., *Smalltalk with Style*

Returning the class instead of an instance

```
MyClass>>new
    super new initialize
```

```
MyClass>>new
    ^ super new initialize
```

Returns the class `MyClass` (`self`) and not the new instance!

Looping initialization

```
Packet class>> new
   ^self new initialize
```

This example loops!

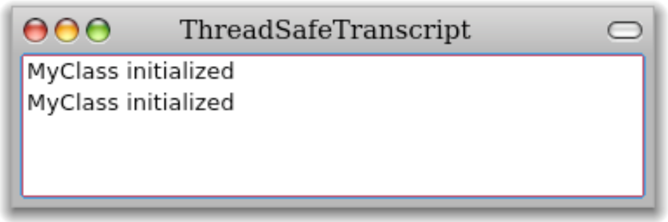
In Pharo, new objects are initialized by default!

```
Behavior>>new
   ^ self basicNew initialize
```

Super new initialize

> `super new initialize` is usually redundant
  - In Pharo, this is done automatically (in Behavior)
  - Your objects will be initialized twice!

```
Object subclass: #MyClass
   …

MyClass>>initialize
   Transcript show: self class name;
      show: ' initialized'; cr.

MyClass class>>new
   ^ super new initialize
```



```
MyClass initialized
MyClass initialized
```

Super initialize

> Don’t forget to initialize any inherited state!

```
MyClass>>initialize
    super initialize.
    …
```

Establish super invariants before establishing own invariant (as in Java)

Roadmap

> Common syntactic errors
> Common semantic errors
> Encapsulation errors
> Class/instance errors
> **Debugging patterns**

Debug printing

> Basic printing
>
> — You can use the Transcript to display progress

```
Transcript cr; show: ‘The total= ’, self total printString.
```

> Optional printing
>
> — Use a global or a class to control printing information

```
Debug
   ifTrue: [Transcript show: self total printString]
```

```
Debug > 4
   ifTrue: [Transcript show: self total printString]
```

```
Debug print: [Transcript show: self total printString]
```

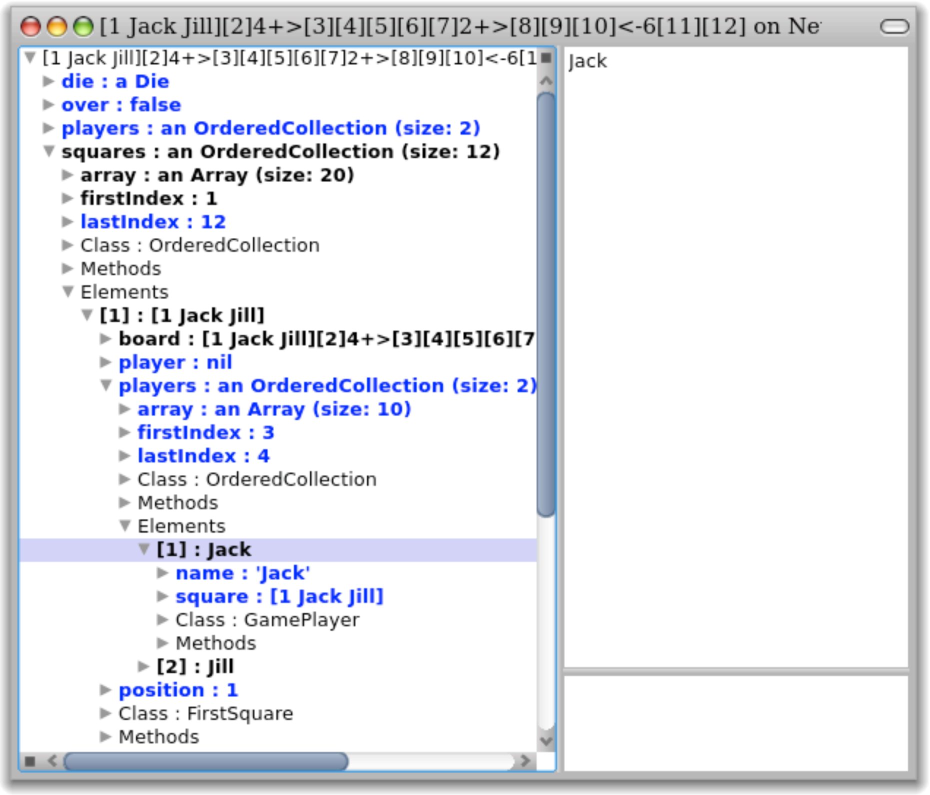
Tests are your friends!

- Resist the temptation to write debugging print methods
  - Write a test instead!
- Resist the temptation to evaluate ad hoc expressions in a Workspace
  - Write a test instead!

  - ***Tests are reusable***
    - *You will have to spend the effort debugging anyway*
    - *Amortize the investment by coding your debugging effort as tests*

The Inspector is your friend!

> You can inspect anything
  - Inspect any expression
  - View the `printString` state
  - Interact with any object
  - Inspect instance variables
  - Navigate through the system

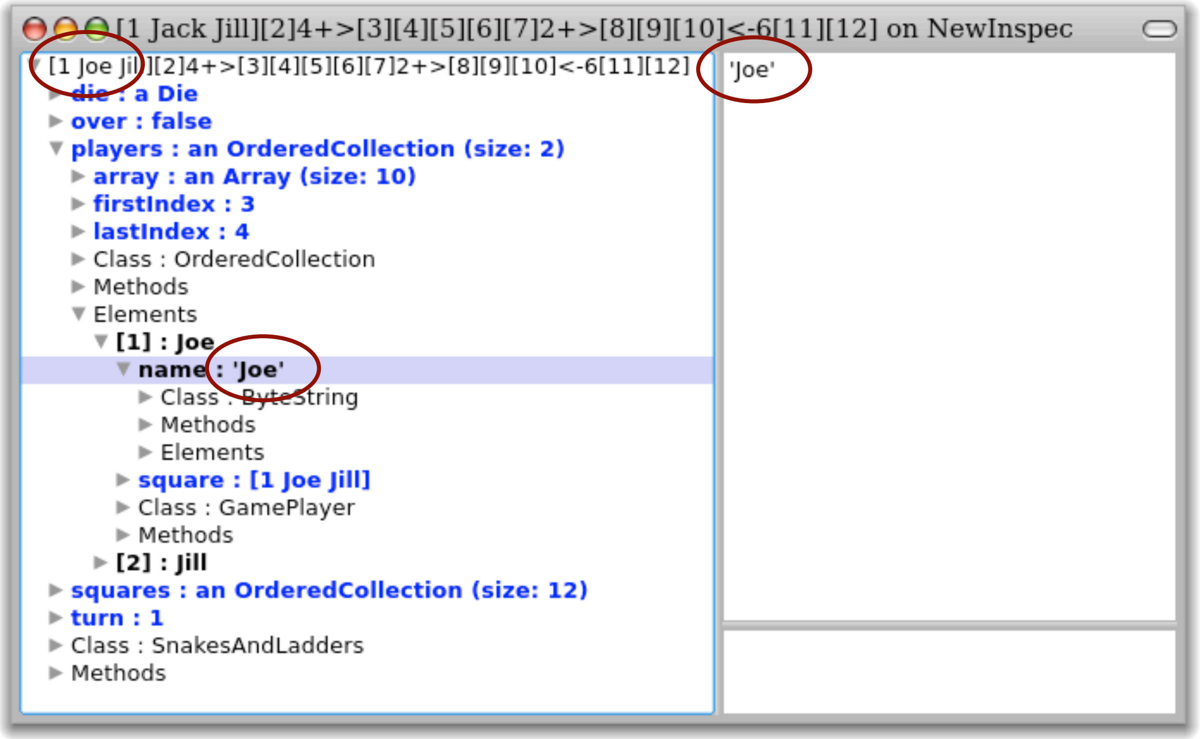


```
[1 Jack Jill][2]4+>[3][4][5][6][7]2+>[8][9][10]<-6[11][12] on Ne
▼ [1 Jack Jill][2]4+>[3][4][5][6][7]2+>[8][9][10]<-6[1
  ▶ die : a Die
  ▶ over : false
  ▶ players : an OrderedCollection (size: 2)
  ▼ squares : an OrderedCollection (size: 12)
    ▶ array : an Array (size: 20)
    ▶ firstIndex : 1
    ▶ lastIndex : 12
    ▶ Class : OrderedCollection
    ▶ Methods
    ▼ Elements
      ▼ [1] : [1 Jack Jill]
        ▶ board : [1 Jack Jill][2]4+>[3][4][5][6][7
        ▶ player : nil
        ▼ players : an OrderedCollection (size: 2)
          ▶ array : an Array (size: 10)
          ▶ firstIndex : 3
          ▶ lastIndex : 4
          ▶ Class : OrderedCollection
          ▶ Methods
          ▼ Elements
            ▼ [1] : Jack
              ▶ name : 'Jack'
              ▶ square : [1 Jack Jill]
              ▶ Class : GamePlayer
              ▶ Methods
            ▶ [2] : Jill
        ▶ position : 1
        ▶ Class : FirstSquare
        ▶ Methods
```

```
Jack
```

Use the Inspector to make ad hoc changes

> You can use the Inspector as an ad hoc interface to modify the state of a running system
  — Use this sparingly!



```
[1 Jack Jill][2]4+>[3][4][5][6][7]2+>[8][9][10]<-6[11][12] on NewInspec
▼ [1 Joe Jill][2]4+>[3][4][5][6][7]2+>[8][9][10]<-6[11][12]      'Joe'
  ▶ die : a Die
  ▶ over : false
  ▼ players : an OrderedCollection (size: 2)
    ▶ array : an Array (size: 10)
    ▶ firstIndex : 3
    ▶ lastIndex : 4
    ▶ Class : OrderedCollection
    ▶ Methods
    ▼ Elements
      ▼ [1] : Joe
        ▼ name : 'Joe'
          ▶ Class : ByteString
          ▶ Methods
          ▶ Elements
        ▶ square : [1 Joe Jill]
        ▶ Class : GamePlayer
        ▶ Methods
      ▶ [2] : Jill
  ▶ squares : an OrderedCollection (size: 12)
  ▶ turn : 1
  ▶ Class : SnakesAndLadders
  ▶ Methods
```

If we change the name of a GamePlayer, this will be reflected in the running system.

Modify a running system

> You can change the code *on the fly* while you are running the system
- — Keep the Inspector open
- — Keep the Debugger open

> *You do not have to:*
- — Close the application and any views (inspectors, debuggers)
- — Implement your changes
- — Compile
- — Restart

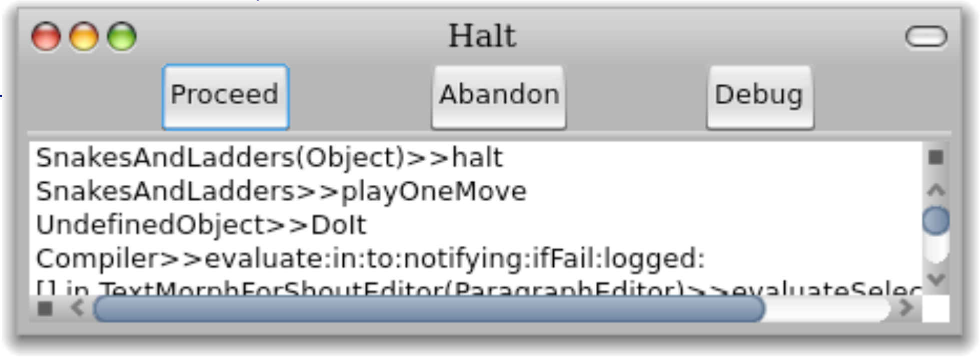
Well, sometimes you have to …

> ***Just keep everything running while you are changing things***

Breakpoints

> Send the message `self halt` to start the debugger at an arbitrary location

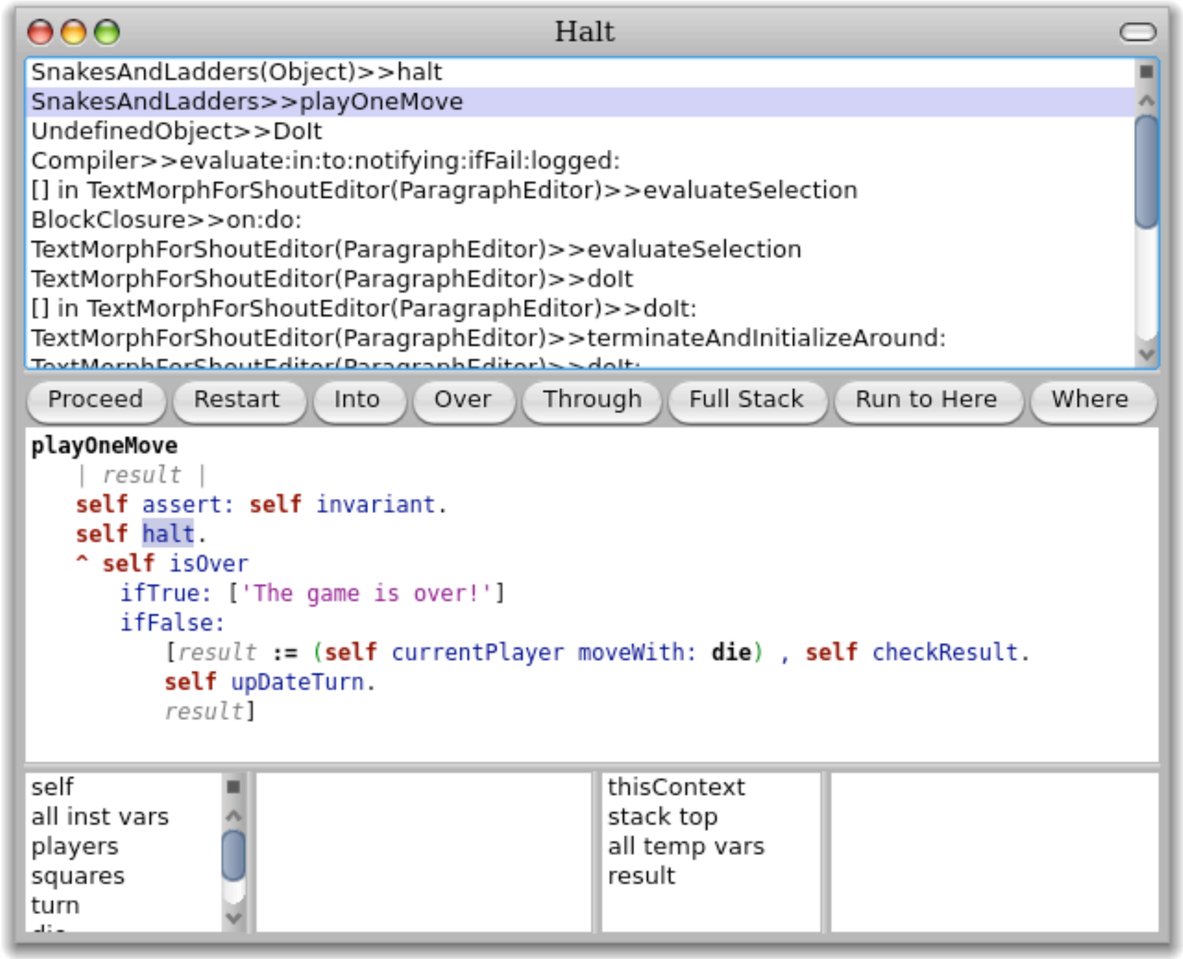
```
SnakesAndLadders>>playOneMove
    | result |
    self assert: self invariant.
    self halt.
    ^ self isOver
      …
```



```
SnakesAndLadders(Object)>>halt
SnakesAndLadders>>playOneMove
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:logged:
[] in TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelec
```

Debugging

Step over or into methods to track the state



```
SnakesAndLadders(Object)>>halt
SnakesAndLadders>>playOneMove
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:logged:
[] in TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
BlockClosure>>on:do:
TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
TextMorphForShoutEditor(ParagraphEditor)>>doIt
[] in TextMorphForShoutEditor(ParagraphEditor)>>doIt:
TextMorphForShoutEditor(ParagraphEditor)>>terminateAndInitializeAround:
```

```
playOneMove
    | result |
    self assert: self invariant.
    self halt.
    ^ self isOver
        ifTrue: ['The game is over!']
        ifFalse:
            [result := (self currentPlayer moveWith: die) , self checkResult.
            self upDateTurn.
            result]
```

```
self
all inst vars
players
squares
turn
```

```
thisContext
stack top
all temp vars
result
```

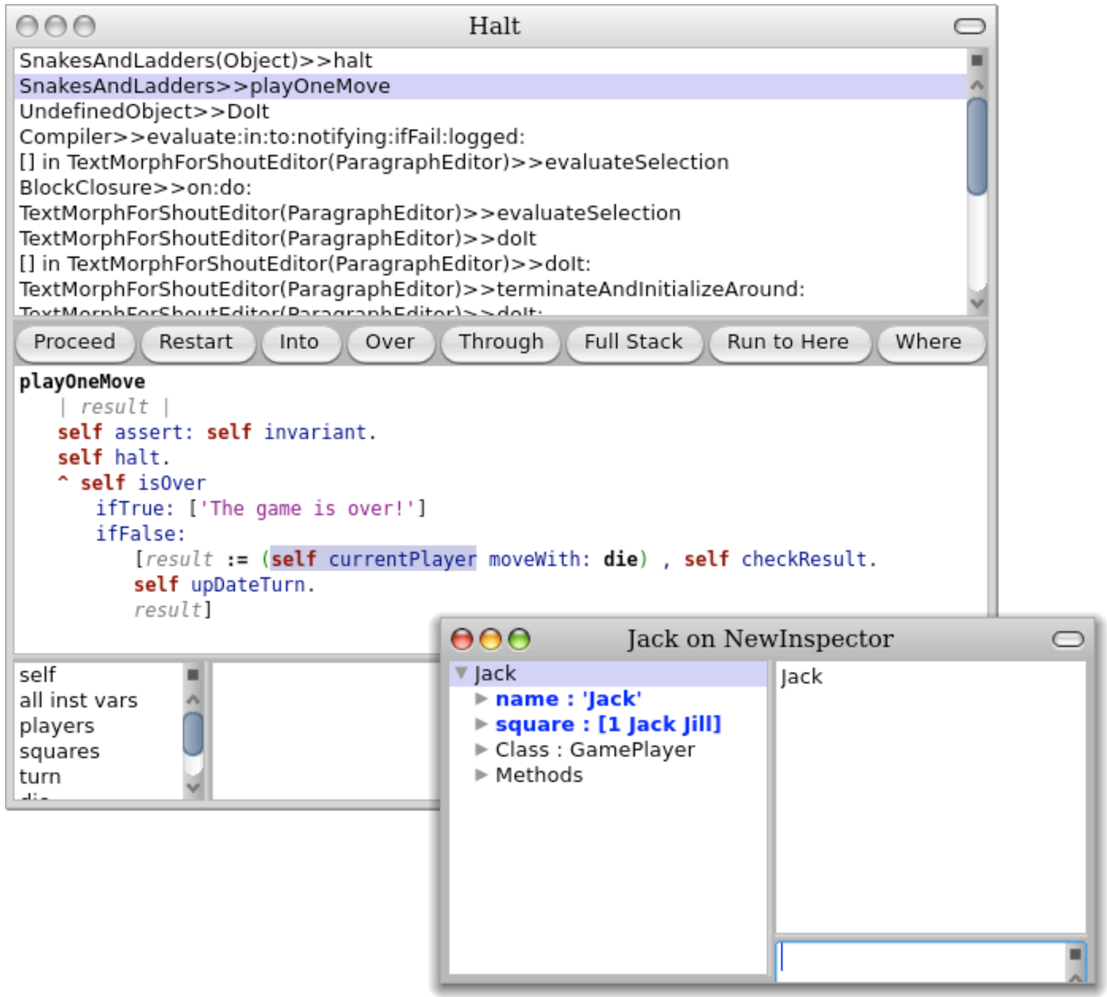
The Debugger is your friend!

Everything is an object!

> You can:
  - Inspect any entity
  - Evaluate any code
  - Modify code on the fly

Don't forget:

- Keep the Debugger open!



```
SnakesAndLadders(Object)>>halt
SnakesAndLadders>>playOneMove
UndefinedObject>>DoIt
Compiler>>evaluate:in:to:notifying:ifFail:logged:
[] in TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
BlockClosure>>on:do:
TextMorphForShoutEditor(ParagraphEditor)>>evaluateSelection
TextMorphForShoutEditor(ParagraphEditor)>>doIt
[] in TextMorphForShoutEditor(ParagraphEditor)>>doIt:
TextMorphForShoutEditor(ParagraphEditor)>>terminateAndInitializeAround:
```

```
playOneMove
    | result |
    self assert: self invariant.
    self halt.
    ^ self isOver
        ifTrue: ['The game is over!']
        ifFalse:
            [result := (self currentPlayer moveWith: die) , self checkResult.
            self upDateTurn.
            result]
```

```
self
all inst vars
players
squares
turn
```

```
▼ Jack
  ▶ name : 'Jack'
  ▶ square : [1 Jack Jill]
  ▶ Class : GamePlayer
  ▶ Methods
```

```
Jack
```

Dangling self halt

- When you have finished debugging, don't forget to remove any `self halt` in the code!
  - Running all the tests should catch this!

The Browser is your friend!

Learn to tinker with the system

> Example:
  — How can we browse all methods that send to `super`?

> We follow a browsing path:
  1. “browse”
  2. `Object>>browse`
  3. `Object>>systemNavigation`
  4. `SystemNavigation`
  5. `SystemNavigation>>browseMethodsWithSourceString:`

> First solution:

```
SystemNavigation default
   browseMethodsWithSourceString: 'super'
```

A bit slow, and contains many false negatives

The Message Name Finder is your friend!

> We continue browsing:
1. `SystemNavigation>>browseMethodsWith*`
2. `SystemNavigation>>browseAllSelect:`

> Query the Message Name Finder for “super”
— Yields `CompiledMethod>>sendsToSuper`

> Better solution:

```
SystemNavigation default
   browseAllSelect: [:method | method sendsToSuper ]
```

Fast, and accurate!

What you should know!

- ✎ *When should you explicitly return* `self`*?*
- ✎ *Why shouldn't you redefine methods named* `basic*`*?*
- ✎ *Why are blocks not full closures?*
- ✎ *How do you provide access to instance variables that are collections, without breaking encapsulation?*
- ✎ *What is one of the most important uses of* `super`*?*
- ✎ *How does programming with Smalltalk differ from programming in a conventional static language?*

Can you answer these questions?

- ✎ *What will happen if you redefine the method `class`?*
- ✎ *When should you define accessors for instance variables?*
- ✎ *How can explicit references to class names make your application fragile?*
- ✎ *Where is the method `halt` defined?*

License

http://creativecommons.org/licenses/by-sa/3.0/