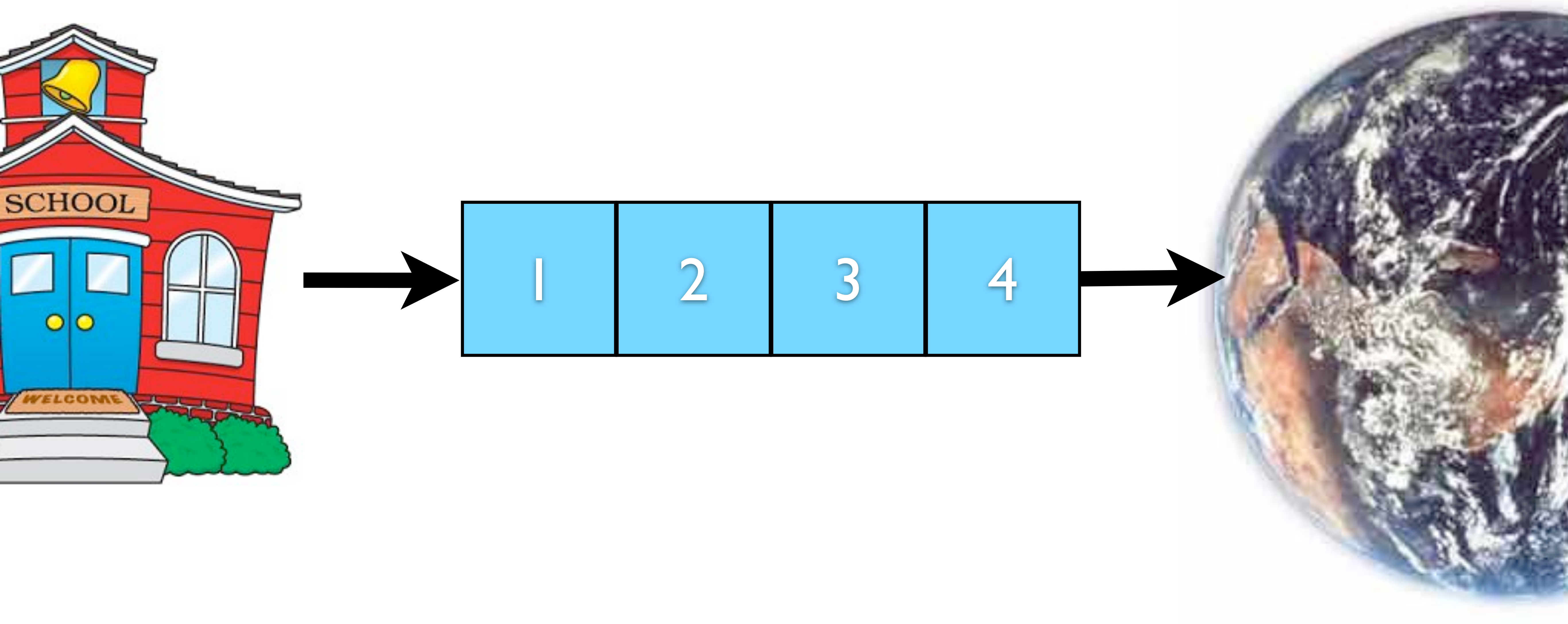
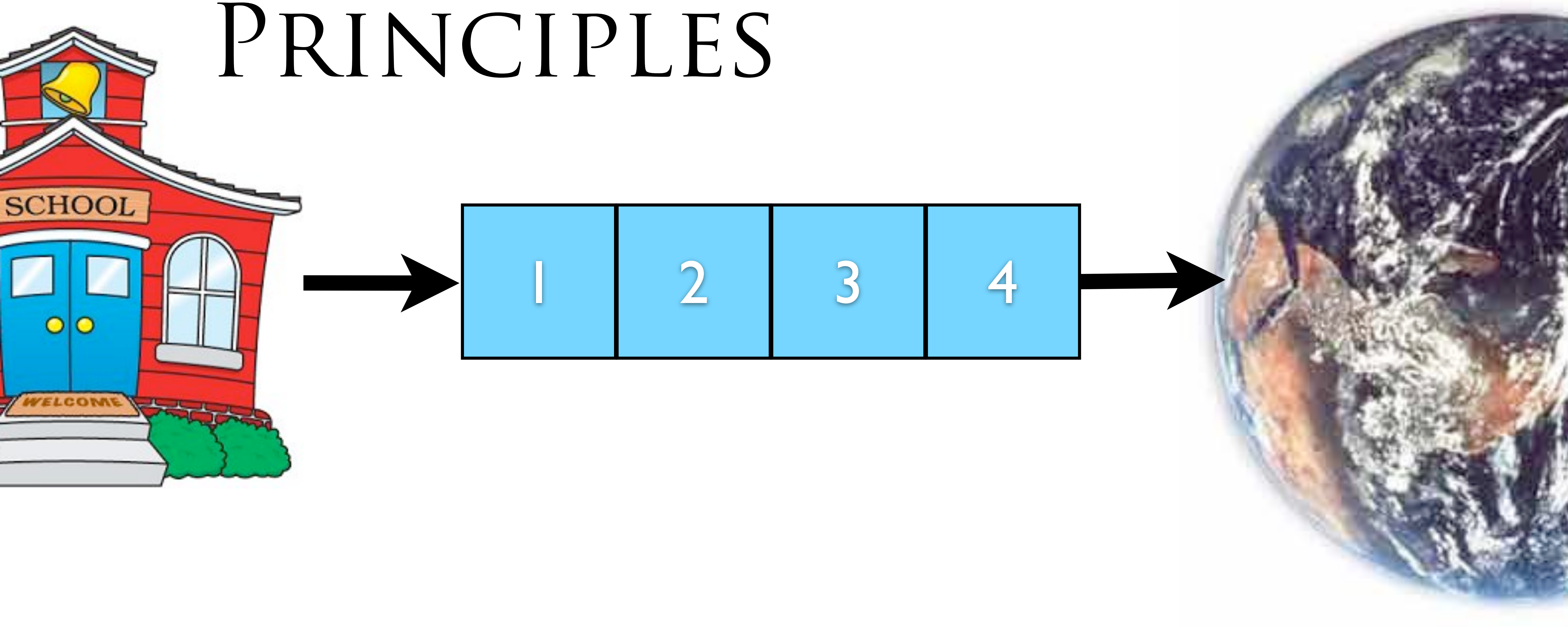
From Principles to Practice with Class in the First Year

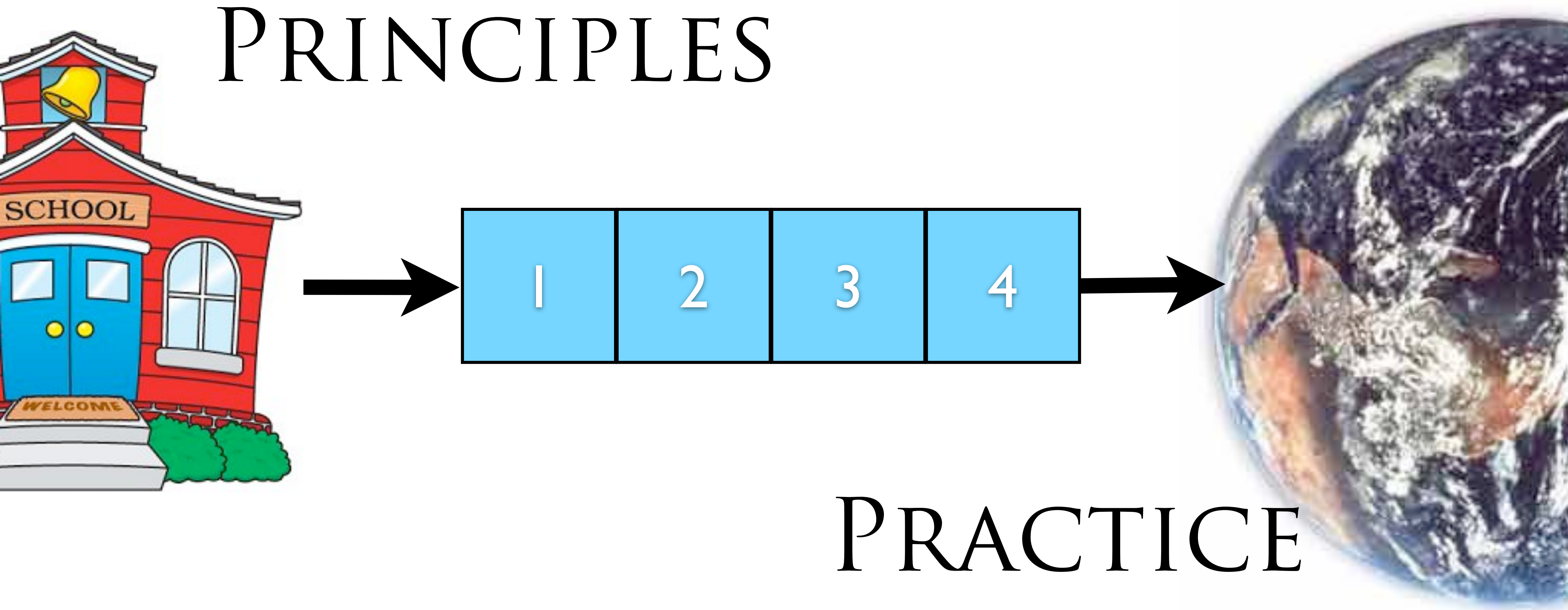
Sam Tobin-Hochstadt
David Van Horn



NORTHEASTERN UNIVERSITY
LVX VERITAS VIRTVS

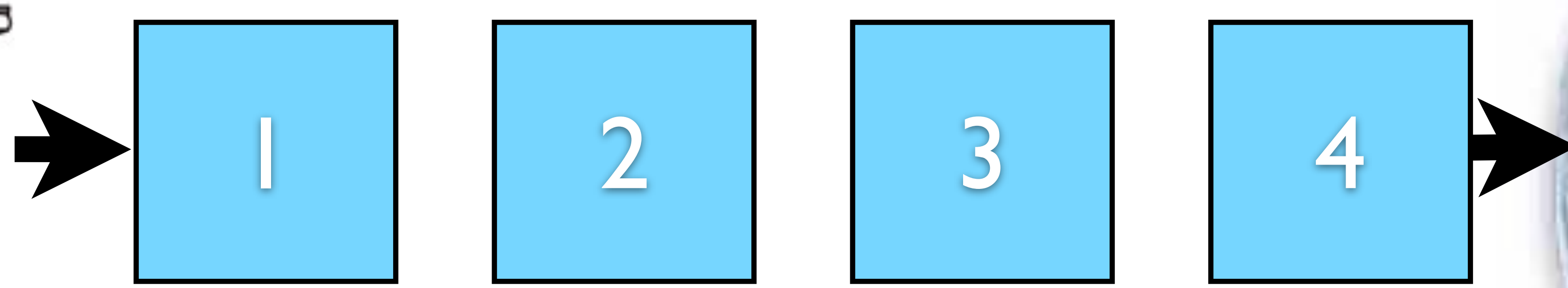
Principles





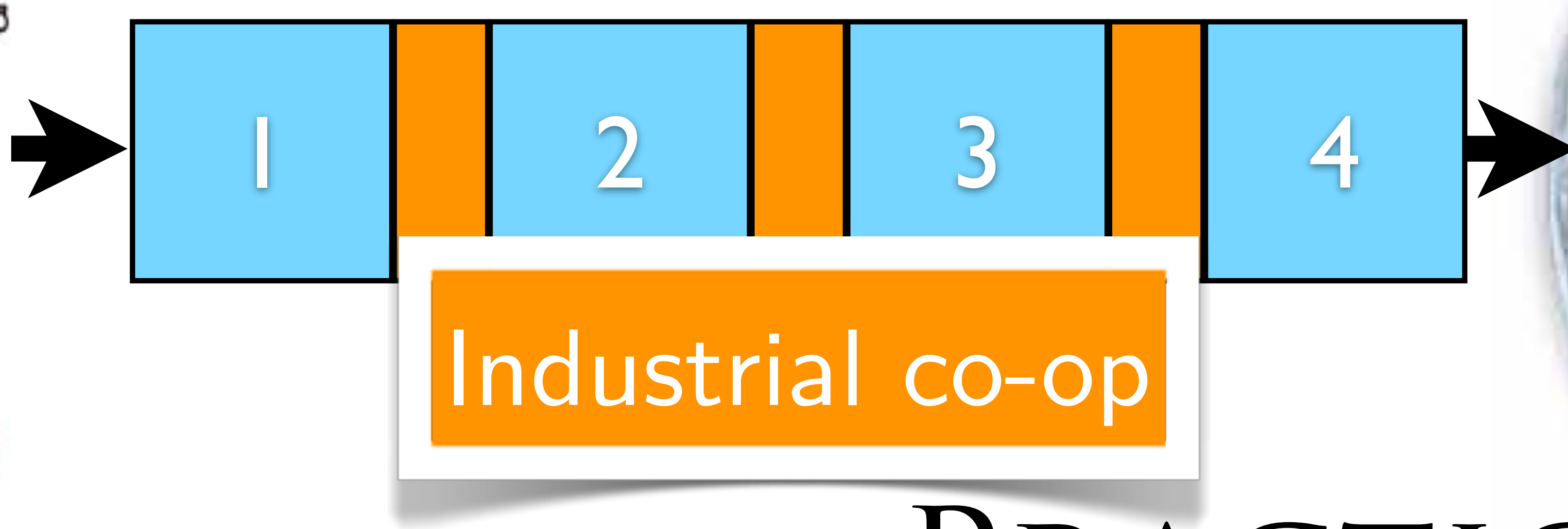
NORTHEASTERN UNIVERSITY
LVX VERITAS VIRTVS

Principles



Practice

Principles



Practice

Principles

I

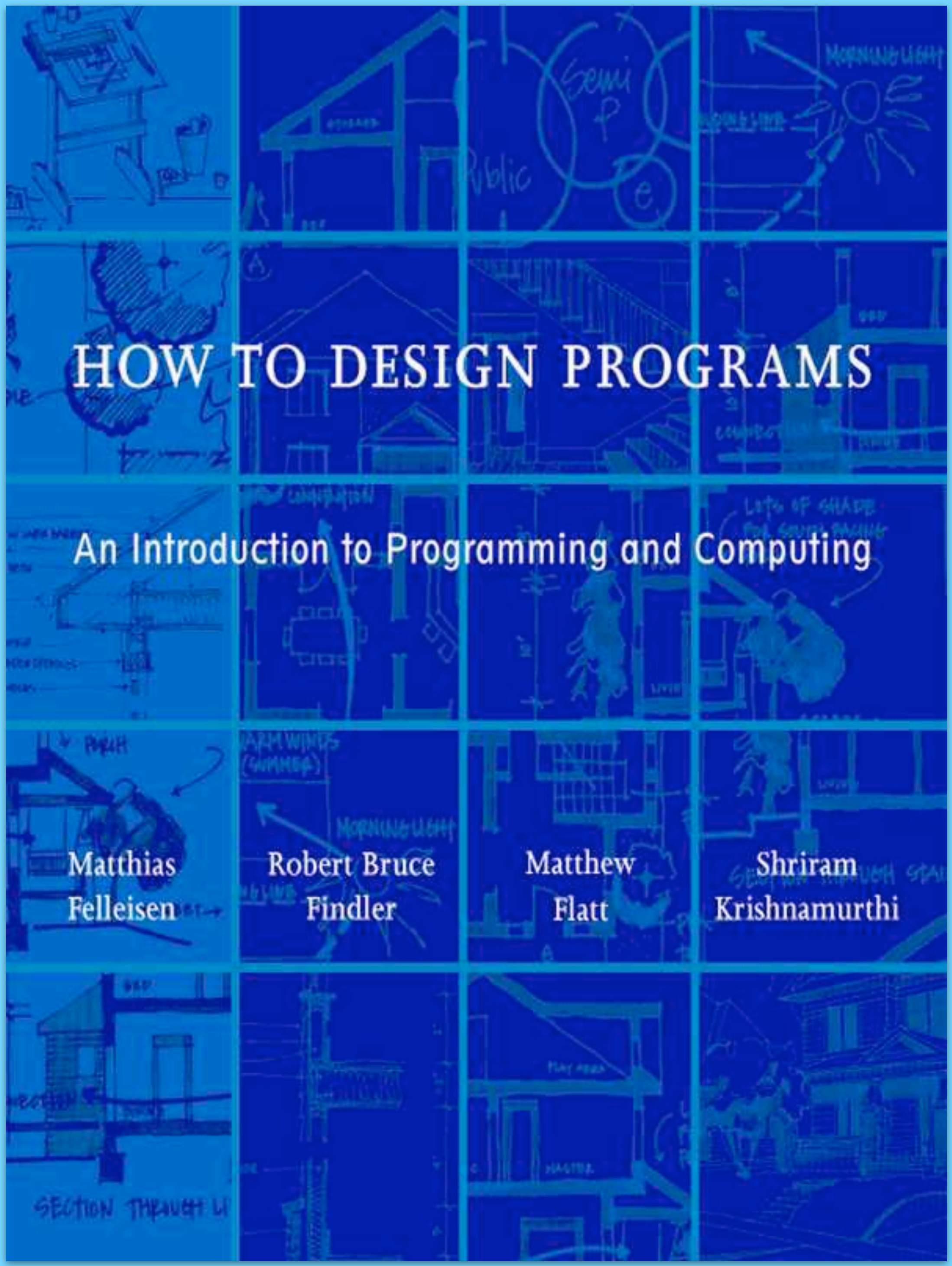
Industrial co-op

Practice

Principles

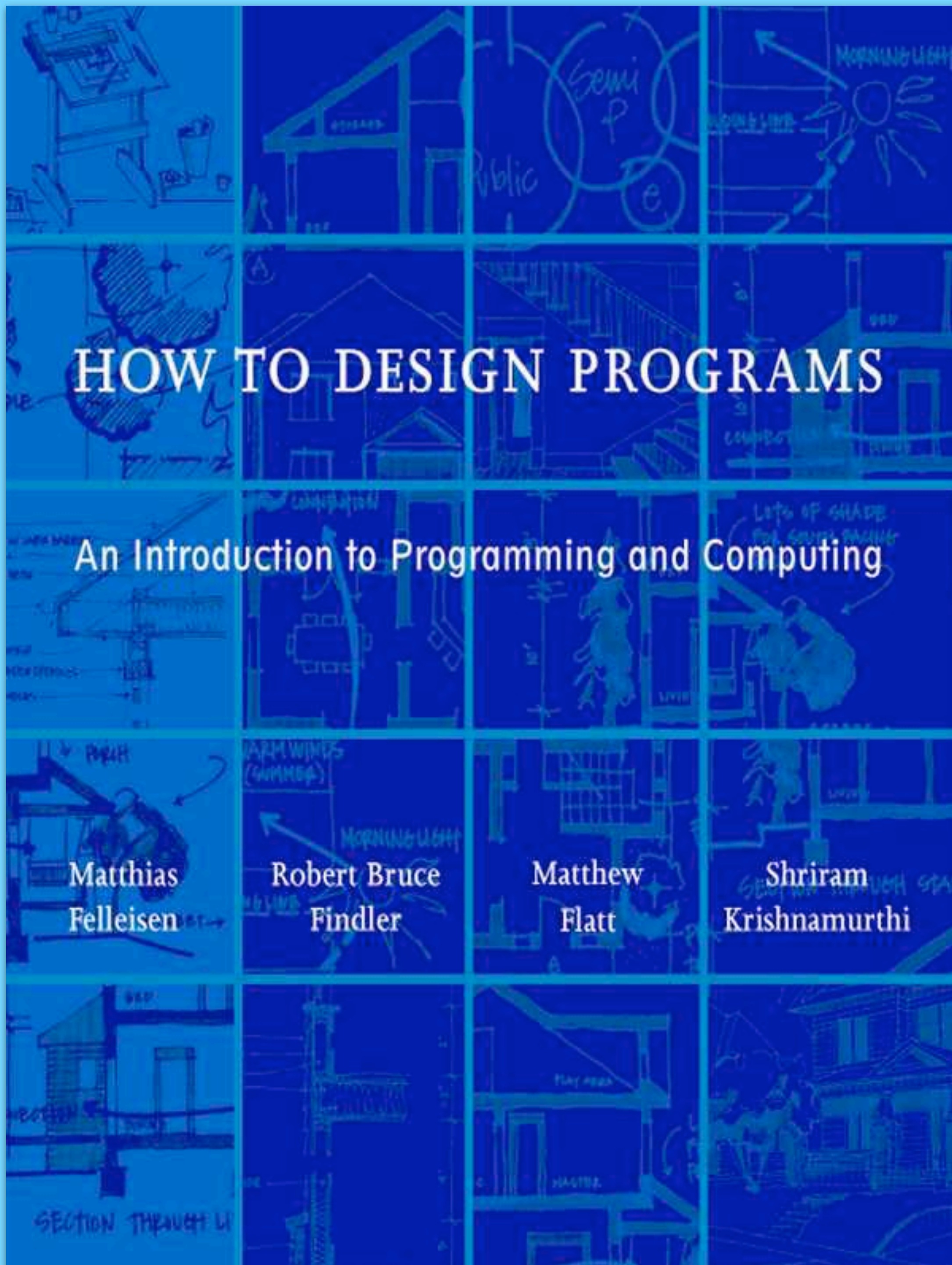
Practice

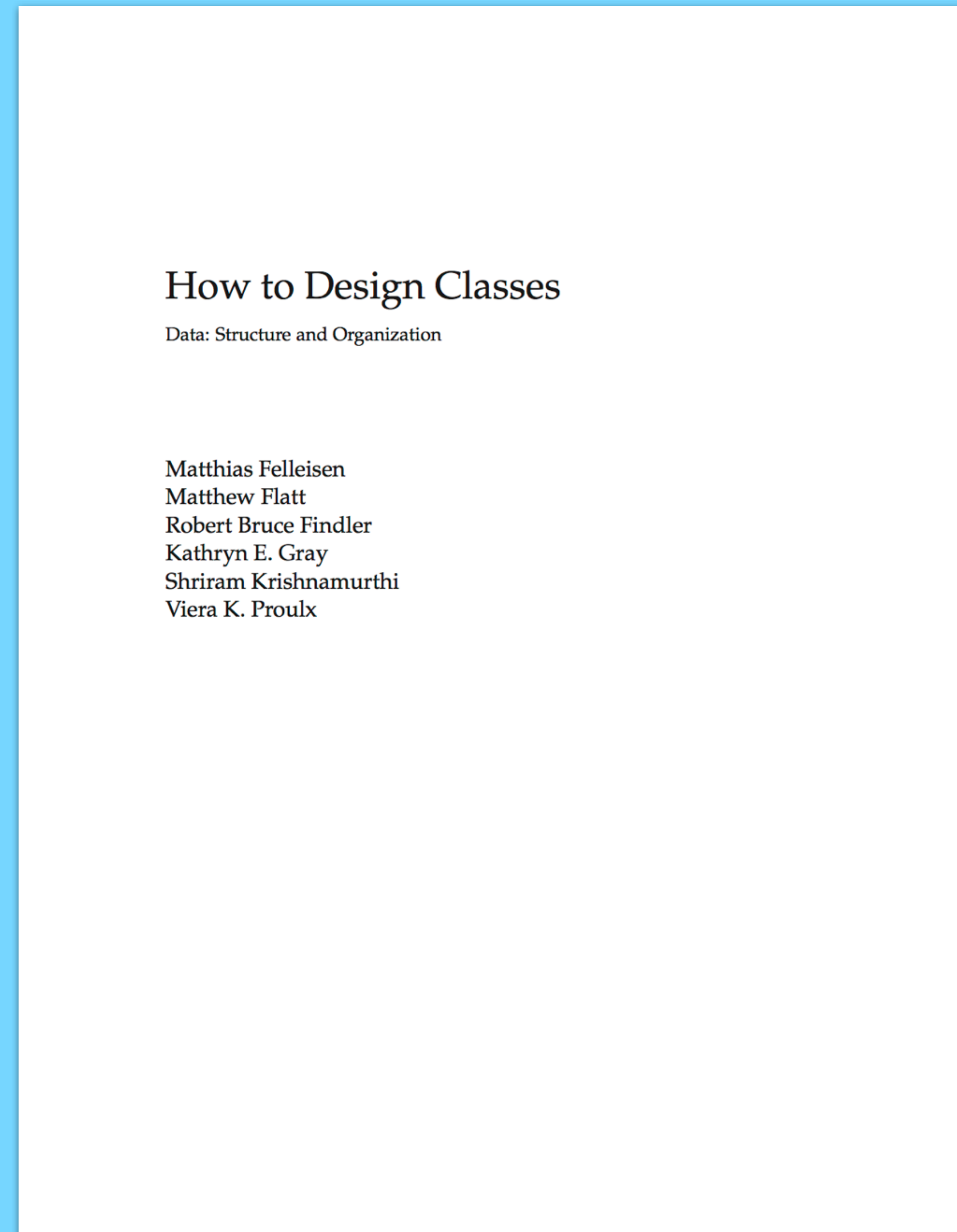
Principles



Practice

Principles



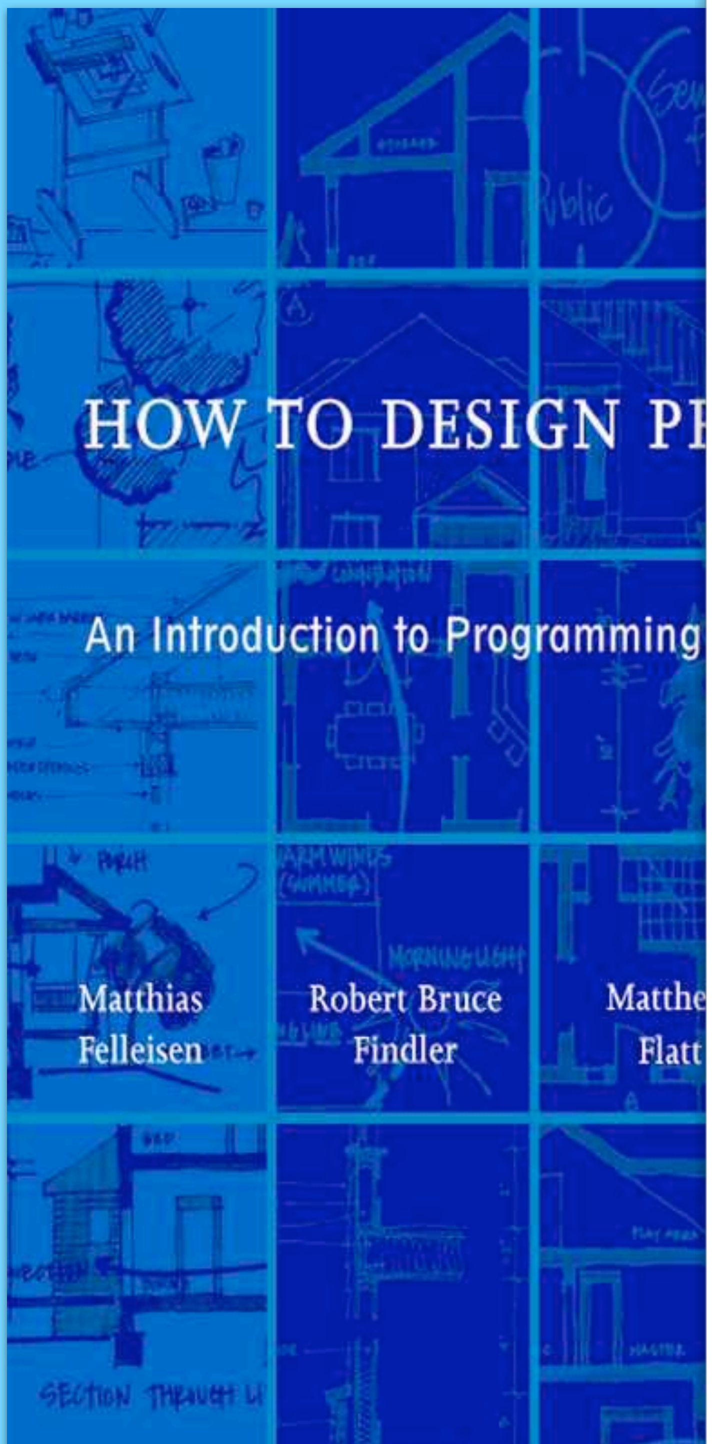

How to Design Classes

Data: Structure and Organization

Matthias Felleisen
Matthew Flatt
Robert Bruce Findler
Kathryn E. Gray
Shriram Krishnamurthi
Viera K. Proulx

Practice

PRINCIPLES



Under consideration for publication in J. Functional Programming

EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA
Robert Bruce Findler, University of Chicago, Chicago, IL, USA
Matthew Flatt, University of Utah, Salt Lake City, UT, USA
Shriram Krishnamurthi, Brown University, Providence, RI, USA
Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman's *Structure and Interpretation of Computer Programs* radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to shortcomings of the book and the whimsies of Scheme. Worse, the experiment left people with a bad impression of Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed *How to Design Programs*. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) (Abelson *et al.*, 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, SICP focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to SICP and Scheme. The book became a major bestseller for MIT Press.[1] Along with SICP, the Scheme programming language (Sussman & Steele Jr.,

[1] According to Bob Prior (editor at MIT Press), SICP sold 45,000 copies in its first five years [personal communication, 9 June 2003].

Classes

PRACTICE

EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA

Robert Bruce Findler, University of Chicago, Chicago, IL, USA

Matthew Flatt, University of Utah, Salt Lake City, UT, USA

Shriram Krishnamurthi, Brown University, Providence, RI, USA

Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Abstract

Twenty years ago Abelson and Sussman's *Structure and Interpretation of Computer Programs* radically changed the intellectual landscape of introductory computing courses. Instead of teaching some currently fashionable programming language, it employed Scheme and functional programming to teach important ideas. Introductory courses based on the book showed up around the world and made Scheme and functional programming popular. Unfortunately, these courses quickly disappeared again due to shortcomings of the book and the whimsies of Scheme. Worse, the experiment left people with a bad impression of Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed *How to Design Programs*. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) (Abelson *et al.*, 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, SICP focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to SICP and Scheme. The book became a major bestseller for MIT Press.[1] Along with SICP, the Scheme programming language (Sussman & Steele Jr.,

[1] According to Bob Prior (editor at MIT Press), SICP sold 45,000 copies in its first five years [personal communication, 9 June 2003].

EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA

Robert Bruce Findler, University of Chicago, Chicago, IL, USA

Matthew Flatt, University of Utah, Salt Lake City, UT, USA

Shriram Krishnamurthi, Brown University, Providence, RI, USA

Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

1. *Introduce only those language constructs that are necessary to teach programming principles*

Twenty years ago
grams radically ch
stead of teaching
and functional pro
book showed up a
Unfortunately, th
and the whimsies
Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Specifically, we present a framework for discussing the first-year curriculum and, based on it, the design rationale for our book and course, dubbed *How to Design Programs*. The approach emphasizes the systematic design of programs. Experience shows that it works extremely well as a preparation for a course on object-oriented programming.

1 History and critique

The publication of Abelson and Sussman's *Structure and Interpretation of Computer Programs* (SICP) (Abelson *et al.*, 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, SICP focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

Over a short period, many universities in the US and around the world switched their first course to SICP and Scheme. The book became a major bestseller for MIT Press.[1] Along with SICP, the Scheme programming language (Sussman & Steele Jr.,

[1] According to Bob Prior (editor at MIT Press), SICP sold 45,000 copies in its first five years [personal communication, 9 June 2003].

EDUCATIONAL PEARL

The Structure and Interpretation of the Computer Science Curriculum

Matthias Felleisen, Northeastern University, Boston, MA, USA

Robert Bruce Findler, University of Chicago, Chicago, IL, USA

Matthew Flatt, University of Utah, Salt Lake City, UT, USA

Shriram Krishnamurthi, Brown University, Providence, RI, USA

Email: {matthias,robby,mflatt,shriram}@plt-scheme.org

Twenty years ago
grams radically ch
stead of teaching
and functional pro
book showed up a
Unfortunately, th
and the whimsies
Scheme and functional programming in general.

In this pearl, we propose an alternative role for functional programming in the first-year curriculum. Speci
and, based on it,
Programs. The ap
that it works extr

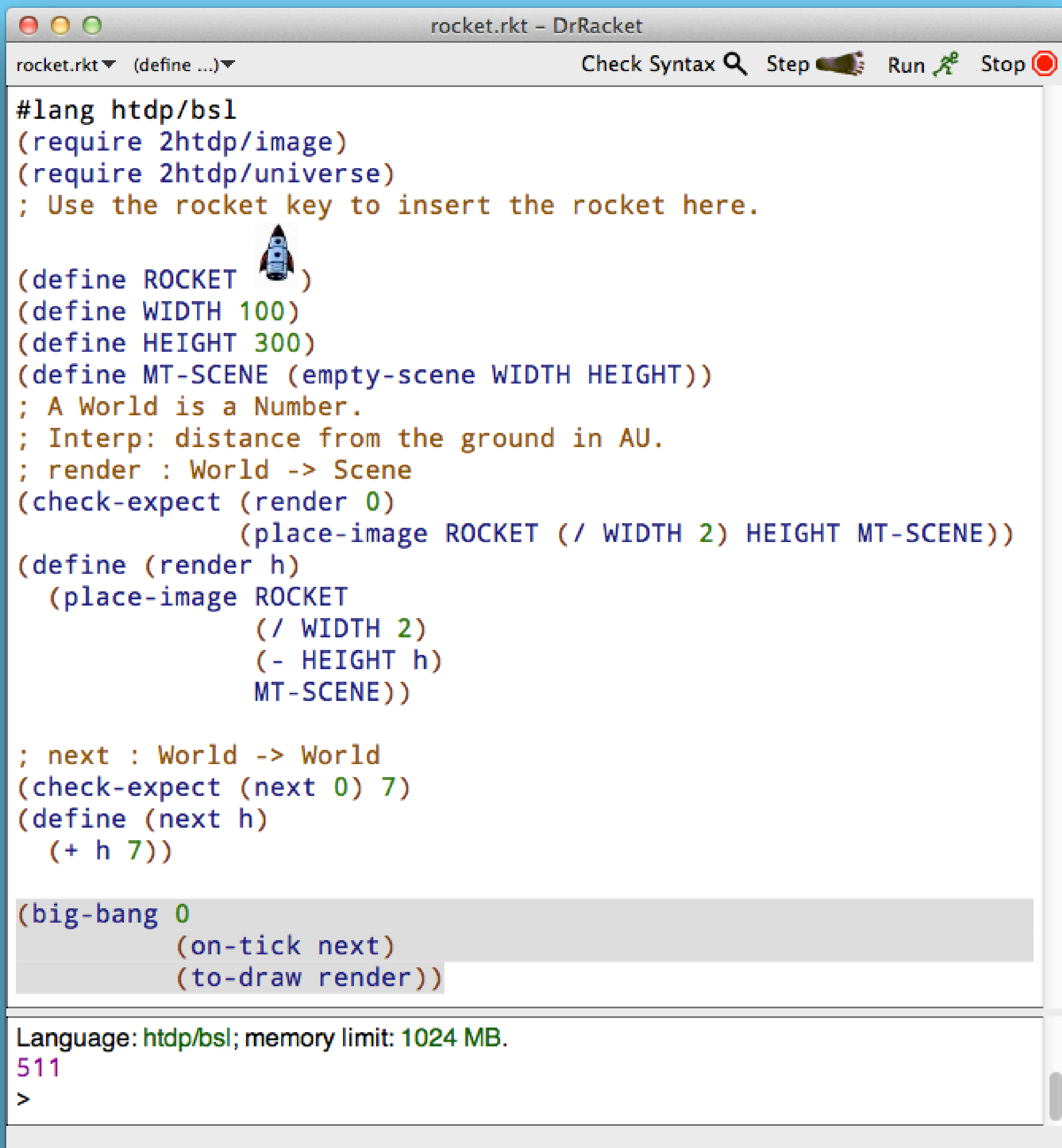
1. *Introduce only those language constructs that are necessary to teach programming principles*

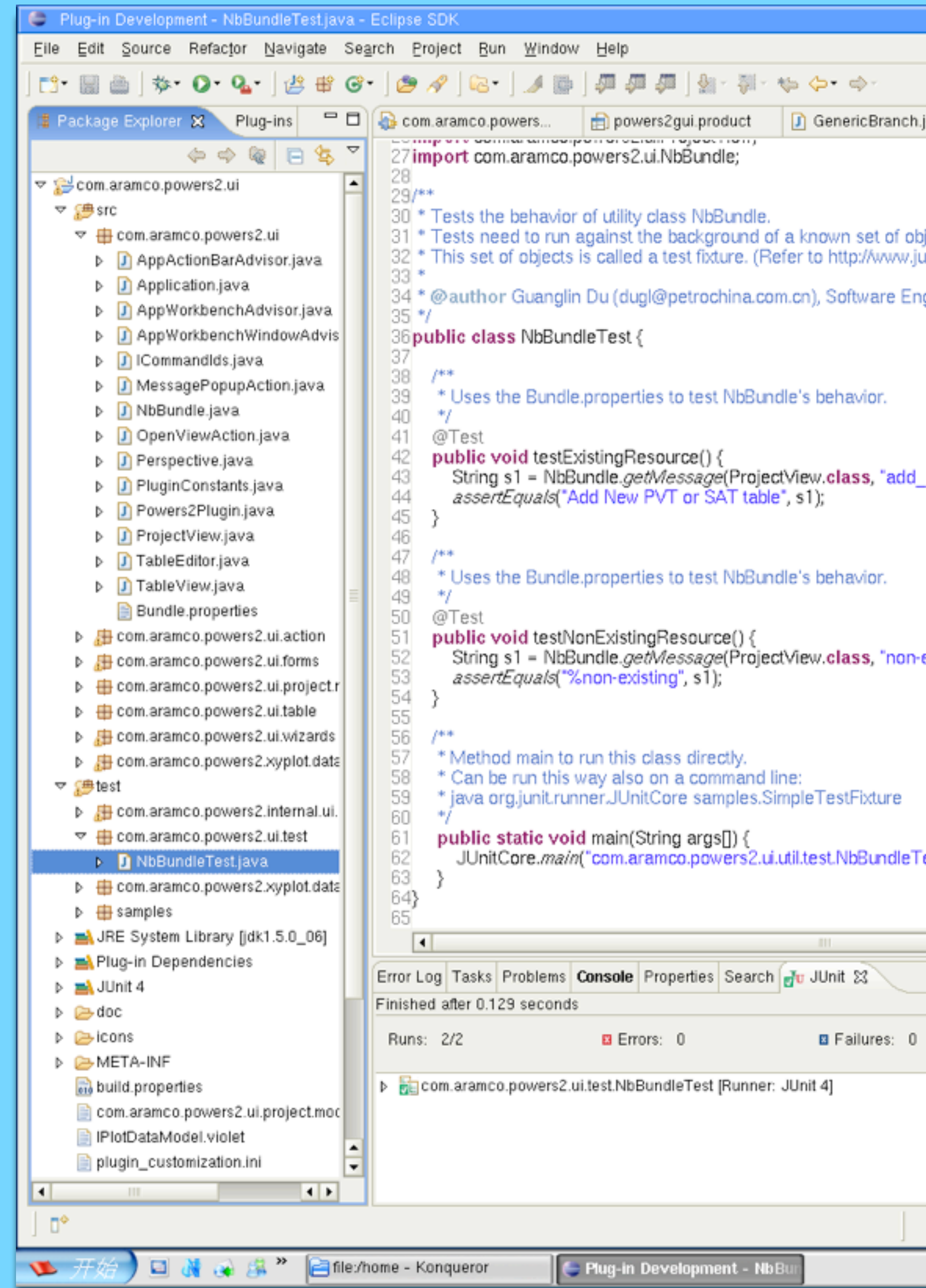
2. *Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time*

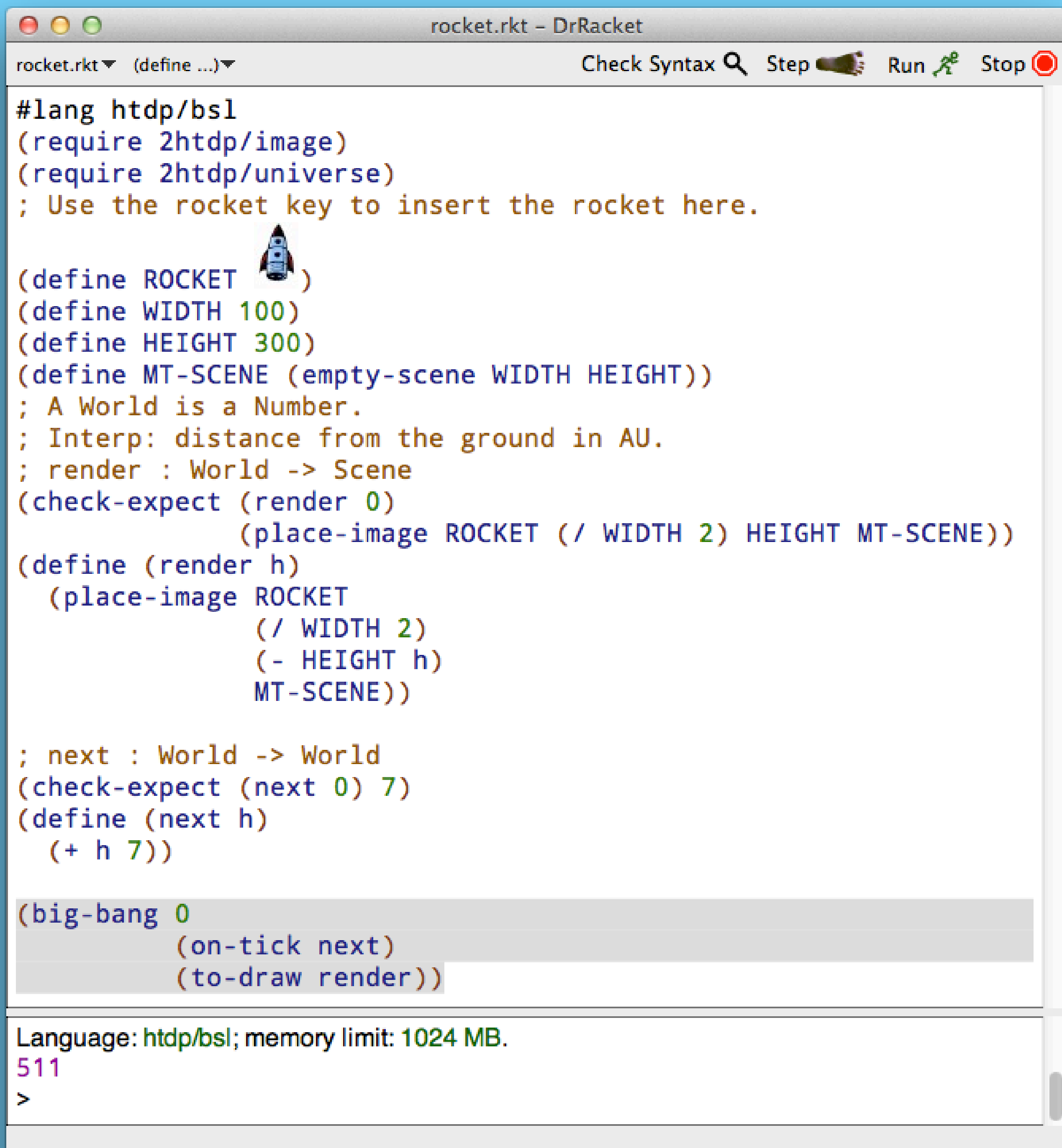
The publication
puter Programs (SICP) (Abelson *et al.*, 1985) revolutionized the landscape of the introductory computing curriculum in the 1980s. Most importantly, the book liberated the introductory course from the tyranny of syntax. Instead of arranging a course around the syntax of a currently fashionable programming language, SICP focused the first course on the study of important ideas in computing: functional abstraction, data abstraction, streams, data-directed programming, implementation of message-passing objects, interpreters, compilers, and register machines.

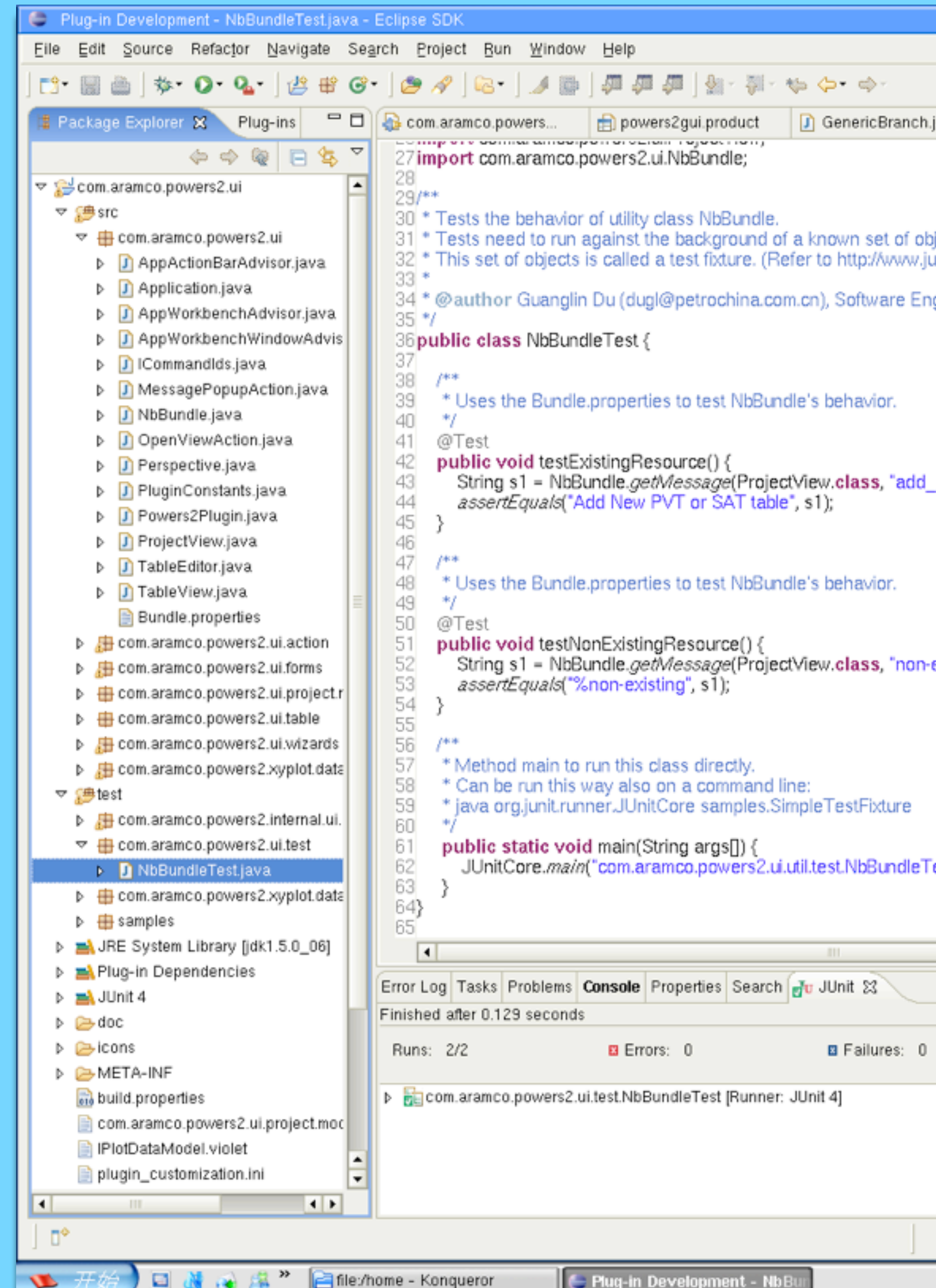
Over a short period, many universities in the US and around the world switched their first course to SICP and Scheme. The book became a major bestseller for MIT Press.[1] Along with SICP, the Scheme programming language (Sussman & Steele Jr.,

[1] According to Bob Prior (editor at MIT Press), SICP sold 45,000 copies in its first five years [personal communication, 9 June 2003].


rocket.rkt – DrRacket
rocket.rkt (define ...) Check Syntax Step Run Stop
#lang htdp/bsl
(require 2htdp/image)
(require 2htdp/universe)
; Use the rocket key to insert the rocket here.
(define ROCKET)
(define WIDTH 100)
(define HEIGHT 300)
(define MT-SCENE (empty-scene WIDTH HEIGHT))
; A World is a Number.
; Interp: distance from the ground in AU.
; render : World -> Scene
(check-expect (render 0)
(place-image ROCKET (/ WIDTH 2) HEIGHT MT-SCENE))
(define (render h)
(place-image ROCKET
(/ WIDTH 2)
(- HEIGHT h)
MT-SCENE))
; next : World -> World
(check-expect (next 0) 7)
(define (next h)
(+ h 7))
(big-bang 0
(on-tick next)
(to-draw render))
Language: htdp/bsl; memory limit: 1024 MB.
511
>


Plug-in Development - NbBundleTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer Plug-ins
com.aramco.powers... powers2gui.product GenericBranch.j
com.aramco.powers2.ui
src
com.aramco.powers2.ui
AppActionBarAdvisor.java
Application.java
AppWorkbenchAdvisor.java
AppWorkbenchWindowAdvis
ICommandIds.java
MessagePopupAction.java
NbBundle.java
OpenViewAction.java
Perspective.java
PluginConstants.java
Powers2Plugin.java
ProjectView.java
TableEditor.java
TableView.java
Bundle.properties
com.aramco.powers2.ui.action
com.aramco.powers2.ui.forms
com.aramco.powers2.ui.project.r
com.aramco.powers2.ui.table
com.aramco.powers2.ui.wizards
com.aramco.powers2.xyplot.data
test
com.aramco.powers2.internal.ui.
com.aramco.powers2.ui.test
NbBundleTest.java
com.aramco.powers2.xyplot.data
samples
JRE System Library [jdk1.5.0_06]
Plug-in Dependencies
JUnit 4
doc
icons
META-INF
build.properties
com.aramco.powers2.ui.project.moc
IPlotDataModel.violet
plugin_customization.ini
import com.aramco.powers2.ui.NbBundle;
/**
* Tests the behavior of utility class NbBundle.
* Tests need to run against the background of a known set of obj
* This set of objects is called a test fixture. (Refer to http://www.ju
*
* @author Guanglin Du (dugl@petrochina.com.cn), Software Eng
*/
public class NbBundleTest {
/**
* Uses the Bundle.properties to test NbBundle's behavior.
*/
@Test
public void testExistingResource() {
String s1 = NbBundle.getMessage(ProjectView.class, "add_
assertEquals("Add New PVT or SAT table", s1);
}
/**
* Uses the Bundle.properties to test NbBundle's behavior.
*/
@Test
public void testNonExistingResource() {
String s1 = NbBundle.getMessage(ProjectView.class, "non-e
assertEquals("%non-existing", s1);
}
/**
* Method main to run this class directly.
* Can be run this way also on a command line:
* java org.junit.runner.JUnitCore samples.SimpleTestFixture
*/
public static void main(String args[]) {
JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleTe
}
}
Error Log Tasks Problems Console Properties Search JUnit
Finished after 0.129 seconds
Runs: 2/2 Errors: 0 Failures: 0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
开始 file:/home - Konqueror Plug-in Development - NbBu


rocket.rkt – DrRacket
rocket.rkt (define ...)
Check Syntax Step Run Stop
#lang htdp/bsl
(require 2htdp/image)
(require 2htdp/universe)
; Use the rocket key to insert the rocket here.
(define ROCKET)
(define WIDTH 100)
(define HEIGHT 300)
(define MT-SCENE (empty-scene WIDTH HEIGHT))
; A World is a Number.
; Interp: distance from the ground in AU.
; render : World -> Scene
(check-expect (render 0)
(place-image ROCKET (/ WIDTH 2) HEIGHT MT-SCENE))
(define (render h)
(place-image ROCKET
(/ WIDTH 2)
(- HEIGHT h)
MT-SCENE))
; next : World -> World
(check-expect (next 0) 7)
(define (next h)
(+ h 7))
(big-bang 0
(on-tick next)
(to-draw render))
Language: htdp/bsl; memory limit: 1024 MB.
511
>


Plug-in Development - NbBundleTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer Plug-ins
com.aramco.powers... powers2gui.product GenericBranch.j
com.aramco.powers2.ui
src
com.aramco.powers2.ui
AppActionBarAdvisor.java
Application.java
AppWorkbenchAdvisor.java
AppWorkbenchWindowAdvis
ICommandIds.java
MessagePopupAction.java
NbBundle.java
OpenViewAction.java
Perspective.java
PluginConstants.java
Powers2Plugin.java
ProjectView.java
TableEditor.java
TableView.java
Bundle.properties
com.aramco.powers2.ui.action
com.aramco.powers2.ui.forms
com.aramco.powers2.ui.project.r
com.aramco.powers2.ui.table
com.aramco.powers2.ui.wizards
com.aramco.powers2.xyplot.data
test
com.aramco.powers2.internal.ui.
com.aramco.powers2.ui.test
NbBundleTest.java
com.aramco.powers2.xyplot.data
samples
JRE System Library [jdk1.5.0_06]
Plug-in Dependencies
JUnit 4
doc
icons
META-INF
build.properties
com.aramco.powers2.ui.project.moc
IPlotDataModel.violet
plugin_customization.ini
27 import com.aramco.powers2.ui.NbBundle;
29 /**
30 * Tests the behavior of utility class NbBundle.
31 * Tests need to run against the background of a known set of obj
32 * This set of objects is called a test fixture. (Refer to http://www.ju
34 * @author Guanglin Du (dugl@petrochina.com.cn), Software Eng
35 */
36 public class NbBundleTest {
38 /**
39 * Uses the Bundle.properties to test NbBundle's behavior.
40 */
41 @Test
42 public void testExistingResource() {
43 String s1 = NbBundle.getMessage(ProjectView.class, "add_
44 assertEquals("Add New PVT or SAT table", s1);
45 }
47 /**
48 * Uses the Bundle.properties to test NbBundle's behavior.
49 */
50 @Test
51 public void testNonExistingResource() {
52 String s1 = NbBundle.getMessage(ProjectView.class, "non-e
53 assertEquals("%non-existing", s1);
54 }
56 /**
57 * Method main to run this class directly.
58 * Can be run this way also on a command line:
59 * java org.junit.runner.JUnitCore samples.SimpleTestFixture
60 */
61 public static void main(String args[]) {
62 JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleT
63 }
64 }
Error Log Tasks Problems Console Properties Search JUnit
Finished after 0.129 seconds
Runs: 2/2 Errors: 0 Failures: 0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
开始 file:/home - Konqueror Plug-in Development - NbBu

regular, minimal syntax

regular, minimal syntax

complicated irregular syntax

regular, minimal syntax

untyped

complicated irregular syntax

regular, minimal syntax	complicated irregular syntax
untyped	typed

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment

regular, minimal syntax

untyped

pedagogical environment

mathematical numbers

complicated irregular syntax

typed

industrial environment

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?
interaction	

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?
interaction	compilation

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?
interaction	compilation

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?
interaction	compilation
functions and structures	

regular, minimal syntax	complicated irregular syntax
untyped	typed
pedagogical environment	industrial environment
mathematical numbers	machine numbers
images as values	?
interaction	compilation
functions and structures	objects

regular, minimal syntax

untyped

pedagogical environment

mathematical numbers

images as values

interaction

functions and structures

objects

1. ***Introduce only those language constructs that are necessary to teach programming principles***

2. ***Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time***

objects

1. *Introduce only those language constructs that are necessary to teach programming principles*

2. *Choose a language with as few language constructs as possible, and one in which they can be introduced one at a time*

`#lang class`

objects

```
#lang class/0
(define-class posn (fields x y))
(new posn 3 4)
(send (new posn 3 4) x) ;=> 3
(send (new posn 3 4) y) ;=> 4
```

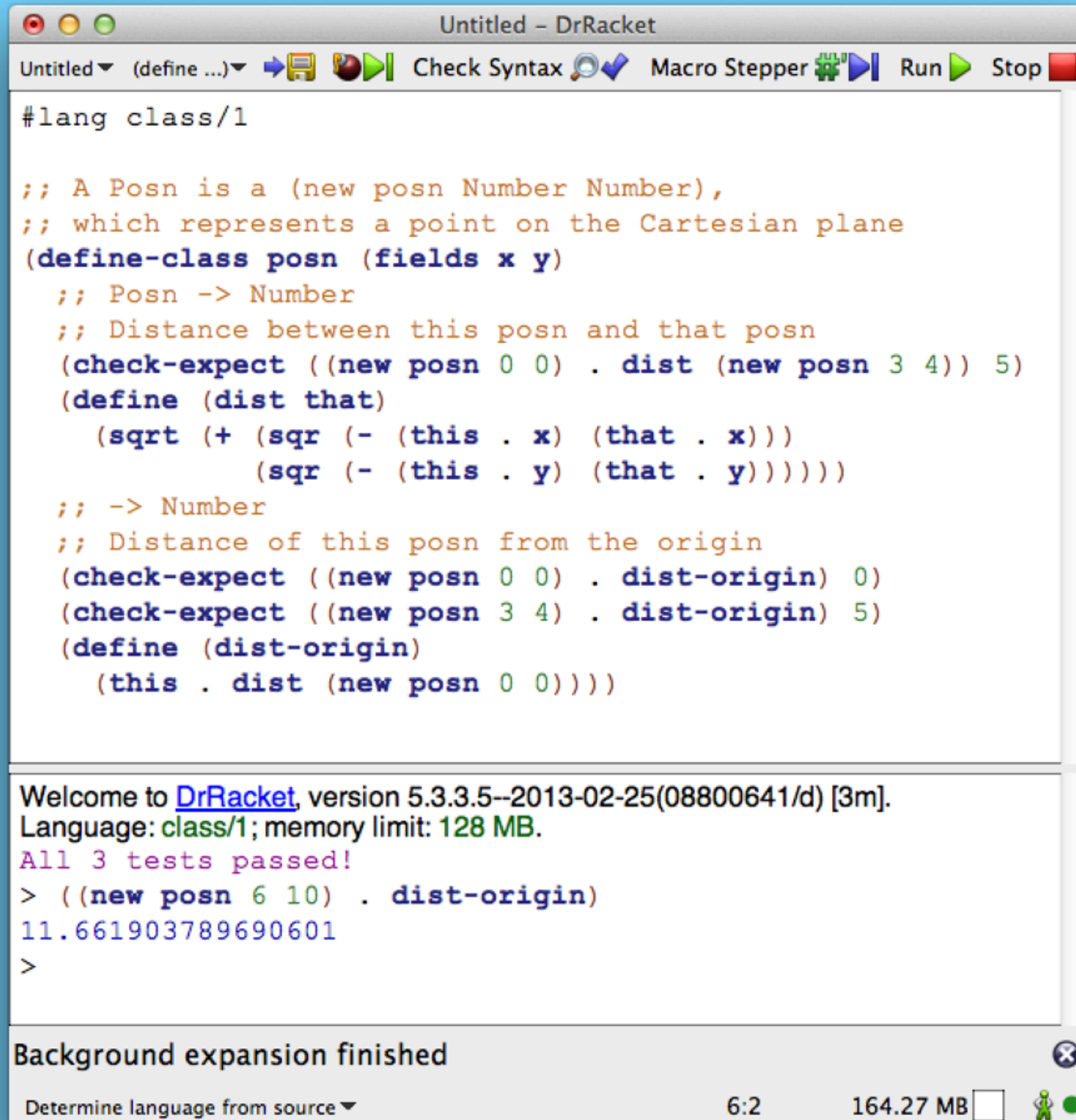
objects

```
#lang class/1
(define-class posn (fields x y))
(new posn 3 4)
((new posn 3 4) . x) ;=> 3
((new posn 3 4) . y) ;=> 4
```

objects

```
#lang class/1
(define-class posn (fields x y)
  ;; Posn -> Number
  ;; Distance between this posn and that posn
  (check-expect ((new posn 0 0) . dist (new posn 3 4)) 5)
  (define (dist that)
    (sqrt (+ (sqr (- (this . x) (that . x)))
             (sqr (- (this . y) (that . y))))))
  ;; -> Number
  ;; Distance of this posn from the origin
  (check-expect ((new posn 0 0) . dist-origin) 0)
  (check-expect ((new posn 3 4) . dist-origin) 5)
  (define (dist-origin)
    (this . dist (new posn 0 0))))
```

objects


Untitled – DrRacket
Untitled (define ...) Check Syntax Macro Stepper Run Stop
#lang class/1

;; A Posn is a (new posn Number Number),
;; which represents a point on the Cartesian plane
(define-class posn (fields x y)
  ;; Posn -> Number
  ;; Distance between this posn and that posn
  (check-expect ((new posn 0 0) . dist (new posn 3 4)) 5)
  (define (dist that)
    (sqrt (+ (sqr (- (this . x) (that . x)))
             (sqr (- (this . y) (that . y))))))
  ;; -> Number
  ;; Distance of this posn from the origin
  (check-expect ((new posn 0 0) . dist-origin) 0)
  (check-expect ((new posn 3 4) . dist-origin) 5)
  (define (dist-origin)
    (this . dist (new posn 0 0))))
Welcome to DrRacket, version 5.3.3.5--2013-02-25(08800641/d) [3m].
Language: class/1; memory limit: 128 MB.
All 3 tests passed!
> ((new posn 6 10) . dist-origin)
11.661903789690601
>
Background expansion finished
Determine language from source
6:2
164.27 MB

objects

```
;; A Tree is one of:
;; - (make-leaf Number)
;; - (make-node Tree Number Tree)
(define-struct leaf (v))
(define-struct node (left v right))

;; sum : Tree -> Number
;; sums the elements of the given tree
(define (sum a-tree)
  (cond
    [(leaf? a-tree) (leaf-v a-tree)]
    [else
     (+ (sum (node-left  a-tree))
        (node-v a-tree)
        (sum (node-right a-tree)))]))
```

```
#lang class/1
;;A Tree is one of:
;; - (new leaf Number)
;; - (new node Tree Number Tree)
;; and implements
;; sum : -> Number
;; sums the elements of this tree
(define-class leaf
  (fields v)
  (define (sum) (this . v)))

(define-class node
  (fields left v right)
  (define (sum)
    (+ (this . left . sum)
       (this . v)
       (this . right . sum))))
```

objects

```
(require 2htdp/image 2htdp/universe)

;; A World is a Number
(define-struct world (n))

;; tock : World -> World
(define (tock w)
  (make-world (add1 (world-n w))))

;; draw : World -> Image
(define (draw w)
  (rotate (modulo (world-n w) 360)
```

```
  ))

;;   : KeyEvent World -> World
(define (press k w) (make-world 0))

(big-bang (make-world 0)
 [on-tick tock]
 [on-draw draw]
 [on-key press])
```

```
#lang class/1
(require 2htdp/image class/universe)

;; A World is a (new world Number)
(define-class world
  (fields n)

  ;; on-tick : -> World
  (define (on-tick)
    (new world (add1 (this . n))))

  ;; to-draw : -> Image
  (define (to-draw)
    (rotate (modulo (this . n) 360)
```

```
    ))

  ;; on-key : KeyEvent -> World
  (define (on-key k) (new world 10)))

(big-bang (new world 0))
```

objects

#lang class

0. objects

#lang class

0. objects

#lang class

0. objects

1. shorthand method call

#lang class

0. objects

1. shorthand method call

2. inheritance

#lang class

0. objects

1. shorthand method call

2. inheritance

3. overriding

#lang class

0. objects

1. shorthand method call

2. inheritance

3. overriding

4. first-class classes

#lang class

0. objects

1. shorthand method call

2. inheritance

3. overriding

4. first-class classes

5. mutation

#lang class

0. objects

1. shorthand method call

2. inheritance

3. overriding

4. first-class classes

5. mutation

Java

```
import tester.*;

interface Tree {
    // sums the elements of this tree
    Integer sum();
}

class Leaf {
    Integer v;
    Leaf(Integer v) { this.v = v; }
    public Integer sum() { return this.v; }
}

class Node {
    Tree left; Integer v; Tree right;
    Node(Tree l, Integer v, Tree r) {
        this.left = l;
        this.v = v;
        this.right = r;
    }

    public Integer sum() {
        return this.left.sum() + this.v + this.right.sum();
    }
}
```

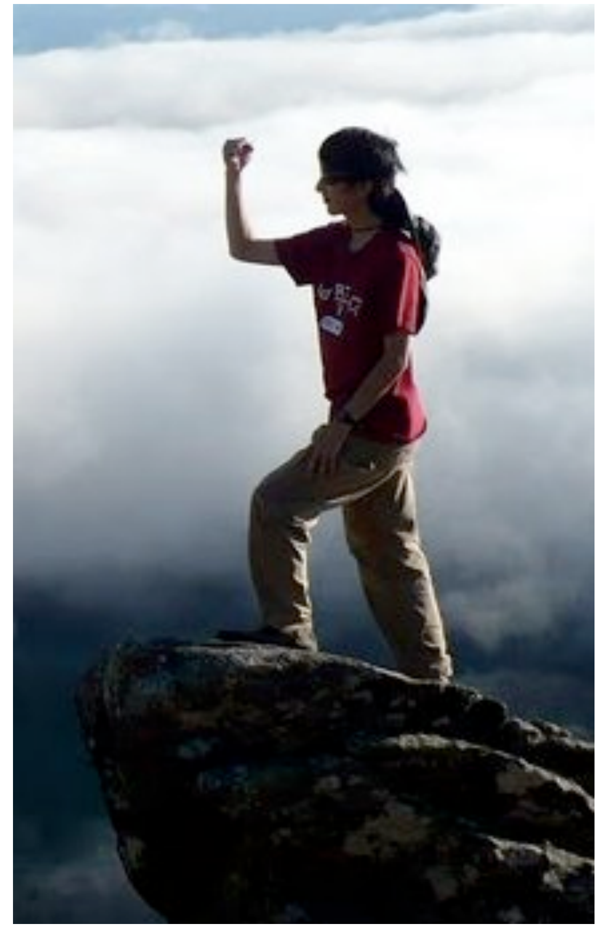
Principles

#lang class

Industrial co-op

Practice

Dan
Brown



Asumu
Takikawa

Nicholas
Labich

NORTHEASTERN UNIVERSITY
LVX
VERITAS
VIRTVS

Principles

#lang class

Industrial co-op

Practice

http://www.ccs.neu.edu/course/cs2510h/