Systematic Program Design (for Freshmen)

Matthias Felleisen (PLT)
Northeastern University, Boston

The Problem:
Computer Science1(and 2)

- in 1978 @ Karlsruhe (Technische Universität): variables, assignments, printing, arrays, loops, procedures, classes and methods

- in 1978 @ Karlsruhe (Technische Universität): variables, assignments, printing, arrays, loops, procedures, classes and methods
- in 2007 @ Anywhere (College, University): variables, assignments, printing, arrays, loops, procedures, classes and methods, ...

- in 1978 @ Karlsruhe (Technische Universität): variables, assignments, printing, arrays, loops, procedures, classes and methods
- in 2007 @ Anywhere (College, University): variables, assignments, printing, arrays, loops, procedures, classes and methods, ...
- ... and perhaps interfaces and inheritance.

- Algol 60/Simula 67
- Pascal
- C
- Scheme
- C++
- Eiffel
- Haskell
- Java
- Alice/Java

- Algol 60/Simula 67
- Pascal
- C
- Scheme
- C++
- Eiffel
- Haskell
- Java
- Alice/Java

8 languages, 30 years:

- Algol 60/Simula 67
- Pascal
- C
- Scheme
- C++
- Eiffel
- Haskell
- Java
- Alice/Java

8 languages, 30 years:

Are we really just a fashion industry?

- Algol 60/Simula 67
- Pascal
- C
- Scheme
- C++
- Eiffel
- Haskell
- Java
- Alice/Java

8 languages, 30 years:

Are we really just a fashion industry?

Can we do better?

- Programming: how do I create programs?
- Computing: how do programs compute?

- Programming: how do I create programs?
- Computing: how do programs compute?

- Systematic Design (problem solving)
- Functional Programming (middle school algebra)

- Programming: how do I create programs?
- Computing: how do programs compute?

- Systematic Design (problem solving)
- Functional Programming (middle school algebra)

... and a little bit of fun

;; Hello World (Lecture 1):

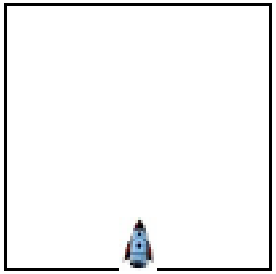
(define (image t)
  (place-image  50 (- 100 (+ t 10))
               (empty-scene 100 100)))

;; run program run:
(run-simulation image)

;; Hello World (Lecture 1):

(define (image t)
  (place-image  50 (- 100 (+ t 10))
                    (empty-scene 100 100)))

;; run program run:
(run-simulation image)



#| Shapes and Mouse Clicks (Lecture 18):

A Shape is one of:

-- a square;
-- a disk; or
-- one shape on top of another.

Design a program that determines whether a mouse click is inside some given Shape.
|#

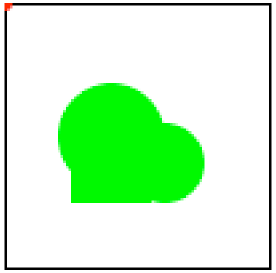
#| Shapes and Mouse Clicks (Lecture 18):

A Shape is one of:

-- a square;

-- a disk; or

-- one shape on top of another.



Design a program that determines whether a mouse click is inside some given Shape.

|#

The Context: Northeastern University and American College Culture

- CS 1: 160-200 students, CS 2: ~100-120
- three lectures/week @ 55-65 mins each
- one lab session/week @ 90 mins

- CS 1: 160-200 students, CS 2: ~100-120
- three lectures/week @ 55-65 mins each
- one lab session/week @ 90 mins

- two office hours/week/instructor
- 10 office hours/teaching assistants
- 20 office hours/tutors, 10 grading hours/tutor
- staff meeting: 90 mins/week; train the assistants; discuss students "with hope"

- one homework set/week
  - goal 1: prepare exam, weight ~20%
  - goal 2: **pair programming**
  - 3-10 problems; 2-4 are graded, randomly
- one quiz/meeting:
  - goal: reinforce daily learning (“keep up”)
  - 1/4 is graded, randomly
- two 3-hour exams/semester
  - goal: systematic design, not outcome
  - week 5 and week 10/11

- Homework projects:
  - goal: revisit projects across 4 weeks
  - hmwk 5: interactive graphical game
  - hmwk 7: ... fix in response to criticism
  - hmwk 9: ... use existing abstractions, create

- Homework projects:
  - goal: revisit projects across 4 weeks
  - hmwk 5: interactive graphical game
  - hmwk 7: ... fix in response to criticism
  - hmwk 9: ... use existing abstractions, create

- Northeastern is a co-op university
  - goal: prepare students for 3rd sem co-op
  - teach principles
  - ... and apply them to something they might encounter in industry

The Curriculum

How to Design Programs - systematic design - model of computation - functions - APIs/frameworks	Discrete Mathematics - sets - relations & functions - combinatorics - in progr. context
How to Design Classes - systematic design - classes/methods - standard API	How to Prove Programs - ideas to conjectures - 1st-order logic - ... with theorem proving - ... about HtDP code

Systematic Design:
The Recipe and Its Dimensions

What is Systematic Program Design

- design strategy:
  - step by step: from problems to solutions
  - iterative refinement: from core to full product

What is Systematic Program Design

- design strategy:
  - step by step: from problems to solutions
  - iterative refinement: from core to full product
- canonical outcomes:
  - problem description plus choice of strategy produces “normalized” results (alpha, beta, tests)

What is Systematic Program Design

- design strategy:
  - step by step: from problems to solutions
  - iterative refinement: from core to full product
- canonical outcomes:
  - problem description plus choice of strategy produces “normalized” results (alpha, beta, tests)
- continuous process:
  - small changes to problem lead to small changes in solutions in a predictable manner

Structural Design: Forms of Data

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

Process Steps

Domain Knowledge

Where CS Works

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.
data def							
purpose							
examples							
template							
code!							
test							

.....

describe the classes of "problem" data;

illustrate with examples

Structural Design

	atomic	en				rs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

state a “contract” aka
type or method signature;

summarize purpose of
program/function concisely

Structural Design

	atomic	enumer.	str			urs.	mut. rec.
data def							
purpose							
examples							
template							
code!							
test							

.....

formulate functional aka behavioral examples for the function/method

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.
data def							
purpose							
examples							
template							
code!							
test							

.....

take inventory of all the information you have about the function in the form of a “function template” or “function organization”

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

- does the data def. identify distinct sub-classes? how many?
- do any of the sub-classes describe compound data?
- do any of the clauses of the data def create self-reference

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

now code; that is, fill in the gaps in the template with functions that combine the values of the expressions

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

- deal with non-rec conditions
- then remind yourself (using the purpose statement and examples) what the existing expressions compute
- combine those computation
possibly “wishing” for mo

Structural Design

	atomic	enumer.	structs	hier.	union	recurs.	mut. rec.	.....
data def								
purpose								
examples								
template								
code!								
test								

now turn examples into
tests and run the test suite

Abstraction and Design

	"program editing"
data def purpose examples template code! test	Abstraction: higher-order data -- FPLs (e.g., Scheme) employ abstraction via parameters -- OOPLs (e.g., Java) use several forms of abstraction (abstract classes, generics, abstract traversals)

“Recursion”

<table>
<tr><td></td><td></td></tr>
<tr><td>data def

purpose

examples

template

code!

test</td><td>Generative Recursion

-- forms of recursion that don’t follow from the structure of the data def. (e.g., quick sort, adaptive integration)

-- OOPL: command pattern</td></tr>
</table>

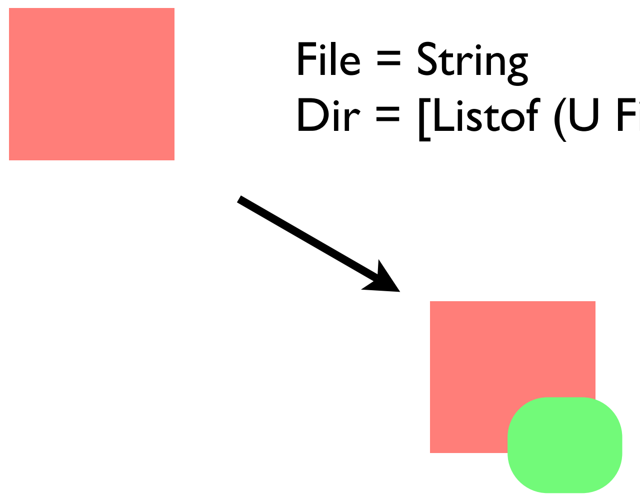
design attributes:

- accumulators:
  - strong induction
- stateful programming:
  - “circularity” of data; sharing
  - “poverty” of interface; efficiency

iterative refinement: programmers are scientists

File = String
Dir = [Listof (U File Dir)]

iterative refinement: programmers are scientists

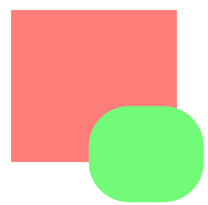


File = String
Dir = [Listof (U File Dir)]

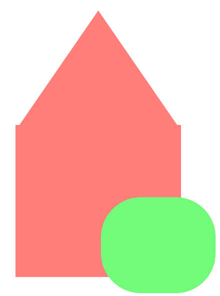
File = String
Dir = (make-dir String [Listof (U File Dir)])

iterative refinement: programmers are scientists

File = String
Dir = [Listof (U File Dir)]



File = String
Dir = (make-dir String [Listof (U File Dir)])



File = (make-file String Nat [Listof Char])
Dir = (make-dir String [Listof (U File Dir)])

Why Systematic Program Design

- teaching:
  - help students overcome an obstacle
  - train teaching assistant to intervene properly
  - empower students to learn on their own
  - grading "perfect" solutions

Why Systematic Program Design

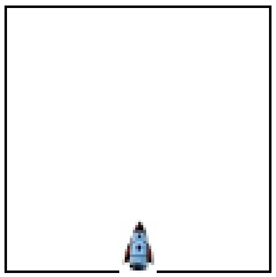
- “industrial” programming:
  - if the design strategy is transparent, program understanding and maintenance are easy
  - small changes to problems are common; everyone should be able to fix them
  - iterative design has become known as “agile” programming

Systematic Design: Some Examples

;; Hello World (Lecture 1):

(define (image t)
  (place-image  50 (- 100 (+ t 10))
                (empty-scene 100 100)))

;; run program run:
(run-simulation image)



How far did the horse buggy travel in one hour, if it leaves Pittsburgh at 10am on Monday, going 10 miles per hour?

t = ...	1	2	3	4
dist. =	10	20	30	40

How far did the horse buggy travel in one hour, if it leaves Pittsburgh at 10am on Monday, going 10 miles per hour?

t = ...	1	2	3	4
dist. =	10	20	30	40

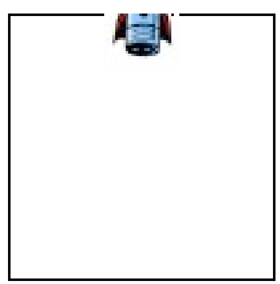
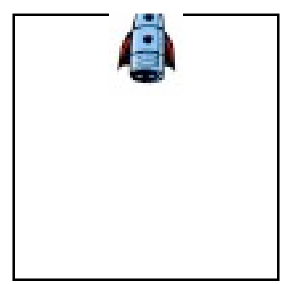
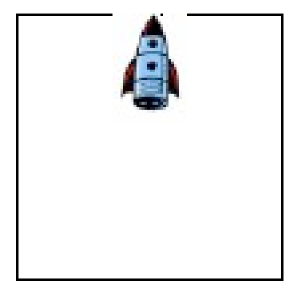
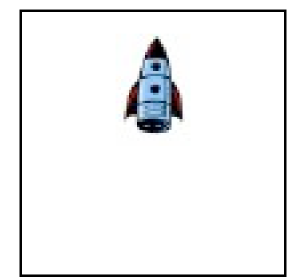
```
(define (dist t) (* 10 t))
```

Let’s make movies of rockets instead:

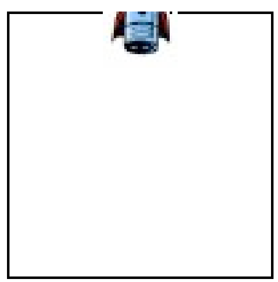
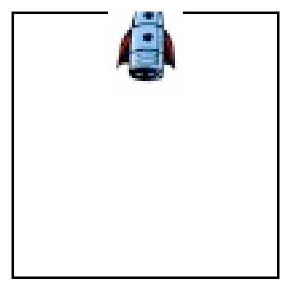
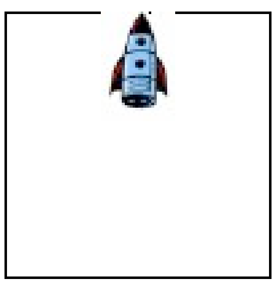
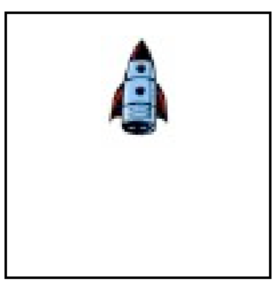
Problem:

Create a a series of images that show a rocket () descending from the top of the screen. The rocket descends at the constant rate of 10 pixels/clock tick, defying all laws of gravity.

Starting with tables:

t = ...	1	2	3	4
scene =				

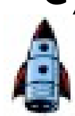
Starting with tables:

t = ...	1	2	3	4
scene =				

(define (image t)
  (place-image 
            50 (* 10 t)
             (empty-scene 100 100)))

Starting with tables:

It's all DOMAIN KNOWLEDGE!

```
(define (image  t)
  (place-image
```


```
               50 (* 10 t)
                (empty-scene 100 100)))
```

To make this fun:

(run-simulation *f*)

To make this fun:

(run-simulation *f*)

applies the function *f* to 0, 1, 2, 3, ...
(*f* 0), (*f* 1), (*f* 2), ... produces images ...
run-simulation displays these images
at the rate of 28 per second (which
we call a "clock tick")

More fun:

```
(big-bang w0 ...
   ;; creates the world, making w0 the initial world
 (on-tick tock) ; installs a "clock tick" handler
 (on-key clack) ; installs a keyboard event handler
 (on-mouse click) ; installs a mouse event handler
 (on-draw render) ; helps render the world as a scene
)
```

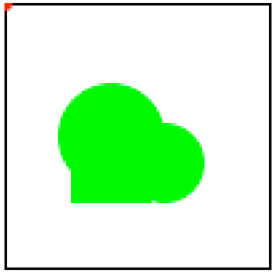
#| Shapes and Mouse Clicks (Lecture 18):

A Shape is one of:

-- a square;

-- a disk; or

-- one shape on top of another.



Design a program that determines whether a mouse click is inside some given Shape.

|#

data definition (English plus data sub-language):

```
(define-struct disk (radius center))
(define-struct square (size ul))
(define-struct over (top bot))

#| A Shape is one of:
    -- (make-square Nat Posn)
    -- (make-disk Nat Posn)
    -- (make-over Shape Shape)
   Interpretation:
    (make-disk r p) is a green disk with center p ...
|#
```

data definition & **data examples**:

```
(define-struct disk (radius center))
(define-struct square (size ul))
(define-struct over (s1 s2))

#| A Shape is one of:
   -- (make-square Nat Posn)
   -- (make-disk Nat Posn)
   -- (make-over Shape Shape)
|#

(define c (make-disk 20 (make-posn 40 50)))
(define s (make-square 30 (make-posn 40 60)))
(define t (make-over s c))
(define q (make-over t (make-disk 15 ...)))
```

signature and purpose statement:

```
;; Shape Posn -> Boolean
;; is position p contained in this shape s?

(define (in? s p) false)
```

functional (behavioral) examples:

```
(define c (make-disk 20 (make-posn 40 50)))
(define s (make-square 30 (make-posn 40 60)))
(define t (make-over s c))

;; Shape Posn -> Boolean
;; does this shape s contain position p?
;; examples:
;;   (make-posn 45 45) is in shape t (why?)
;;   (make-posn 200 200) is NOT in shape t (why?)

(define (in? s p) false)
```

functional examples **as tests**:

```
(define c (make-disk 20 (make-posn 40 50)))
(define s (make-square 30 (make-posn 40 60)))
(define t (make-over s c))

;; Shape Posn -> Boolean
;; does this shape s contain position p?

(check-expect (in? t (make-posn 45 45)) true)
(check-expect (in? t (make-posn 100 100)) false)

(define (in? s p) false)
```

template creation: how many sub-classes are there?

```
#| A Shape is one of:
    -- (make-square Nat Posn)
    -- (make-disk Nat Posn)
    -- (make-over Shape Shape)
|#

(define (in? s p)
  (cond
    [... ...]
    [... ...]
    [... ...]))
```

template creation: how many sub-classes are there?

```
#| A Shape is one of:
    -- (make-square Nat Posn)
    -- (make-disk Nat Posn)
    -- (make-over Shape Shape)
|#

(define (in? s p)
  (cond
    [(square? s) ...]
    [(disk? s)   ...]
    [(over? s)   ...]))
```

template creation: how many are compound data?

```
(define-struct over (top bot))
#| A Shape is one of:
    -- Square
    -- Disk
    -- (make-over Shape Shape)
|#

(define (in? s p)
  (cond
    [(square? s) ...]
    [(disk? s)    ...]
    [(over? s)    ... (over-top s) (over-bottom s) ...]))
```

template creation: are any clauses in the DD recursive

```
#| A Shape is one of:
    -- Square
    -- Disk
    -- (make-over Shape Shape)
|#

(define (in? s p)
  (cond
    [(square? s) ...]
    [(disk? s)    ...]
    [(over? s)    ... (in? (over-top s) p)
                  ... (in? (over-bottom s) p) ...]))
```

let's code: start with non-recursive cases
use examples, make wishes

```
(check-expect (in? (make-disk ...) (make-posn ..)) ...)

(define (in? s p)
  (cond
    [(square? s) (in-square? s p)]
    [(disk? s)   (in-disk? s p)]
    [(over? s)   ... (in? (over-top s) p)
                 ... (in? (over-bottom s) p) ...]))
```

let's code: start with non-recursive cases
use examples, make wishes

```
(check-expect (in? (make-disk ...) (make-posn ..)) ...)

(define (in? s p
  (cond
    [(square? s)
    [(disk? s)
    [(over? s)    ... (in? (over-top s) p)
                  ... (in? (over-bottom s) p) ...]))
```

wish list:

in-square? : Square Posn -> Boolean

does this square contain the position

...

let's code: what do the expressions in the recursive cases compute? use the purpose statement

```
;; does this shape s contain position p?

(define (in? s p)
  (cond
    [(square? s) (in-square? s p)]
    [(disk? s)   (in-disk? s p)]
    [(over? s)   ... (in? (over-top s) p)
                 ... (in? (over-bottom s) p) ...]))
```

let's code: what do the expressions in the recursive cases compute? use the purpose statement

;; does this shape s contain position *p*?

```
(define (in? s p)
  (cond
    [(square? s) (in-square? s p)]
    [(disk? s)
    [(over? s)    ... (in? (over-top s) p)
                  ... (in? (over-bottom s) p) ...]))
```

does the *top* part of *s* contain *p* ?

let's code: what do the expressions in the recursive cases compute? use the purpose statement

;; does this shape s contain position *p*?

```
(define (in? s p)
  (cond
    [(square? s) (in-square? s p)]
    [(disk? s)   (in-disk? s p)]
    [(over? s)   ..
                 ... (in? (over-bottom s) p) ...]))
```

does the *bot* part of *s* contain *p* ?

let's code: combine the results with an existing primitive or make a wish for a function that combines the results properly

```
(define (in? s p)
  (cond
    [(square? s) (in-square? s p)]
    [(disk? s)    (in-disk? s p)]
    [(over? s)    (or (in? (over-top s) p)
                      (in? (over-bottom s) p))]))
```

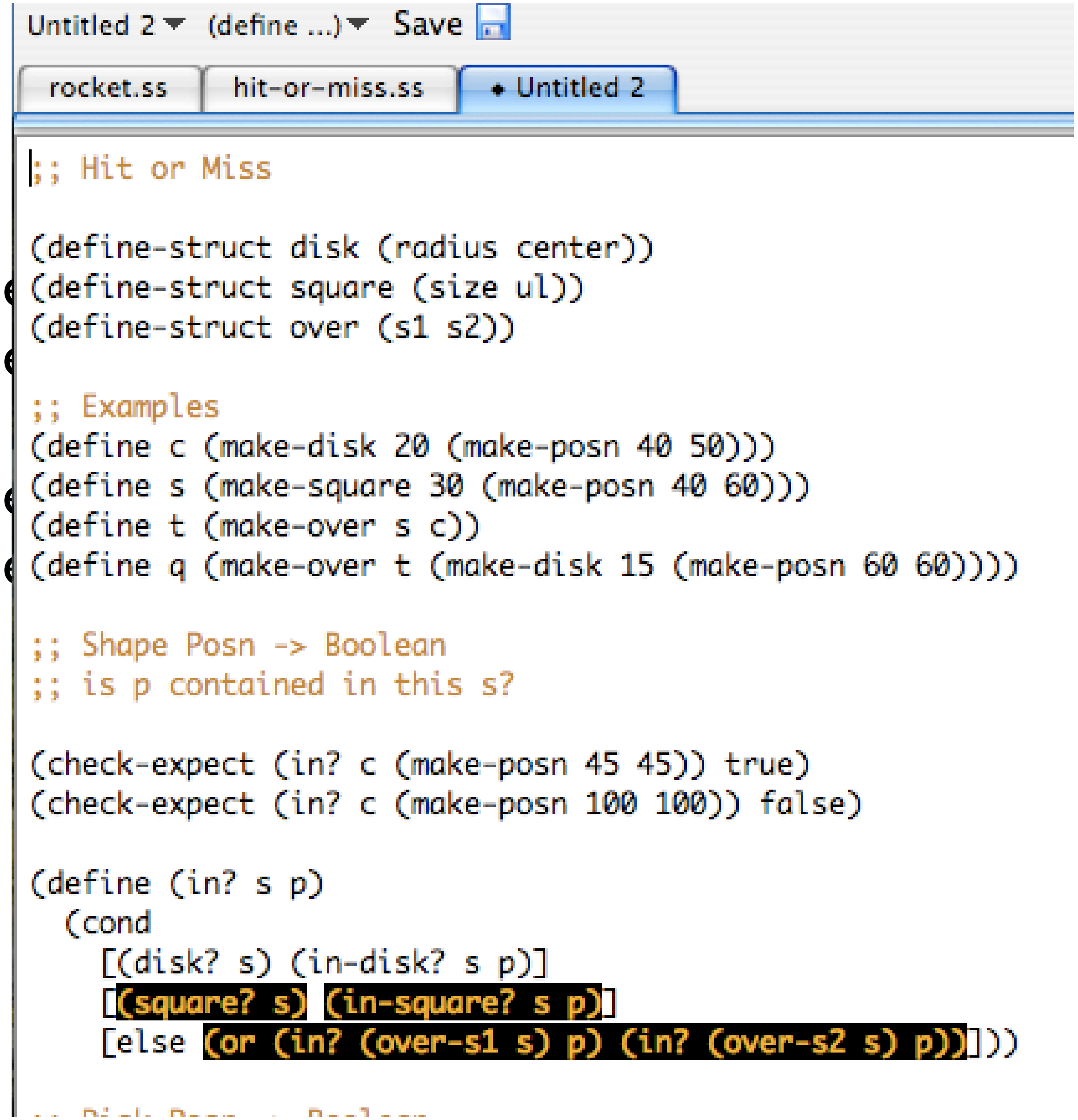
let's code: combine the results with an existing primitive or make a wish for a function that combines the results properly

```
(define (in? s p)
  (cond
    [(square? s) (in
    [(disk? s)     (in
    [(over? s)     (or (in? (over-top s) p)
                       (in? (over-bottom s) p))]))
```

When students are stuck, we work through examples with them, creating tables.

test!

(check-expe
(check-expe
...
(check-expe
(check-expe
...




```
;; Hit or Miss

(define-struct disk (radius center))
(define-struct square (size ul))
(define-struct over (s1 s2))

;; Examples
(define c (make-disk 20 (make-posn 40 50)))
(define s (make-square 30 (make-posn 40 60)))
(define t (make-over s c))
(define q (make-over t (make-disk 15 (make-posn 60 60))))

;; Shape Posn -> Boolean
;; is p contained in this s?

(check-expect (in? c (make-posn 45 45)) true)
(check-expect (in? c (make-posn 100 100)) false)

(define (in? s p)
  (cond
    [(disk? s) (in-disk? s p)]
    [(square? s) (in-square? s p)]
    [else (or (in? (over-s1 s) p) (in? (over-s2 s) p))]))
```

wiring it all up ...

```
(define-struct world (sh ms))
;; World = (make-posn Shape Posn)
;; interpretation: the displayed shape and the last mouse click

(big-bang
        (make-world q (make-posn 0 0))
        (on-mouse (lambda (w x y me)
                      (if (symbol=? 'button-down me)
                          (make-world (world-sh w) (make-posn x y))
                          w)))
        (on-draw (lambda (w)
                     (scene+dot (world-ms w)
                                (scene+shape (empty-scene 200 200)
                                             (world-sh w))))))
```

... and it is NOT about functional programming:

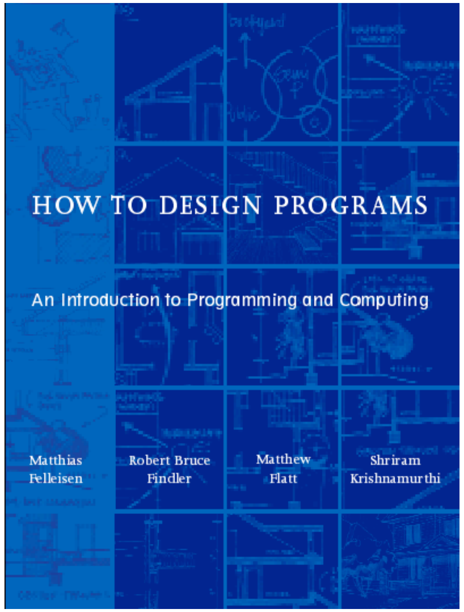
- Java: start with a class diagram
  - **interface** *IShape*
  - **class**(es) *Disk*, *Square*, *Over* **extends** *IShape*
  - boolean in(Posn p) // does *this* shape contain *p*?
  - follow the arrows, don't chase beyond neighbors
- ... yields ***true*** OO designs, aka "design patterns"
- ... though clashes with ***bad*** OOPL implementations

Evaluation:
Students, Colleagues, Industry

HtDP: 13 years, HtDC: 4 years

What we also have:

- the DrScheme IDE
- teaching languages
- text book (MIT Press)
- adapted to Java
- connected to logical reasoning



- HtD{P|C}: hand-and-overs to other instructors
- HtDP evaluations:
  - comparative test at high school
  - evaluation wrt Rice C++ course(s)
  - HtDP/C at NEU
  - Bootcamp for middle schools

- HtD{P|C}: hand-and-overs to other instructors
- HtDP evaluations:
  - comparative test at high school
  - evaluation wrt Rice C++ course(s)
  - HtDP/C at NEU
  - Bootcamp for middle schools

... and yet no amount of evaluation convinces anybody who wishes to teach conventional C++/Java courses

Alief School District: Year 1

Same Teacher

Alief School District: Year 1

Class 1

Class 2

Class 3

Same Teacher

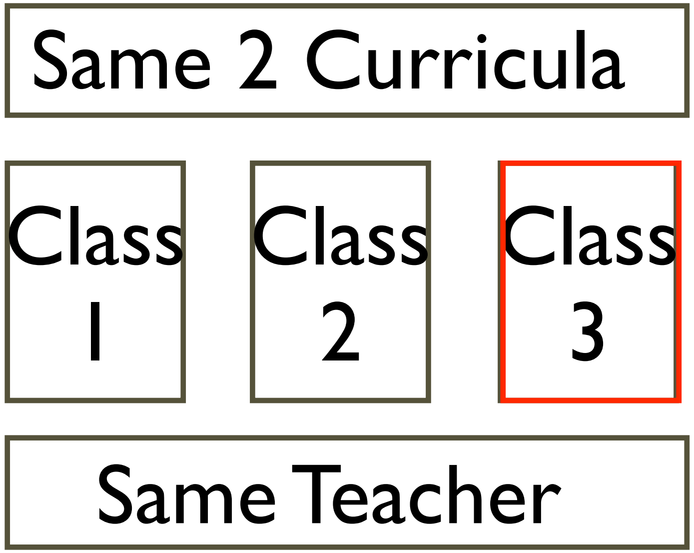
Alief School District: Year 1

Same 2 Curricula

Class 1	Class 2	Class 3

Same Teacher

Alief School District: Year 1



Alief School District: Year 1

Same 2 Curricula

Class 1 | Class 2 | Class 3

Same Teacher

- All students: HtDP preferred by ~70%

Alief School District: Year 1

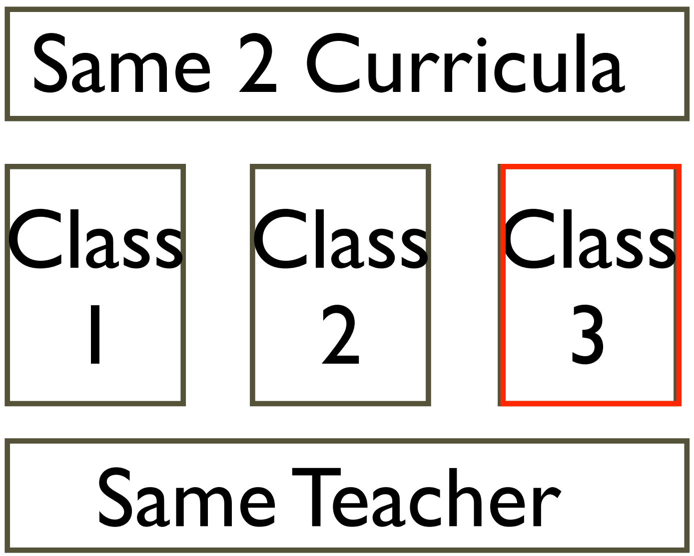
Same 2 Curricula

Class 1	Class 2	Class 3

Same Teacher

- All students: HtDP preferred by ~70%
- The more C++, the more they prefer HtDP

Alief School District: Year 1



- All students: HtDP preferred by ~70%
- The more C++, the more they prefer HtDP
- Female students: prefer HtDP by a ratio over 4:1

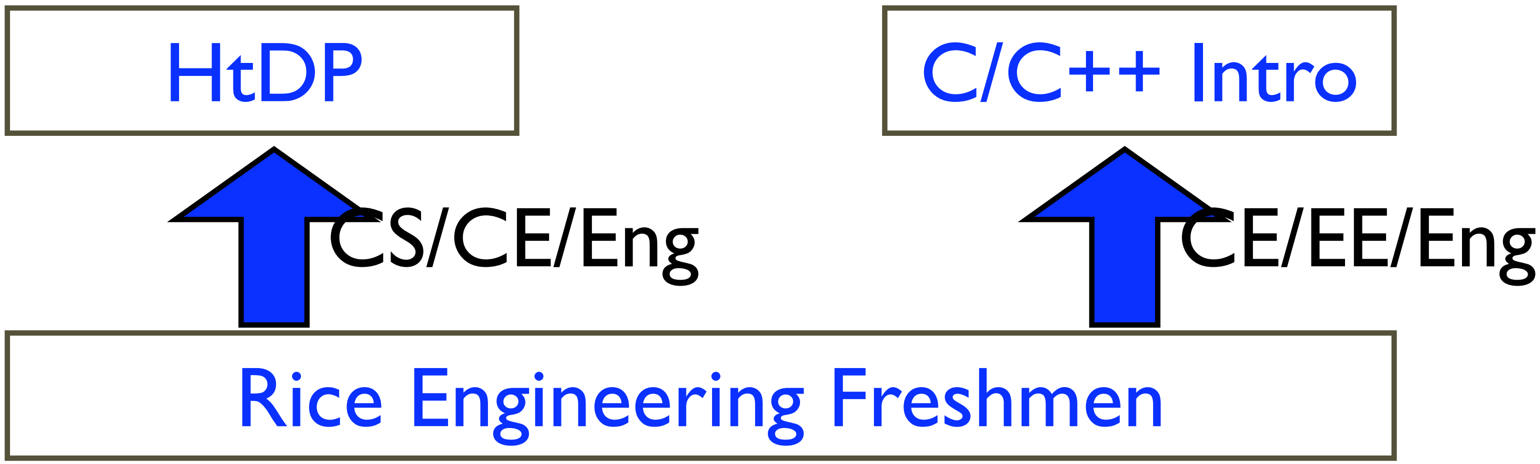
Independent NSF Evaluation:

- 300 high school teachers over 4 years
- when implemented, significant AP improvement
- 90-95%: "this approach changed my mind", best CS introduction ever

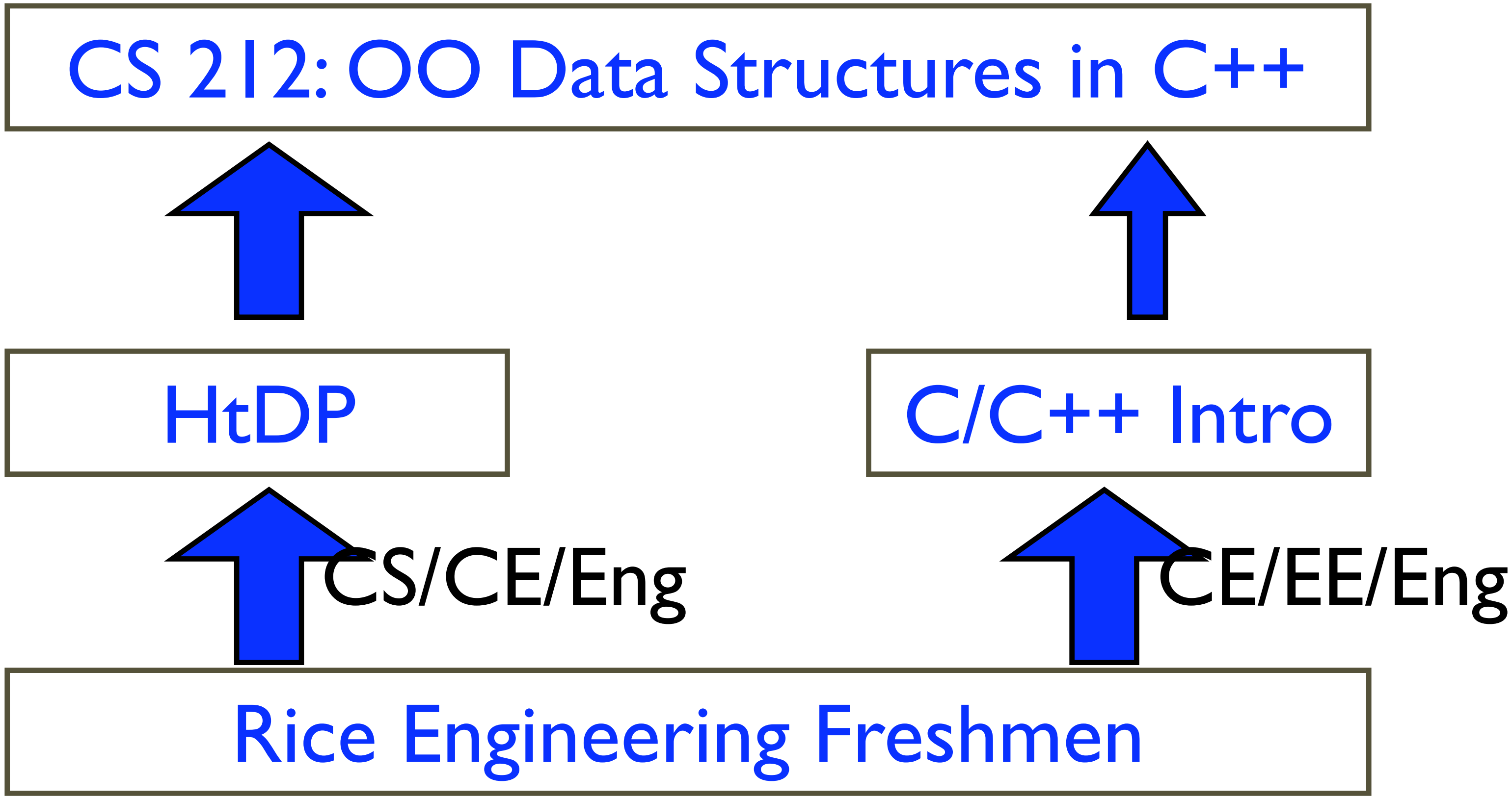
Rice: Year 1

Rice Engineering Freshmen

Rice: Year 1

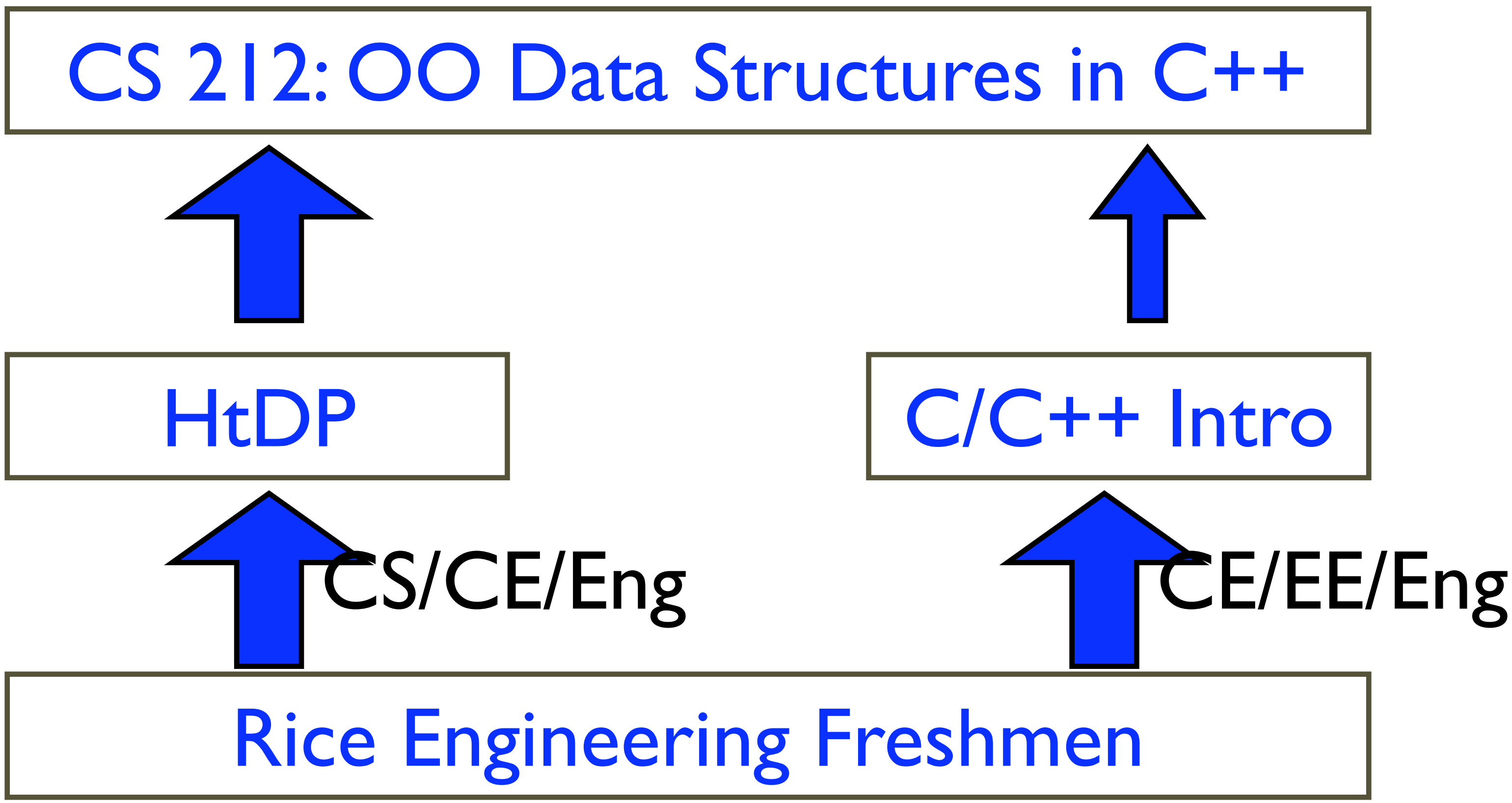


Rice: Year 1



CS 212: OO Data Structures in C++
HtDP
C/C++ Intro
CS/CE/Eng
CE/EE/Eng
Rice Engineering Freshmen

Rice: Year 1



HtDP Students routinely outperform C/C++ students on C++

Northeastern:

Before:

- conventional 1-year OOP (C++, Java)
- co-op students: 2/3 in "tech support"
- co-op employers routinely complain
- down-stream faculty has given up
- faculty retreats on "programming skills"

Northeastern:

After:

- systematic program design (+ models)
- co-op students: 2/3 and more in programming
- ... and "better" employers (MS, Google, Amazon)
- co-op employers praise UGs, complain about MS
- down-stream faculty routinely praise skills (loops)
- graduate dean wishes to "lift" curriculum to MS

Bootstrap:

US School System

High School
(9-12)

Middle School
(5-8)

Elementary School
(K, 1- 4)

Bootstrap:

US School System

High School
(9-12)

Middle School
(5-8)

Elementary School
(K, 1- 4)

Bootstrap:

- after-school
- citizen teachers
- “poor” districts
- 9 weekly meetings
- the “math effect”

Evaluations are Irrelevant

- Observation 1: high school teachers don't change their mind based on evaluations
- Observation 2: college instructors don't change their mind based on evaluations
- Observation 3: colleagues at universities insist on fashions, regardless of evaluations

Evaluations are Irrelevant

- Observation 1: high school teachers don't change their mind based on evaluations
- Observation 2: college instructors don't change their mind based on evaluations
- Observation 3: colleagues at universities insist on fashions, regardless of evaluations

Conclusion: people who demand evaluations wish to stop discussion

Conclusions, Future

Combining principles with pragmatics in the first year is

Combining principles with pragmatics in the first year is

- ... feasible

Combining principles with pragmatics in the first year is

- ... feasible
- ... effective

Combining principles with pragmatics in the first year is

- ... feasible
- ... effective
- ... productive

Combining principles with pragmatics in the first year is

- ... feasible
- ... effective
- ... productive
- It is the ***right*** thing!

Design Principles

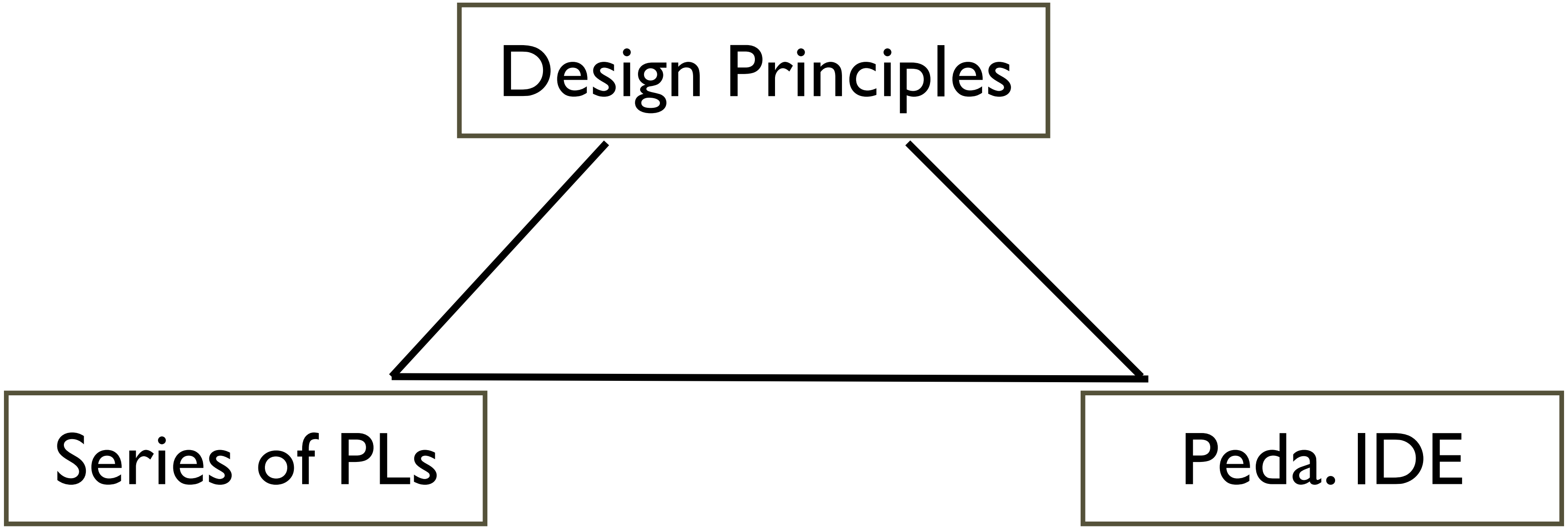
Design Principles

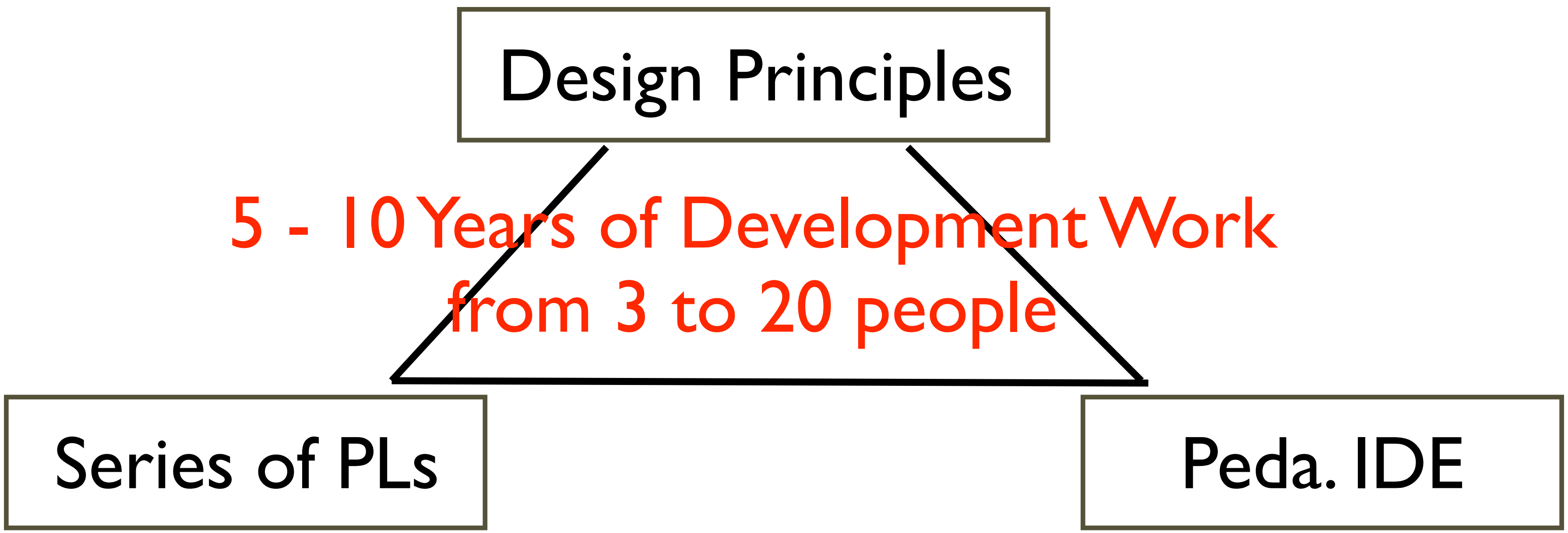
Series of PLs

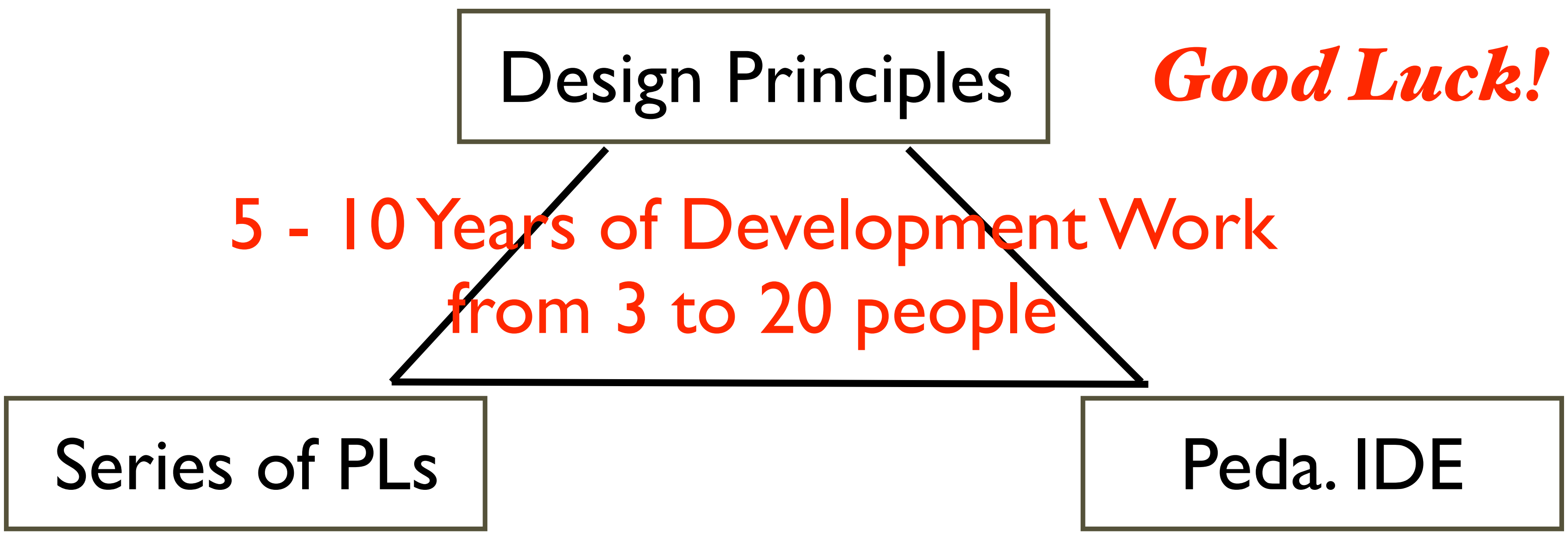
Design Principles

Series of PLs

Peda. IDE


Design Principles
Series of PLs
Peda. IDE


Design Principles
5 - 10 Years of Development Work
from 3 to 20 people
Series of PLs
Peda. IDE


Design Principles
Good Luck!
5 - 10 Years of Development Work
from 3 to 20 people
Series of PLs
Peda. IDE

	Programming	Mathematics
Semester 1	*How to Design Programs*	Discrete
Semester 2	*How to Design Classes*	*How to Prove Programs (ACL2)*

Thank You!

Matthew Flatt
Robert Findler
Shriram Krishnamurthi
Eli Barzilay
John Clements
Kathi Fisler
Kathy Gray
Emmanuel Schanzer
Viera Proulx
and many, many more