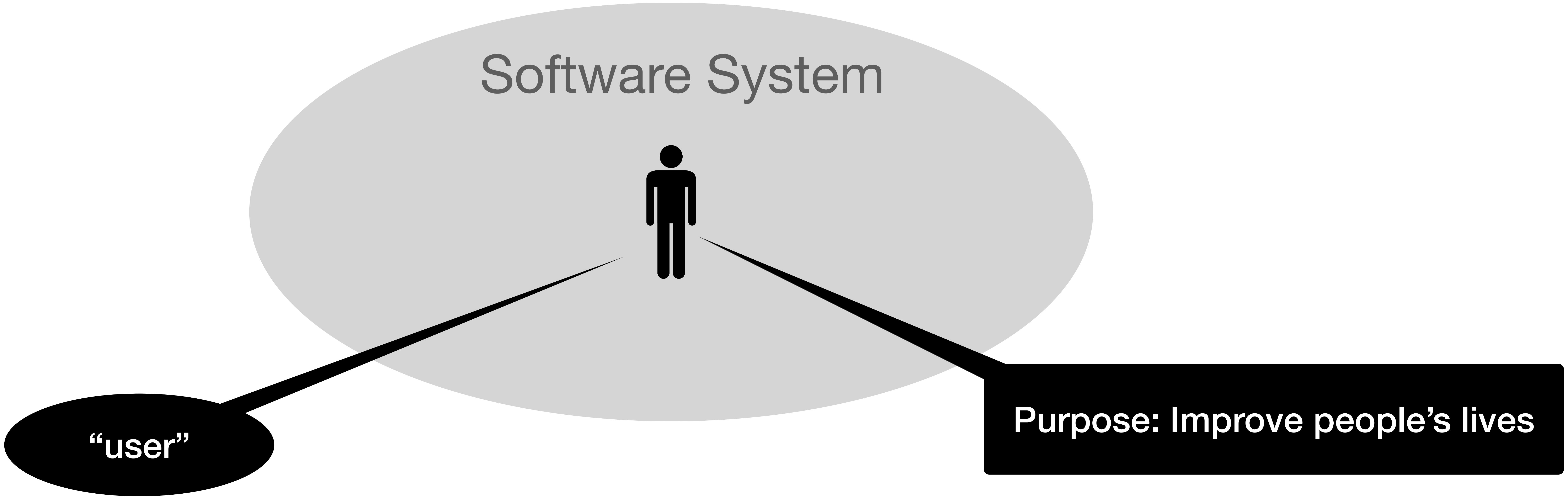
Socially Responsible Software Development

Matthias Felleisen, PLT

I, Me, Myself

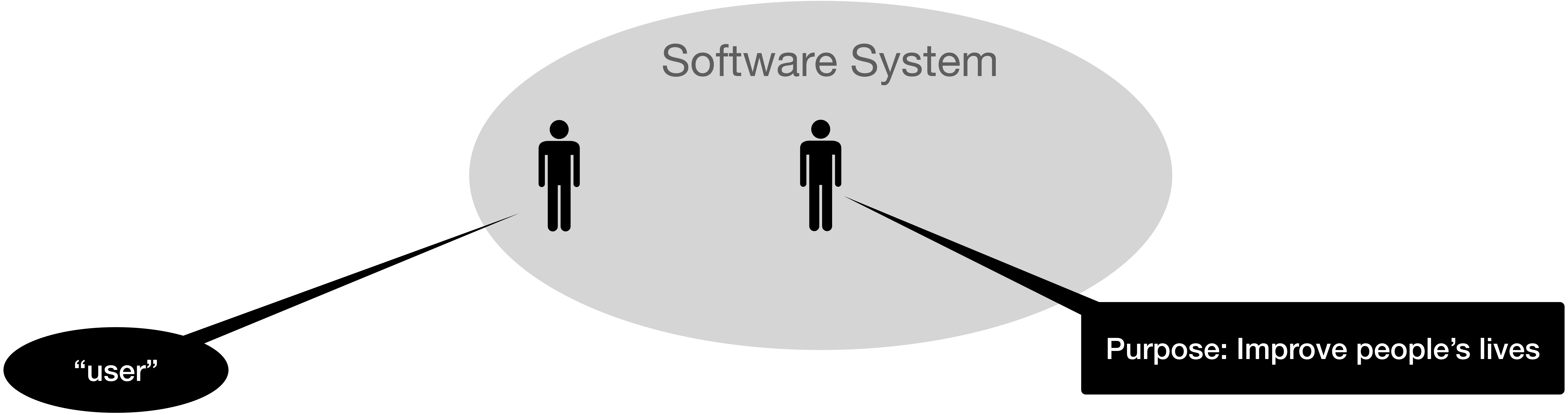
- programming language researcher
- … who cares about *programming*
- founded PLT, which is behind the Racket language
- created alternative programming curriculum (K12, freshman)
- TeachScheme! ~> Bootstrap outreach (20-30K students per year)
- maintained student-facing sw (appr. 50-80 Kloc) for ~28 years
- developed a software development curriculum for ~25 years



Software System

“user”

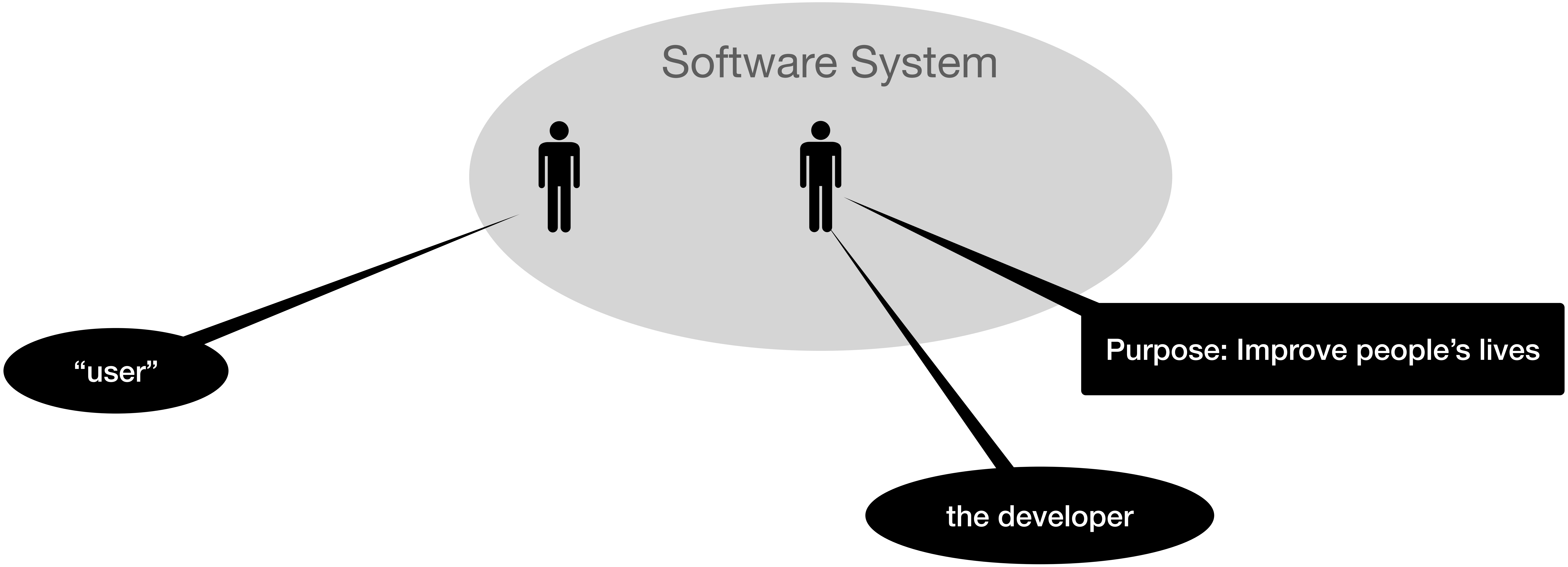
Purpose: Improve people’s lives



Software System

“user”

Purpose: Improve people’s lives



Software System

“user”

the developer

Purpose: Improve people’s lives

Software System

Software System

Software System
Software System
Must read and comprehend old code

Software System
Software System

Must read and comprehend old code

The maintainer.

Software System

Software System

Must read and comprehend old code

Older version of “you”

The maintainer.

Software System
Software System

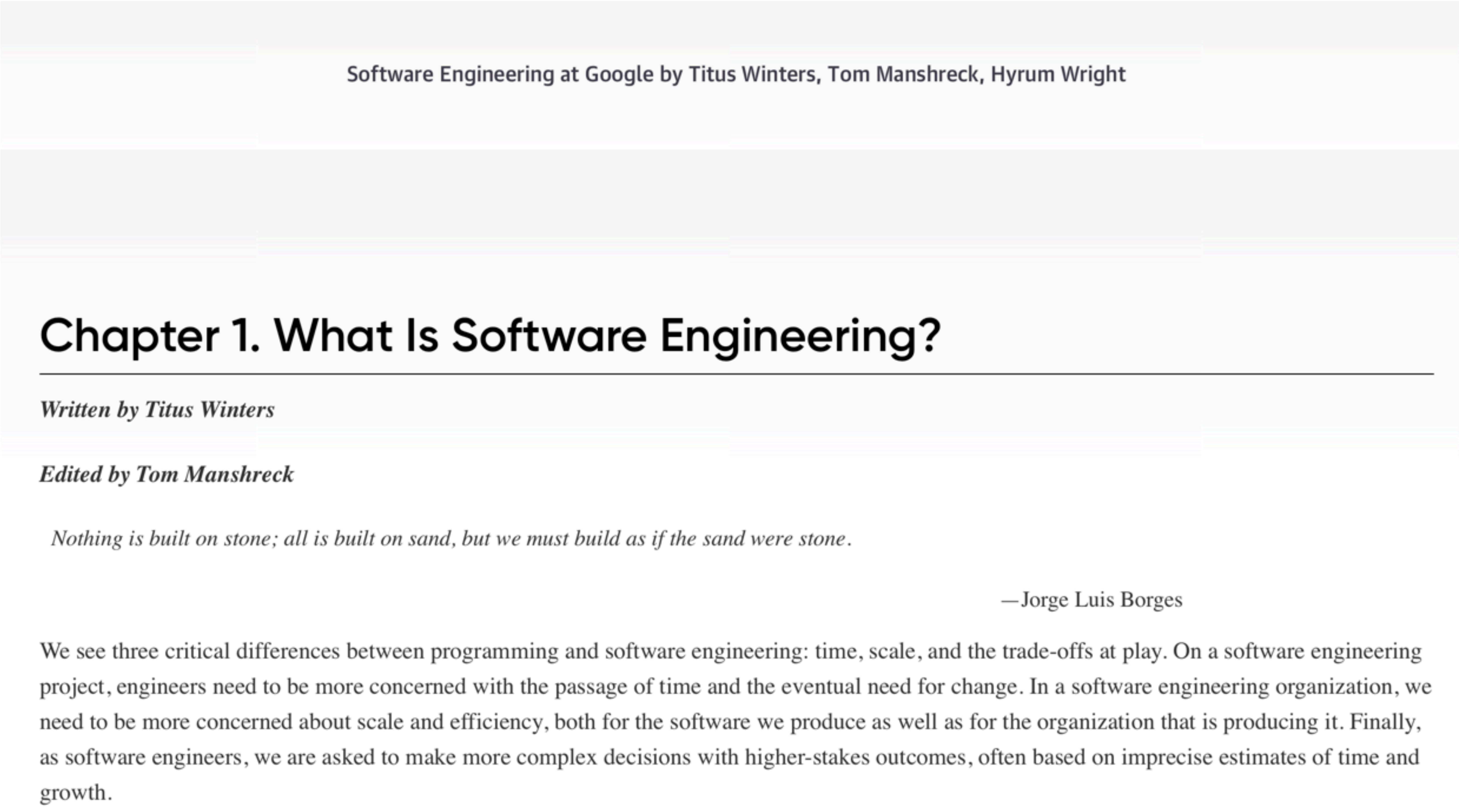
Think of the recipient, always!

Must read and comprehend old code

Older version of “you”

The maintainer.

Preaching to the Choir



Software Engineering at Google by Titus Winters, Tom Manshreck, Hyrum Wright

Chapter 1. What Is Software Engineering?

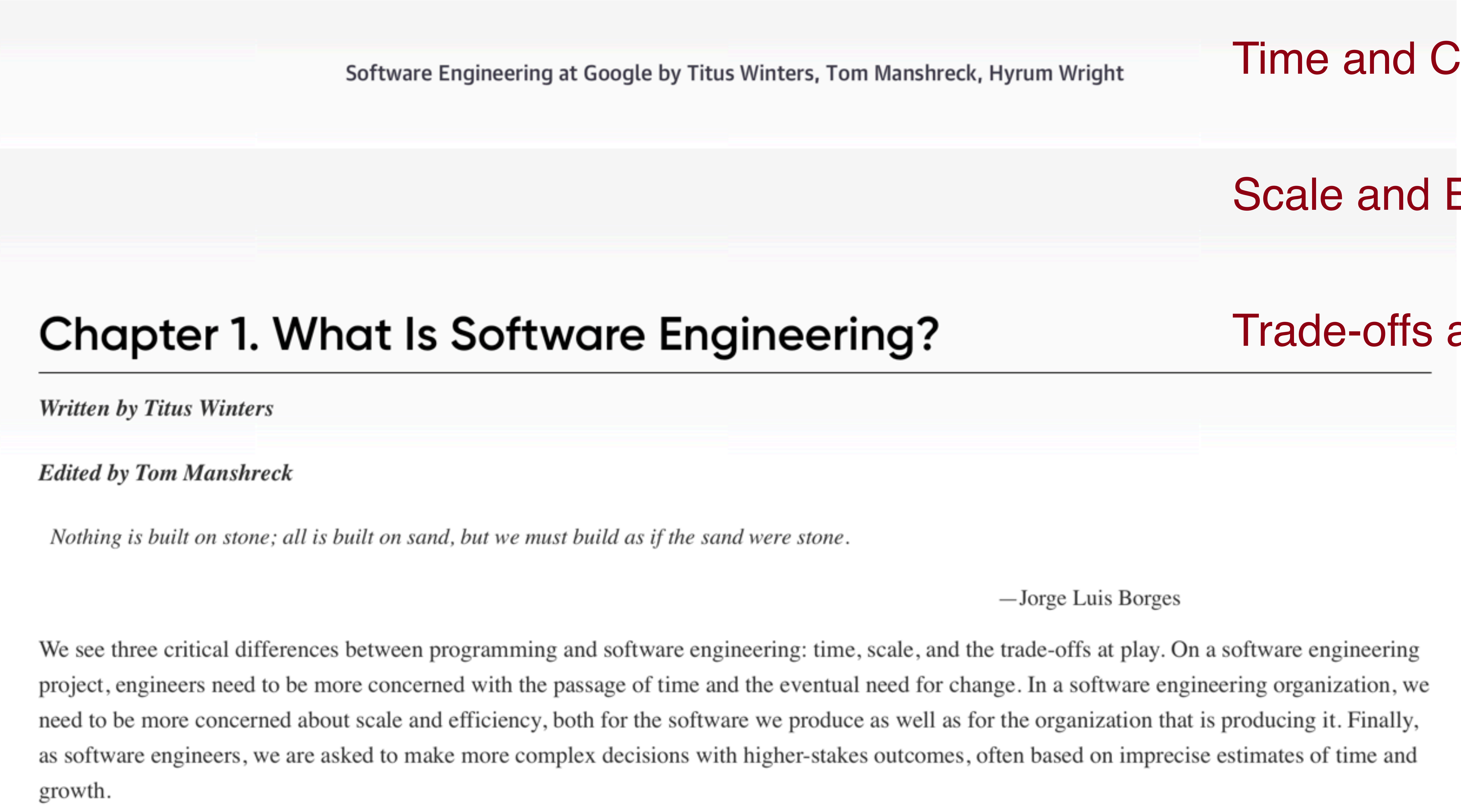
Written by Titus Winters

Edited by Tom Manshreck

> *Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.*
>
> —Jorge Luis Borges

We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

Preaching to the Choir



Software Engineering at Google by Titus Winters, Tom Manshreck, Hyrum Wright

Chapter 1. What Is Software Engineering?

Written by Titus Winters

Edited by Tom Manshreck

> *Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.*
>
> —Jorge Luis Borges

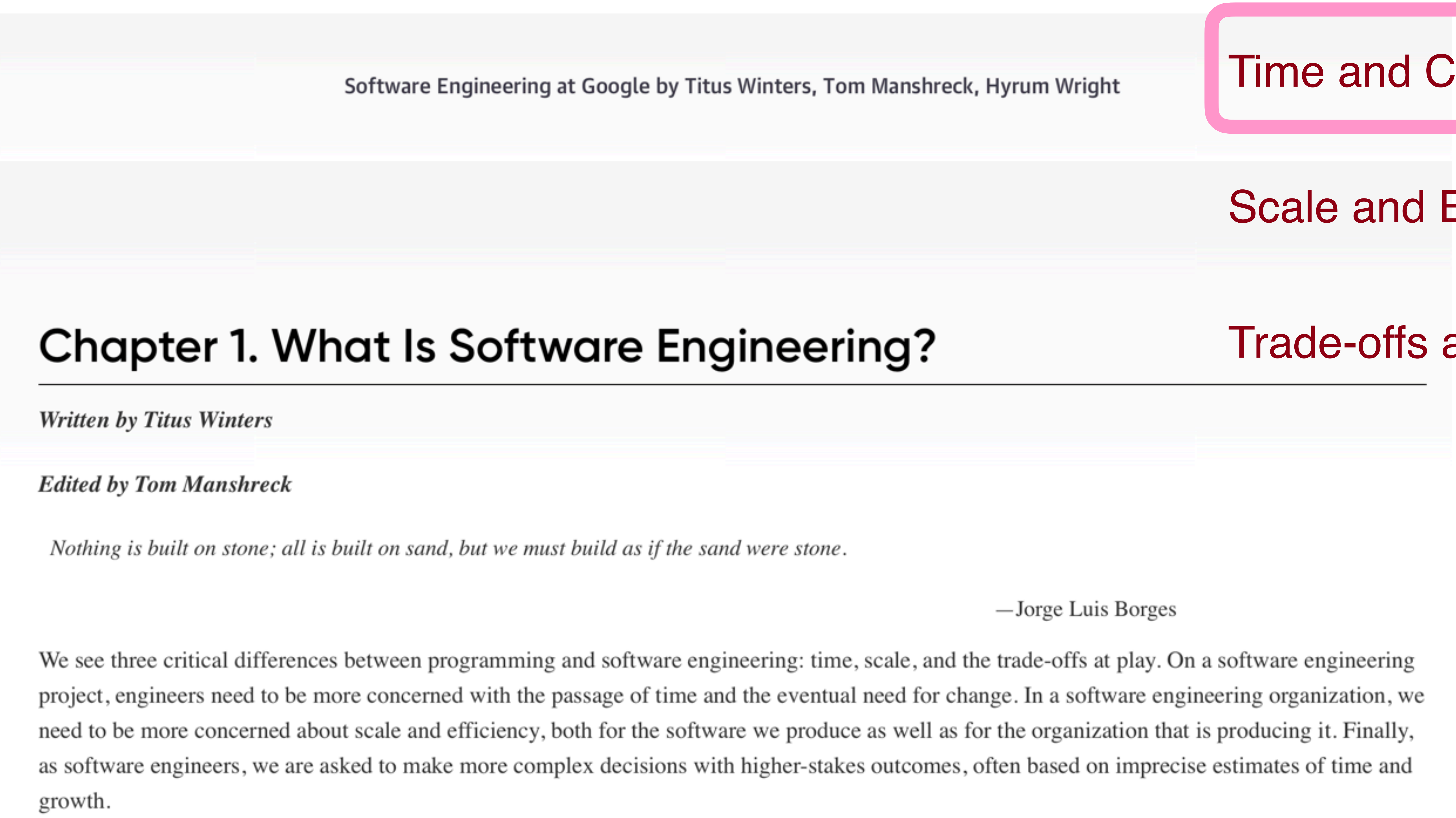
We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

Time and Change

Scale and Efficiency

Trade-offs and Costs

Preaching to the Choir



Software Engineering at Google by Titus Winters, Tom Manshreck, Hyrum Wright

Chapter 1. What Is Software Engineering?

Written by Titus Winters

Edited by Tom Manshreck

> *Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.*
>
> —Jorge Luis Borges

We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

Time and Change

Scale and Efficiency

Trade-offs and Costs

Challenges

How should universities and colleges prepare students for software development properly?

Challenges

How should universities and colleges prepare students for software development properly?

How should industry identify developers with the proper understanding of software?

Challenges

grind leetcode

Programming 101

+

Data Structures & Algo.

$$$

Challenges

grind leetcode

Programming 101

+

Data Structures & Algo.

$$$

How does this process get socially responsible software developers into the right place?

Preaching to the Choir, Again

an internal google email

Challenges, A Solution

I have spent the last > 25 years working on an alternative curriculum to make sure students “get” what software really is and how to do it right.

Challenges, A Solution

I have spent the last > 25 years working on an alternative curriculum to make sure students “get” what software really is and how to do it right.

What have *you* done?

Summary

Students must learn to:

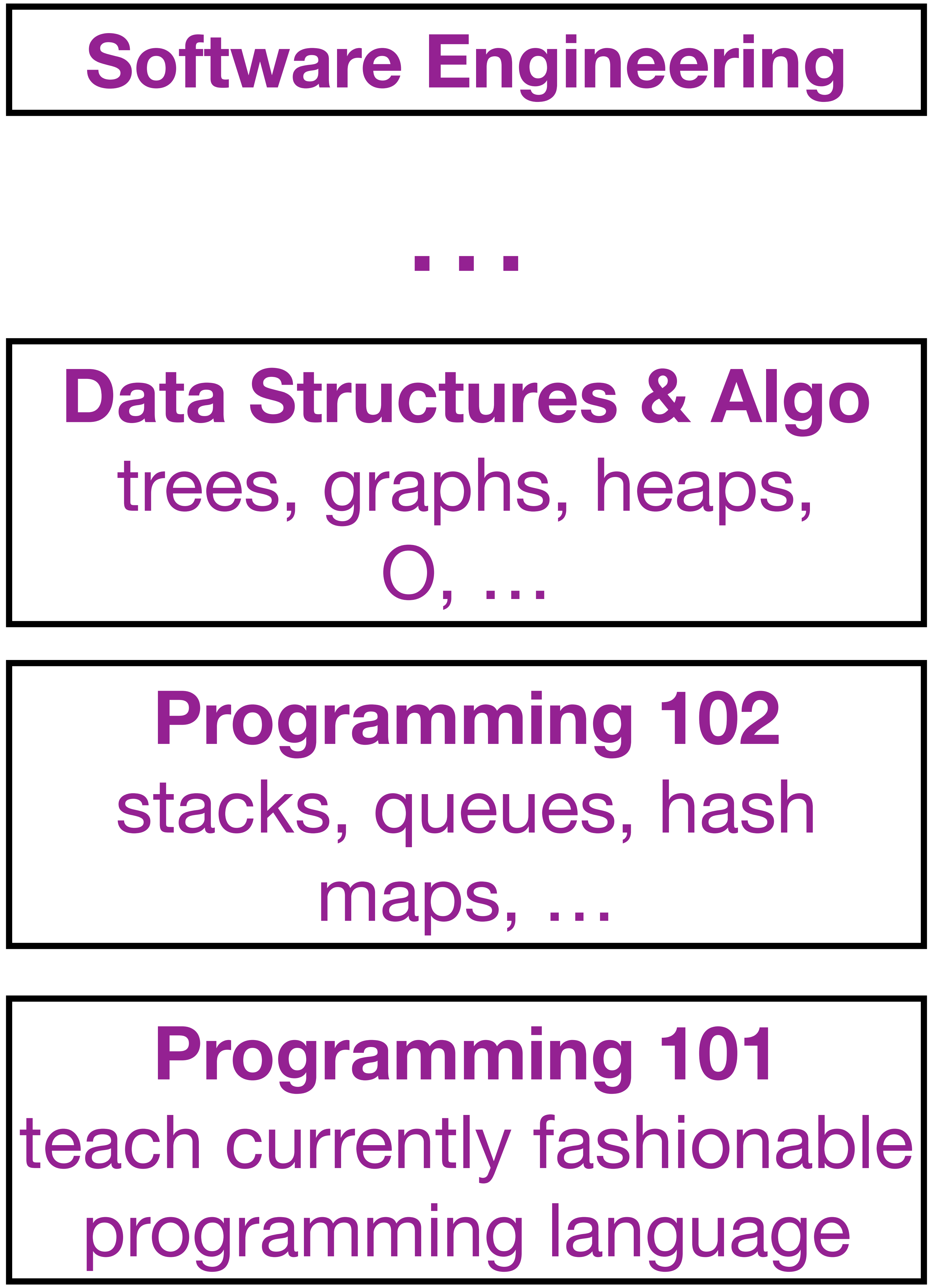
1. Program systematically.
2. Program in pairs.
3. Program with different partners.
4. Program revisions of code.
5. Program revisions of code that isn't theirs.
6. Program "large" systems.
7. Program systematically under stress.
8. Present programs to their peers, regularly and frequently.
9. Review and critique programs of peers, regularly and frequently.

It would be great if industry signaled support for this change.

The Programming Curriculum

The Programming Systematically Curriculum

Curriculum: Traditional vs Sw Dev



Software Engineering

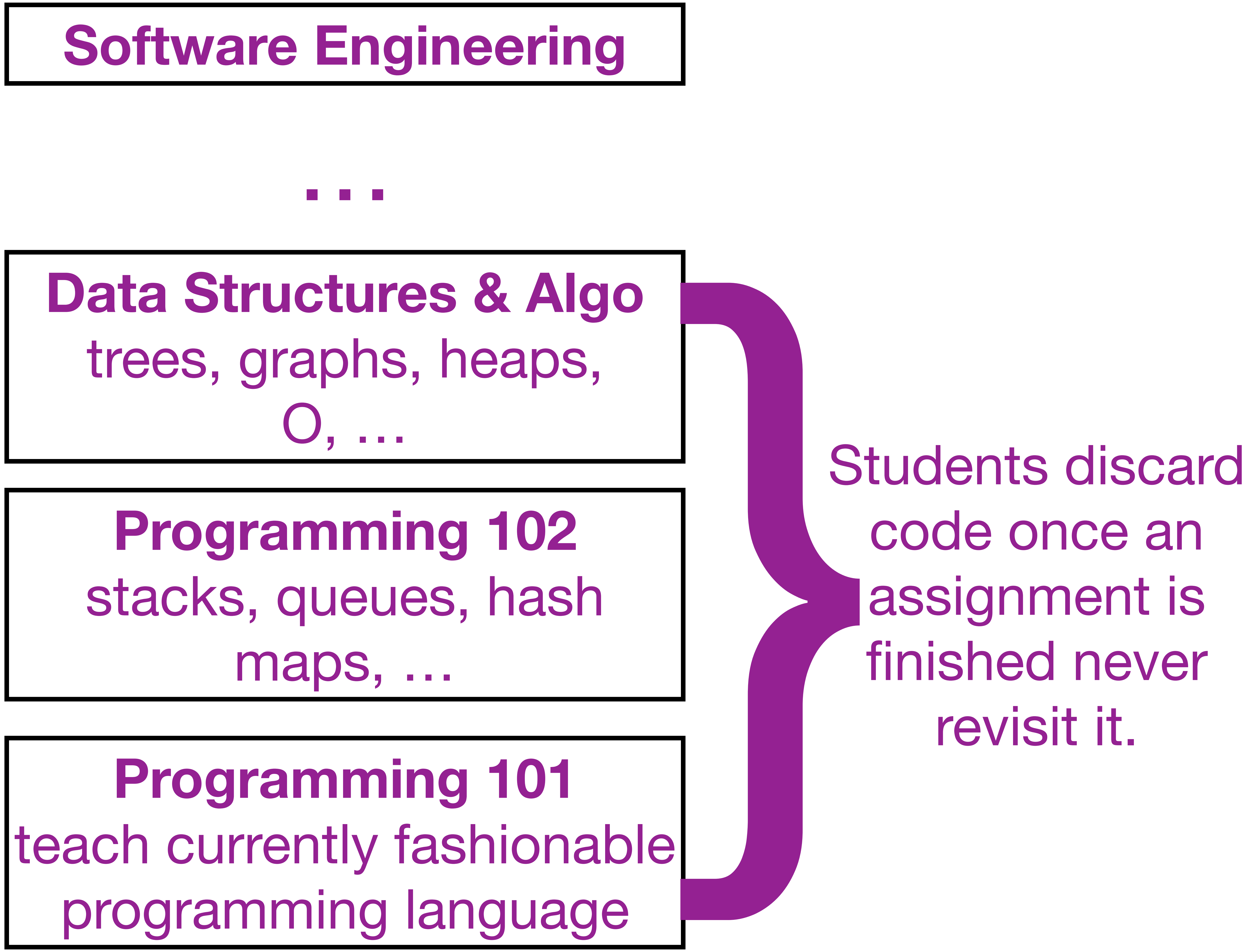
…

Data Structures & Algo
trees, graphs, heaps, O, …

Programming 102
stacks, queues, hash maps, …

Programming 101
teach currently fashionable programming language

Curriculum: Traditional vs Sw Dev



Software Engineering

…

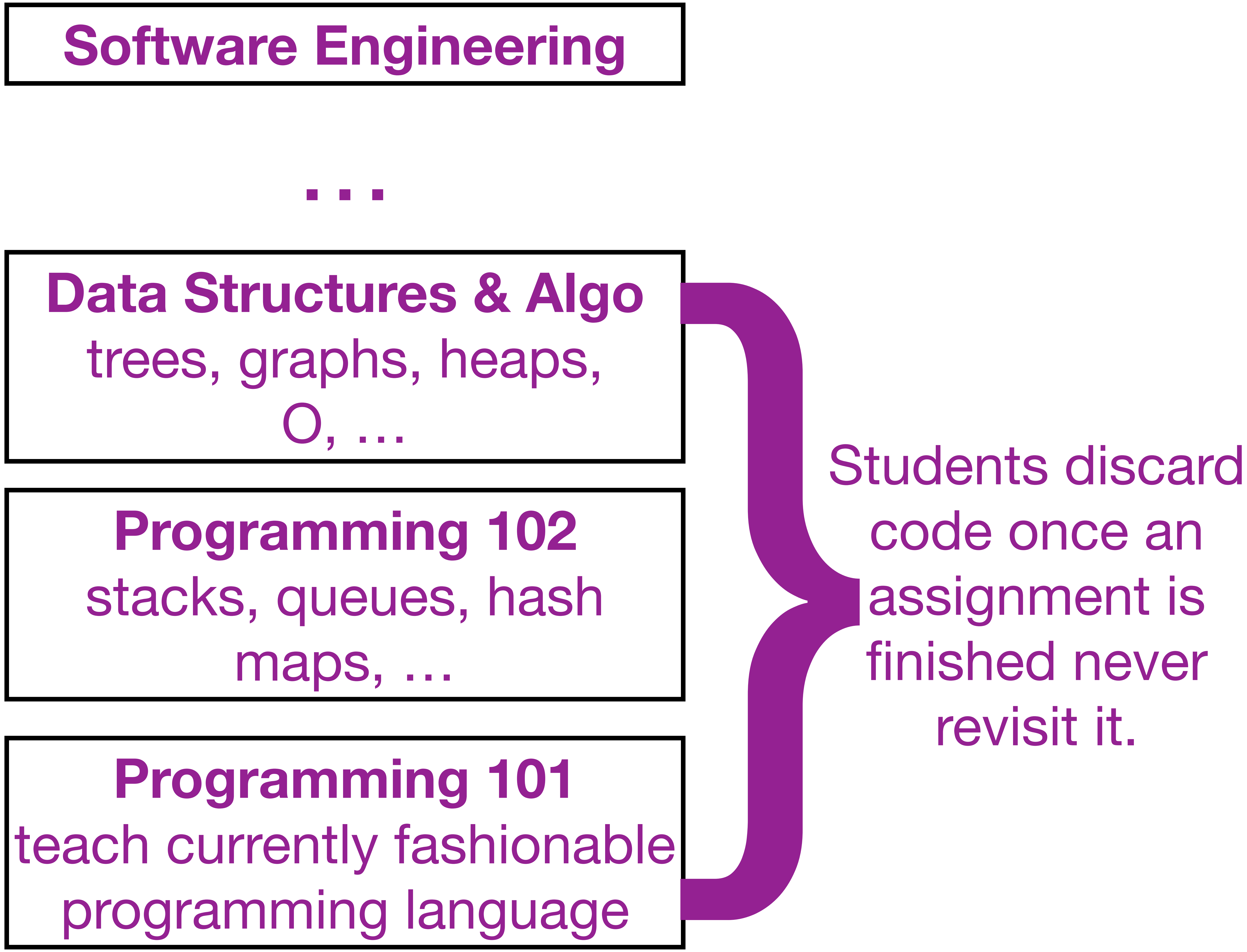
Data Structures & Algo
trees, graphs, heaps, O, …

Programming 102
stacks, queues, hash maps, …

Programming 101
teach currently fashionable programming language

Students discard code once an assignment is finished never revisit it.

Curriculum: Traditional vs Sw Dev



Software Engineering

…

Data Structures & Algo
trees, graphs, heaps, O, …

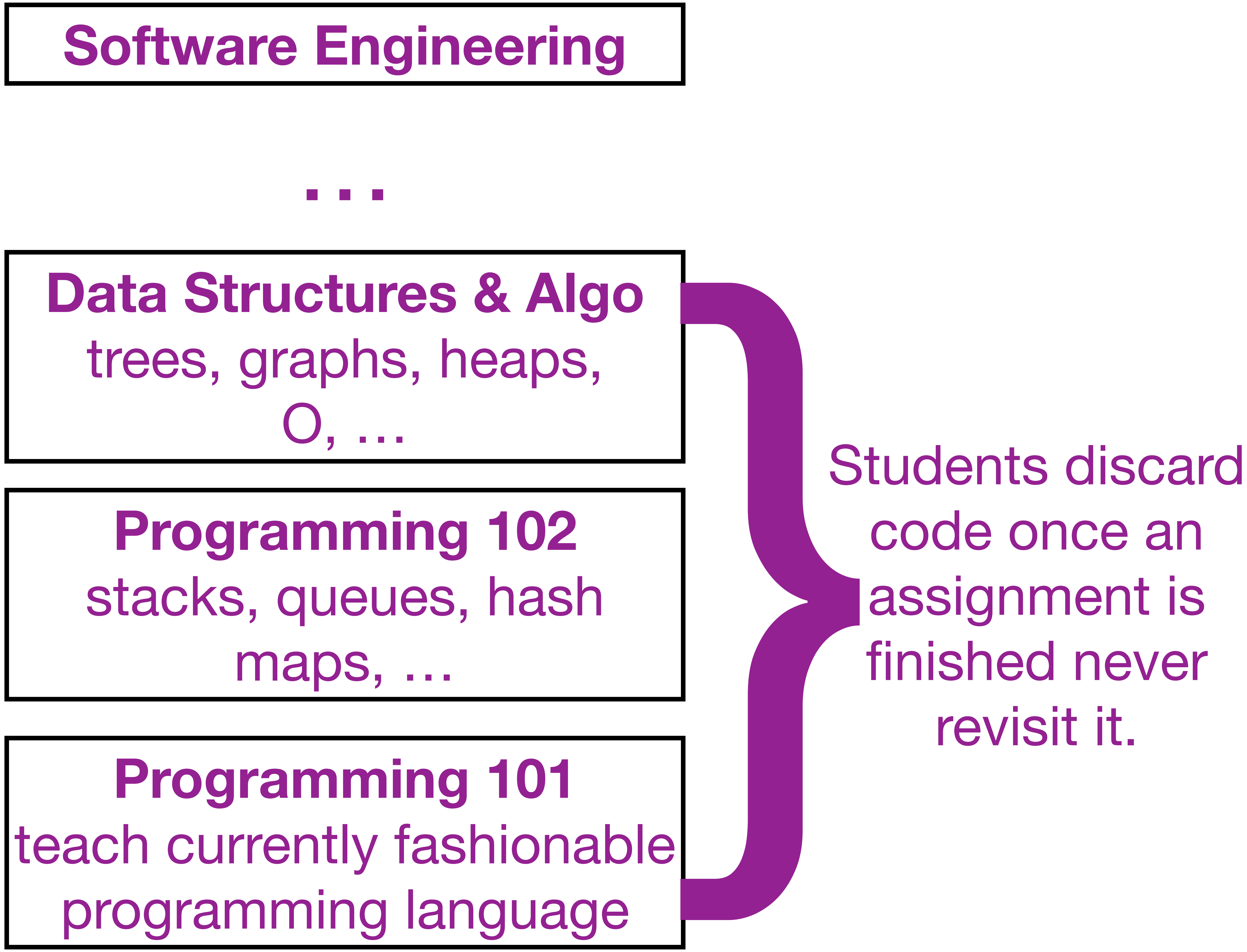
Programming 102
stacks, queues, hash maps, …

Programming 101
teach currently fashionable programming language

Students discard code once an assignment is finished never revisit it.

What changes over the years?

Curriculum: Traditional vs Sw Dev



Software Engineering

. . .

Data Structures & Algo
trees, graphs, heaps,
O, …

Programming 102
stacks, queues, hash
maps, …

Programming 101
teach currently fashionable
programming language

Students discard code once an assignment is finished never revisit it.

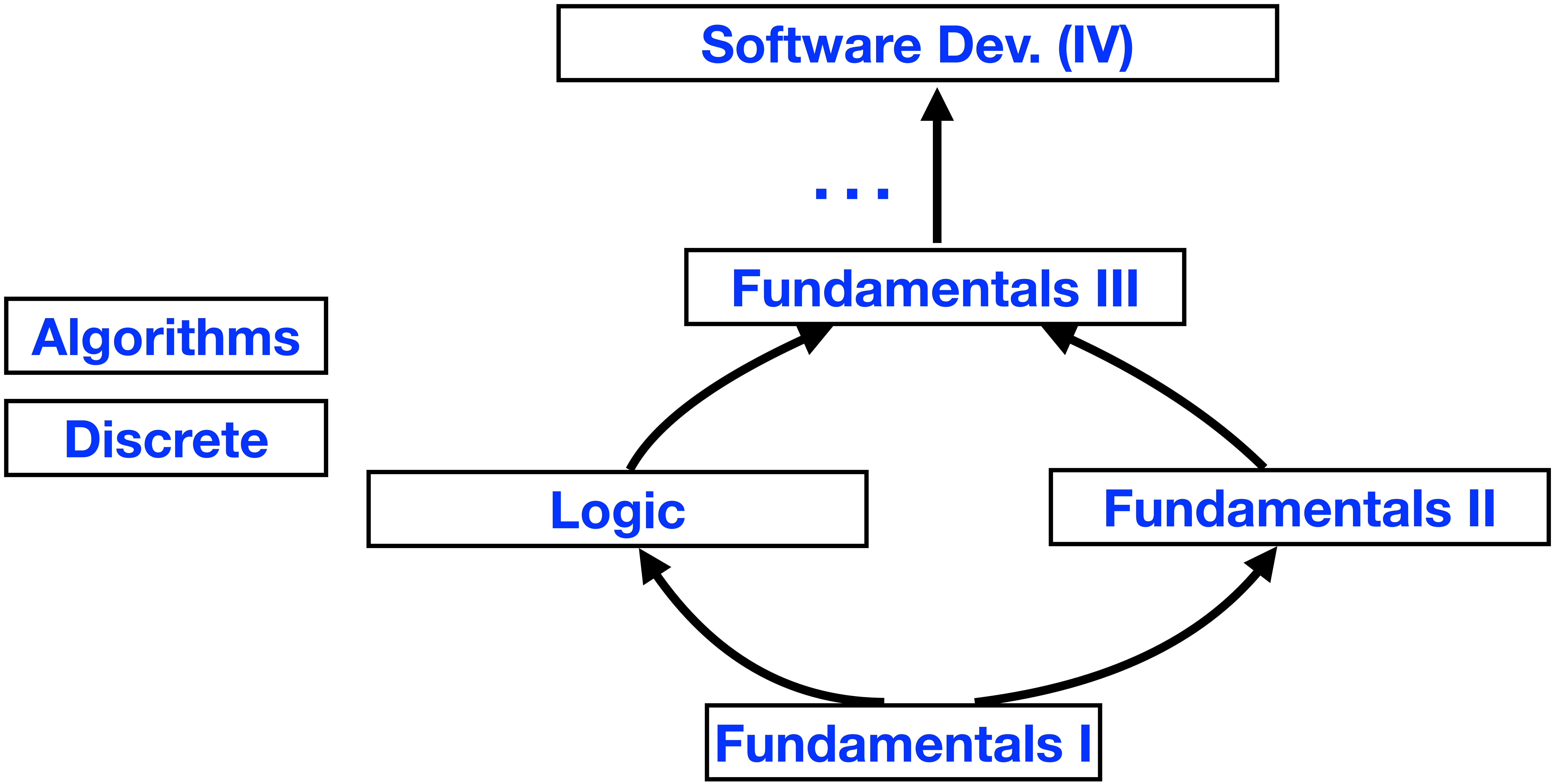
What changes over the years?

The programming language:

- — Algol 60, Simula 67
- — Pascal
- — Modula
- — Scheme
- — C/C++
- — Java
- — Haskell
- — Python

40 years, 10 languages

Curriculum: Traditional vs Sw Dev



Software Dev. (IV)

. . .

Fundamentals III

Algorithms

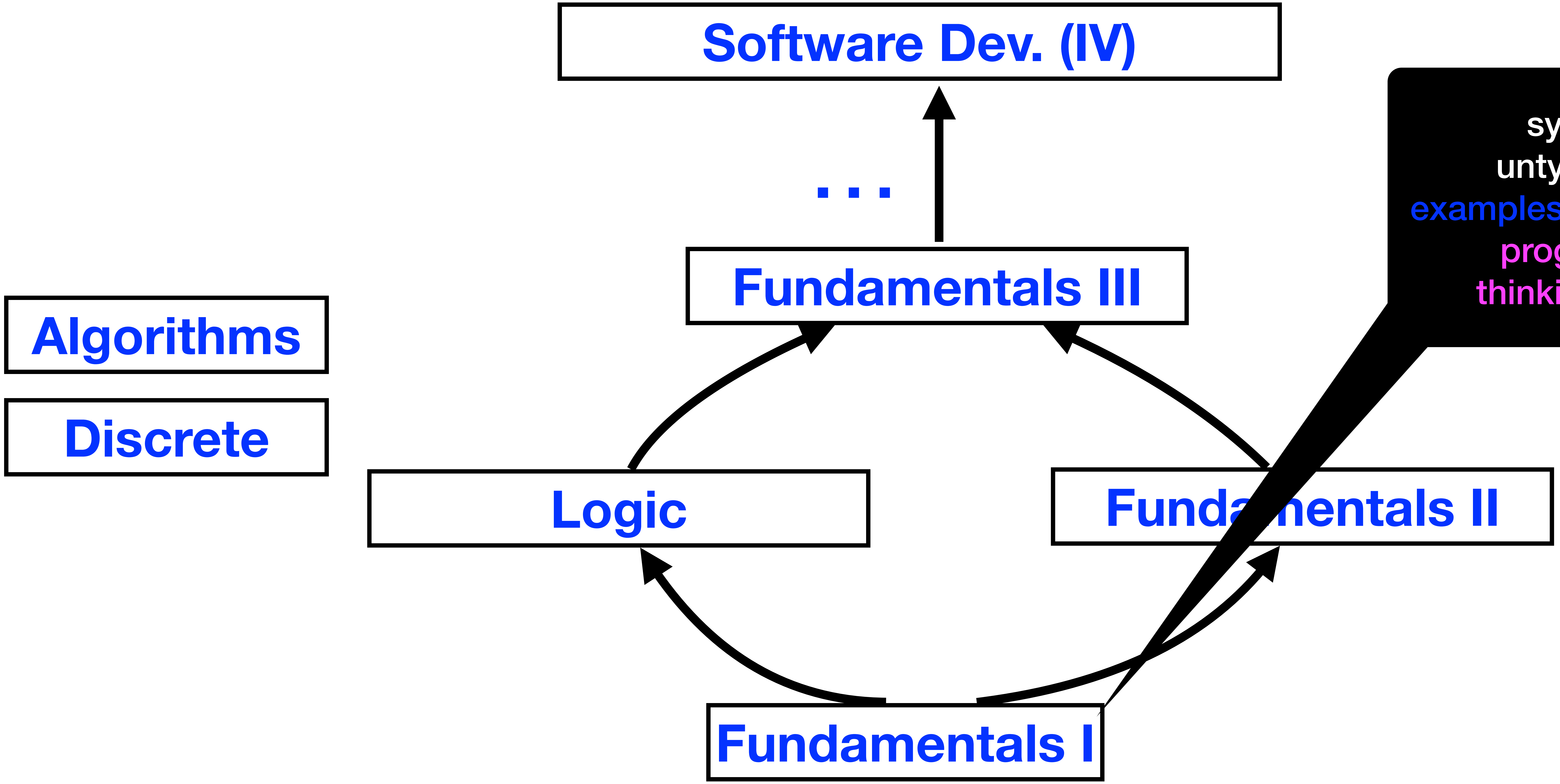
Discrete

Logic

Fundamentals II

Fundamentals I

Curriculum: Traditional vs Sw Dev



systematic development
untyped teaching languages
examples: interactive, distributed prog.
programming with a partner
thinking and speaking out loud

Curriculum: Traditional vs Sw Dev

Software Dev. (IV)

. . .

Fundamentals III

Algorithms

Discrete

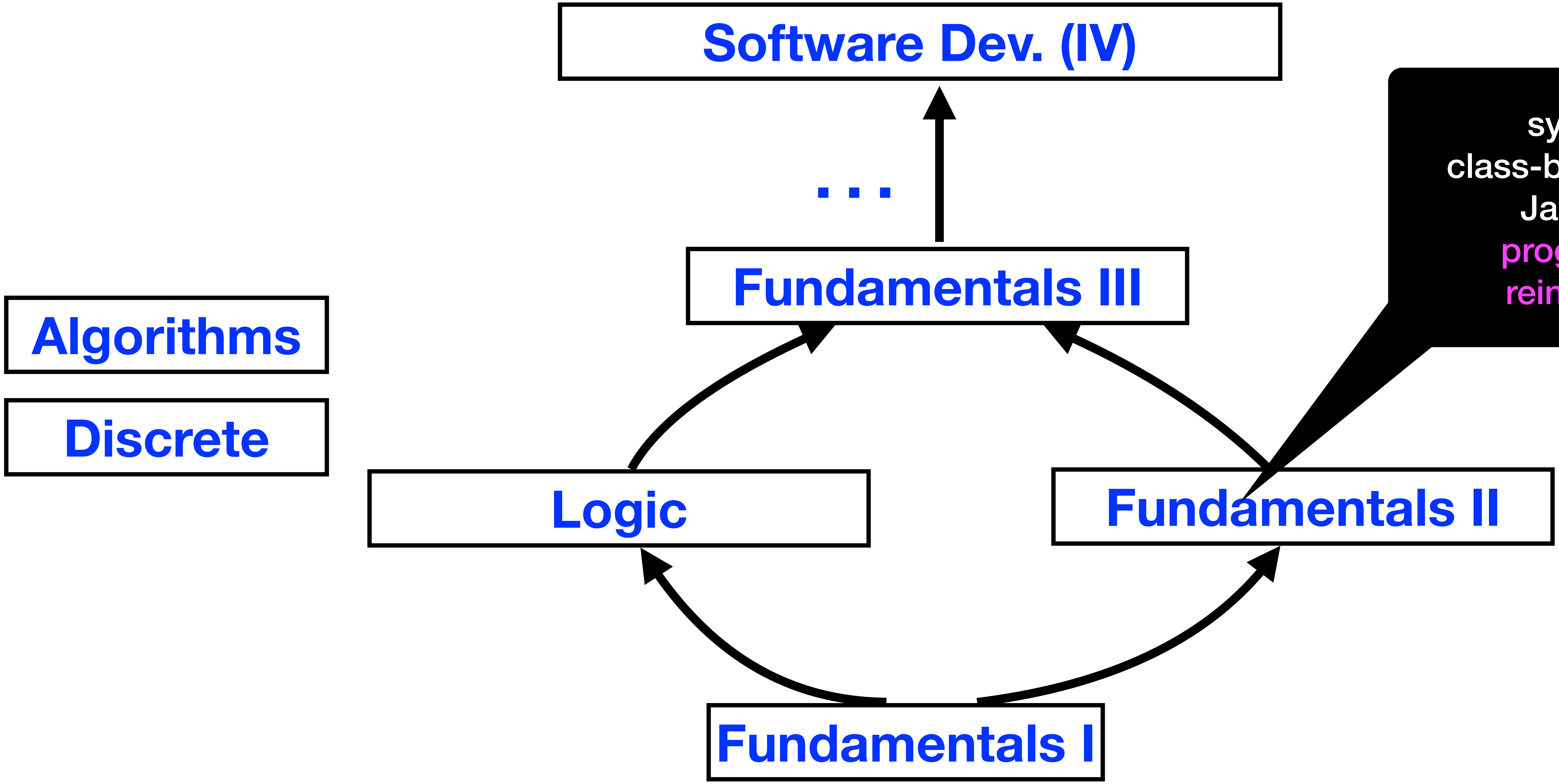
Logic

Fundamentals II

Fundamentals I

programs have properties
properties as Boolean expressions
property testing
Boolean expressions as theorems
proof assistant

Curriculum: Traditional vs Sw Dev



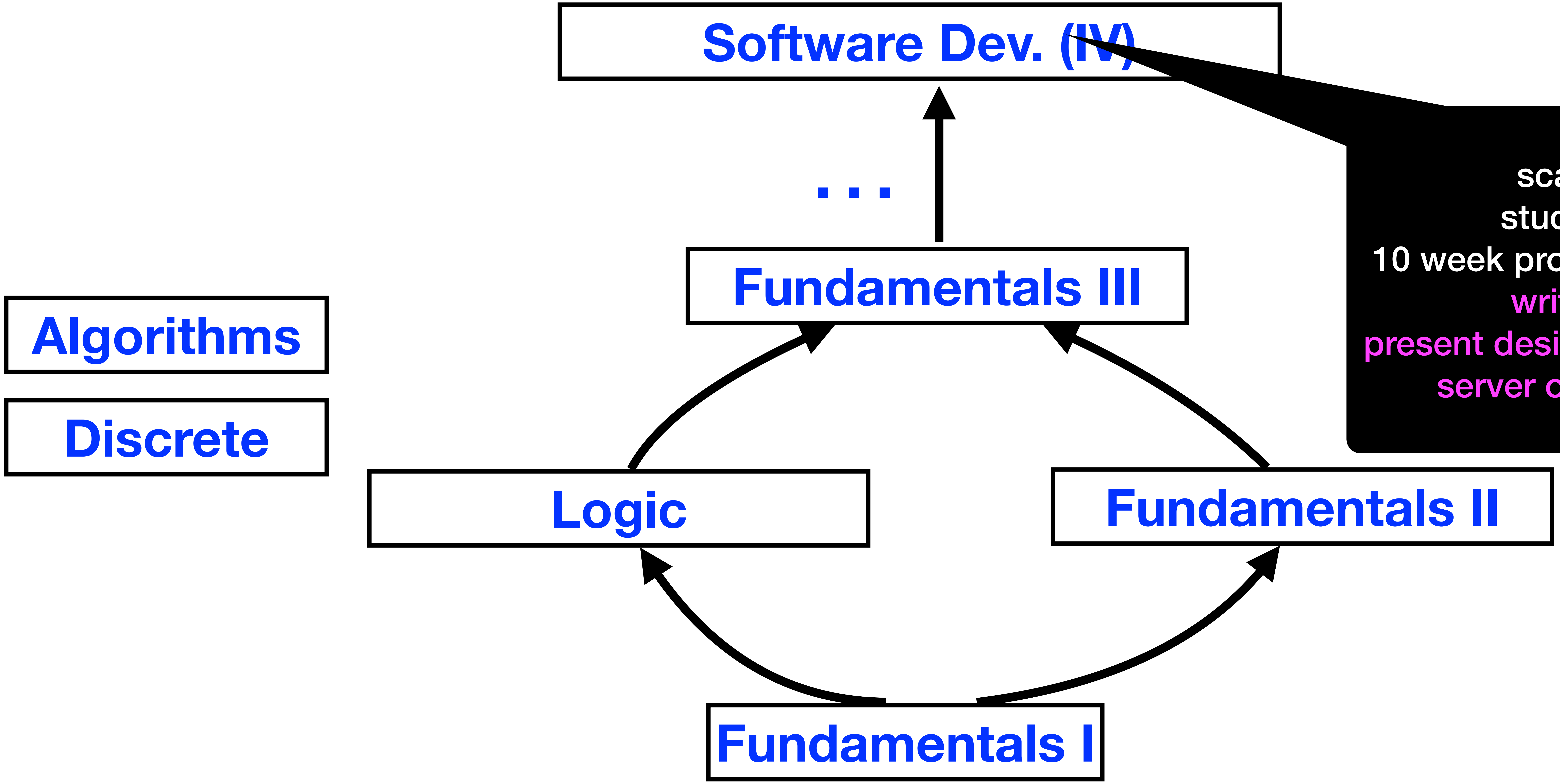
systematic development
class-based, object-oriented prog.
Java (TS, C# would work)
programming with a partner
reinforce pair programming

Curriculum: Traditional vs Sw Dev

Software Dev. (IV)
. . .
Fundamentals III
Algorithms
Discrete
Logic
Fundamentals II
Fundamentals I

patterns & sys. development
Java
2 6-week GUI programs
design interfaces as “wish list item”
swap wishlist, implement, swap impl.

Curriculum: Traditional vs Sw Dev



Software Dev. (IV)

. . .

Fundamentals III

Algorithms

Discrete

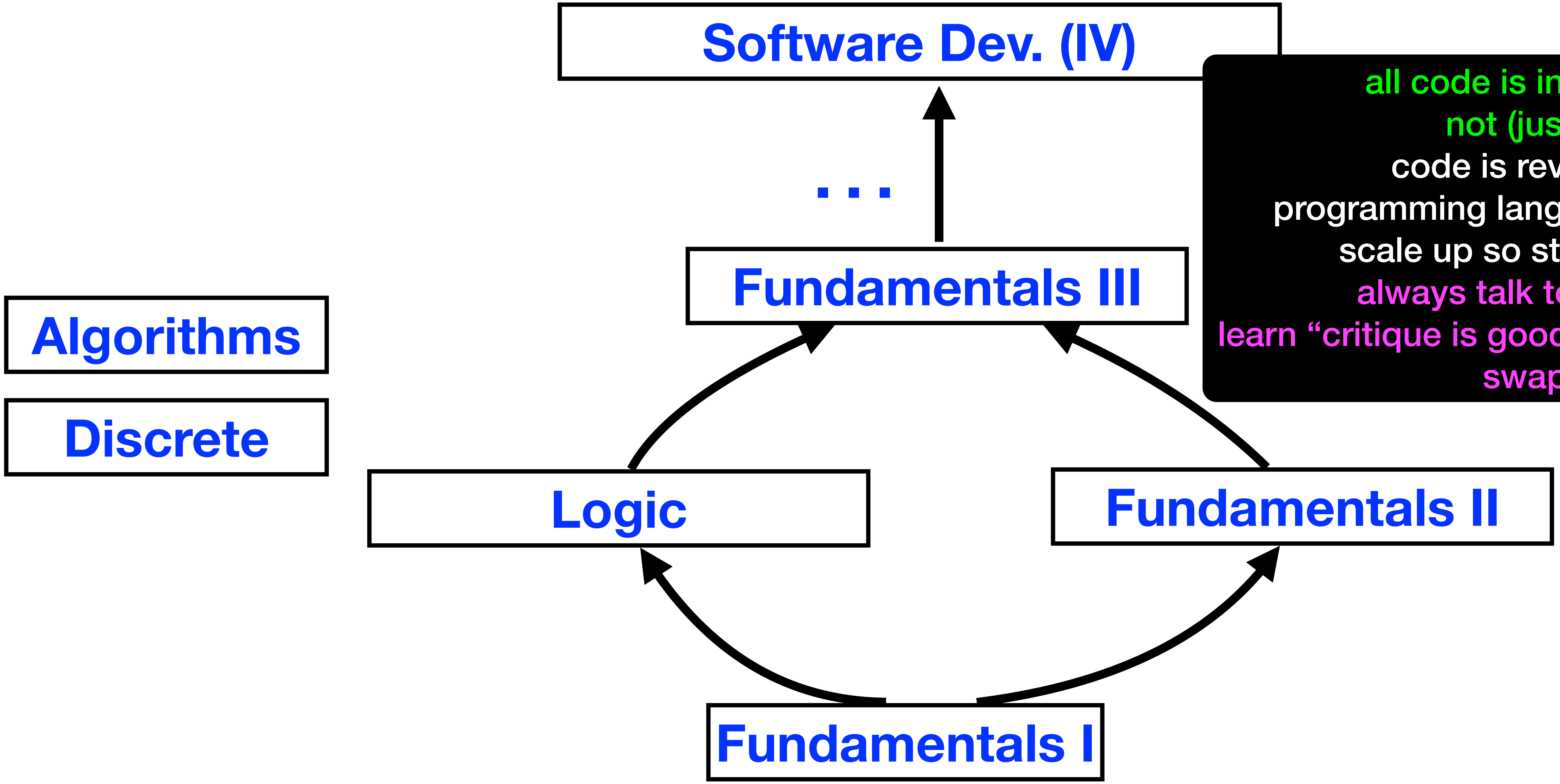
Logic

Fundamentals II

Fundamentals I

scaling it up (10-20Kloc)
student-chosen language
10 week project (config., TCP, JSON, GUI)
write milestones and fail
present design to panel & accept criticism;
server on panel and critique code

Curriculum: Traditional vs Sw Dev



Software Dev. (IV)

…

Fundamentals III

Algorithms

Discrete

Logic

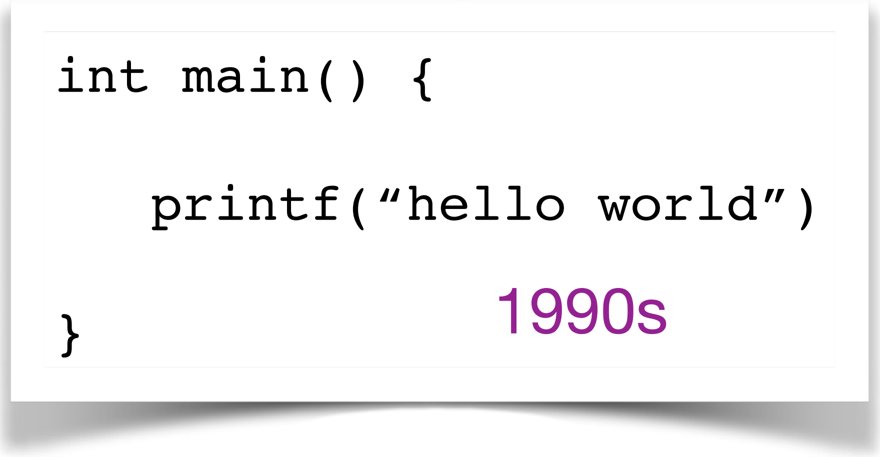
Fundamentals II

Fundamentals I

all code is inspected for quality,
not (just) functionality
code is revisited weeks later
programming language/development stage
scale up so students see problems
always talk to others about code
learn “critique is good” — “egoless programming”
swap code basis

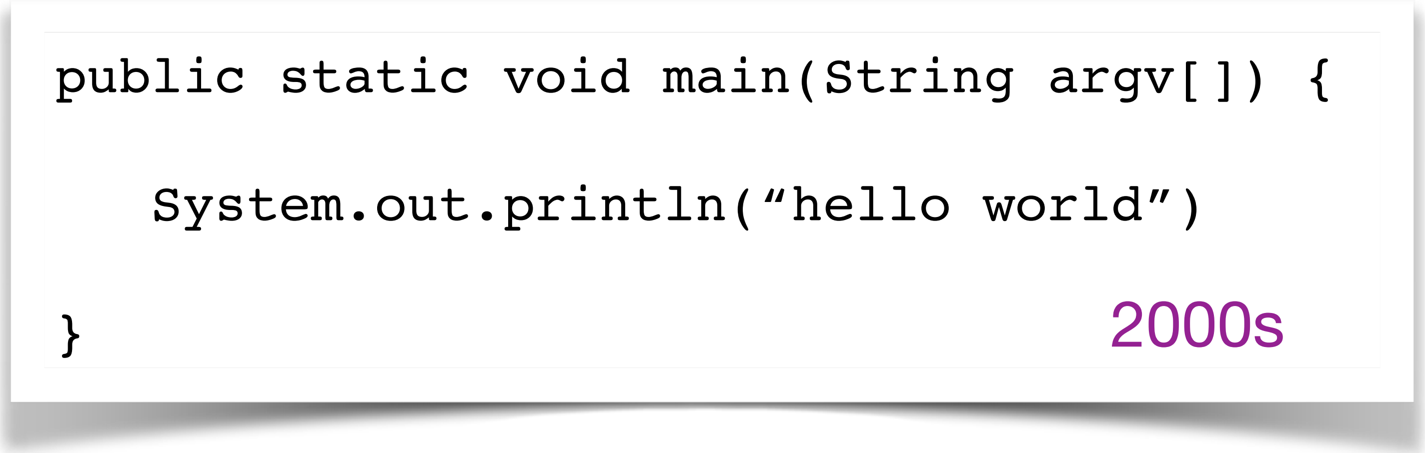
Programming 101

Programming 101, the Old Way



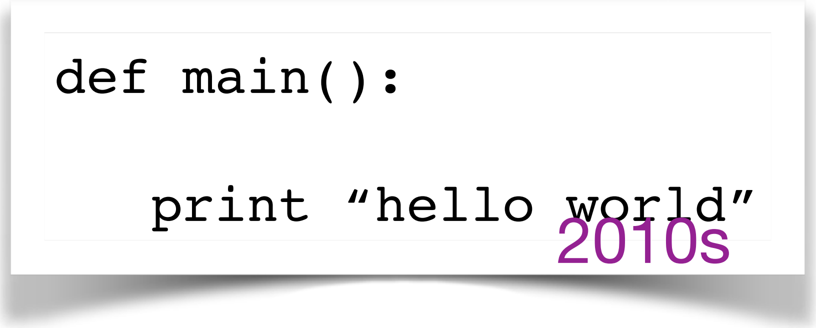
```
int main() {

   printf(“hello world”)

}
```

1990s



```
public static void main(String argv[]) {

   System.out.println(“hello world”)

}
```

2000s

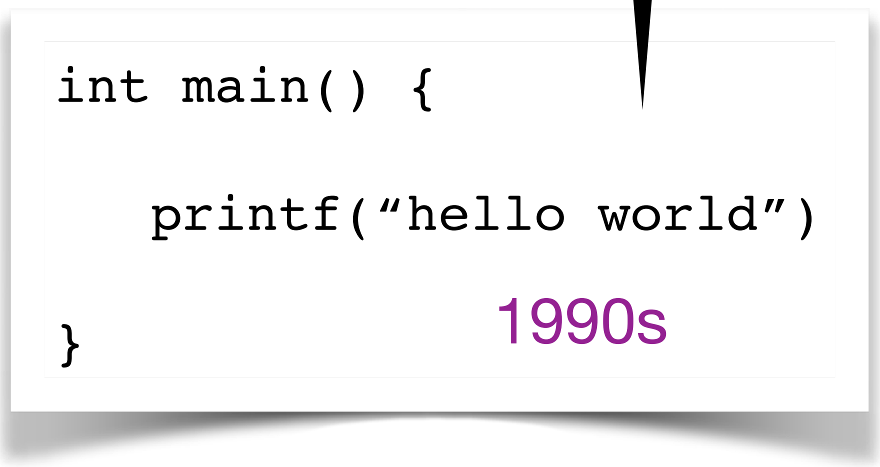


```
def main():

   print “hello world”
```

2010s

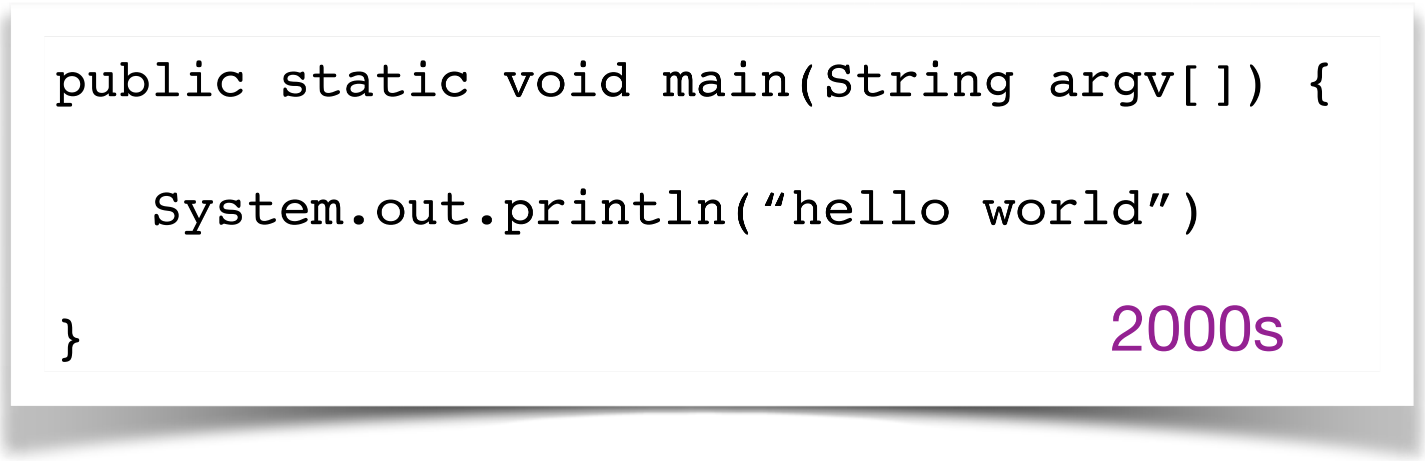
Programming 101, the Old Way

Choose a fashionable language.



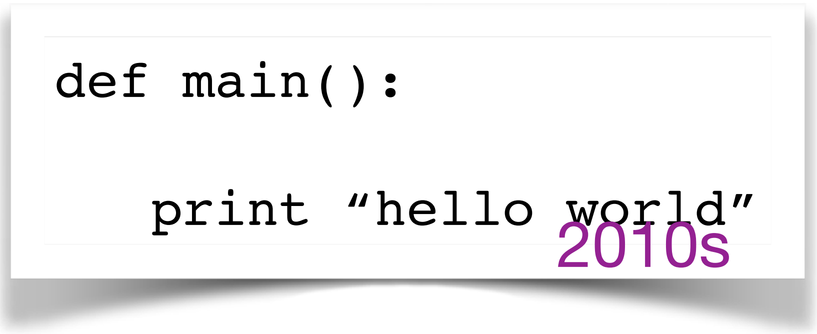
```
int main() {

   printf(“hello world”)

}
```

1990s



```
public static void main(String argv[]) {

   System.out.println(“hello world”)

}
```

2000s



```
def main():

   print “hello world”
```

2010s

Programming 101, the Old Way

Choose a fashionable language.

```
int main() {
   printf(“hello world”)
}
```

1990s

Present one syntactic mechanism after another.

```
public static void main(String argv[]) {
   System.out.println(“hello world”)
}
```

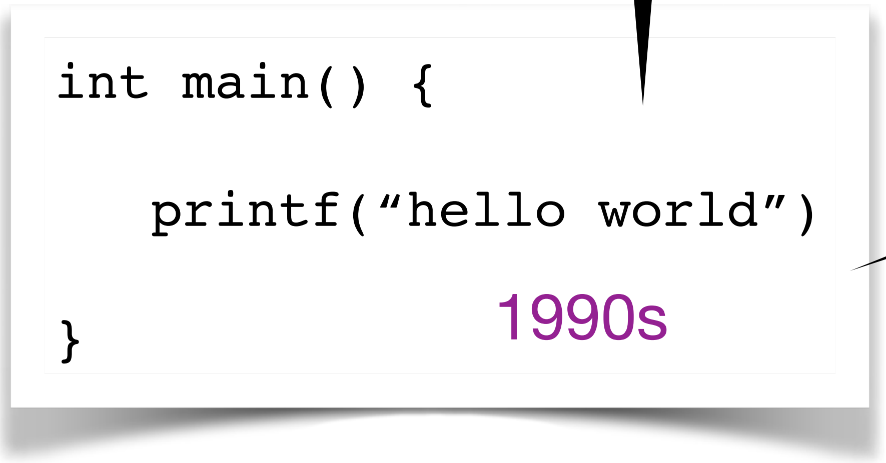
2000s

```
def main():
   print “hello world”
```

2010s

Programming 101, the Old Way

Choose a fashionable language.

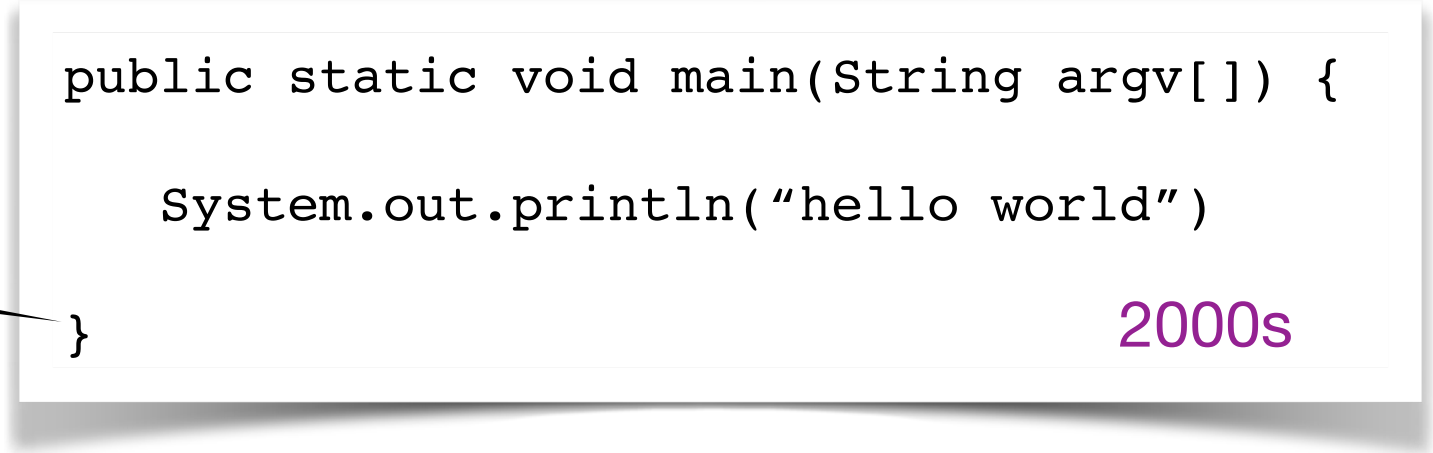


```
int main() {

   printf(“hello world”)

}
```

1990s

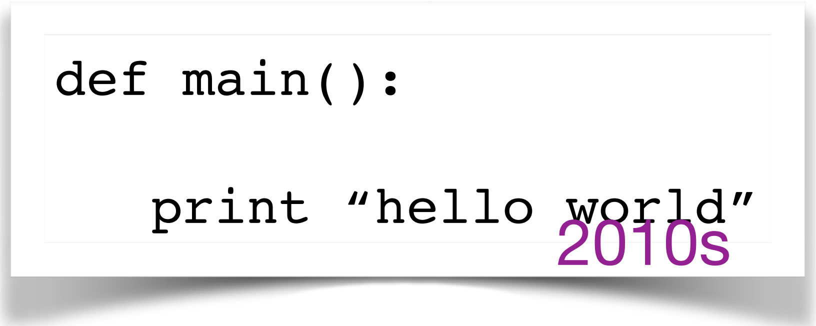
Present one syntactic mechanism after another.

Copy my code and adapt for this slightly different problem.



```
public static void main(String argv[]) {

   System.out.println(“hello world”)

}
```

2000s

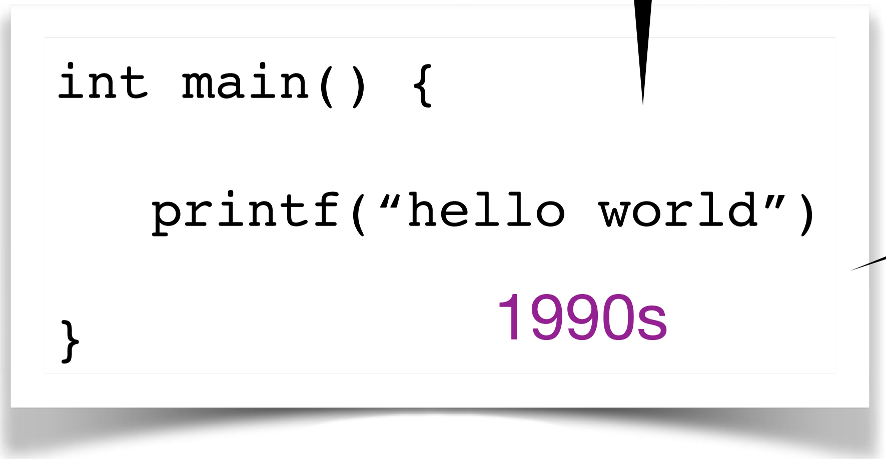


```
def main():

   print “hello world”
```

2010s

Programming 101, the Old Way

Choose a fashionable language.

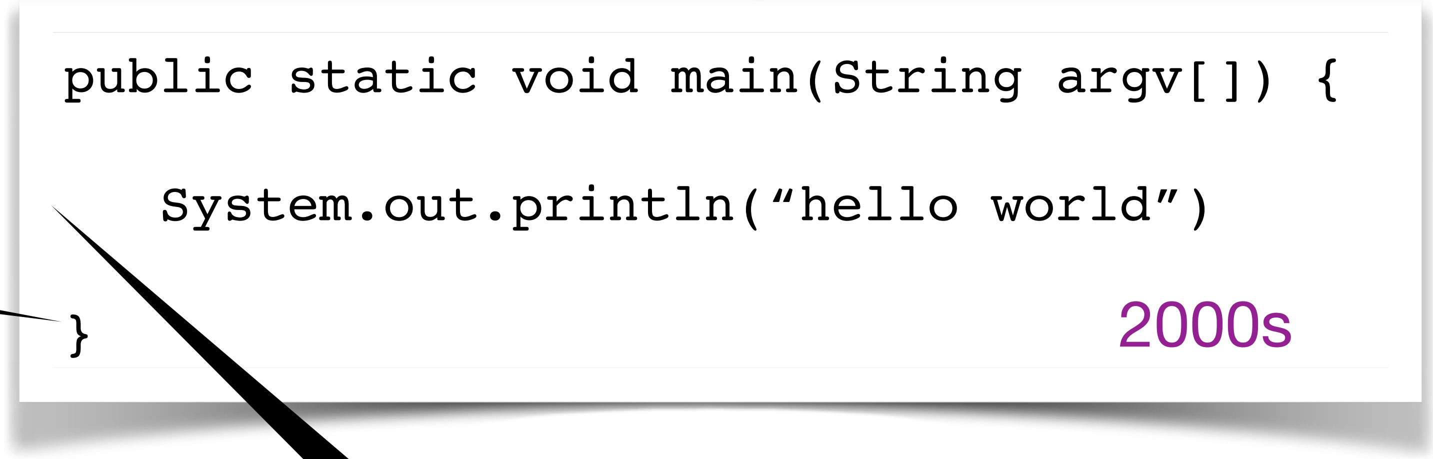


```
int main() {
   printf(“hello world”)
}
```

1990s

Present one syntactic mechanism after another.

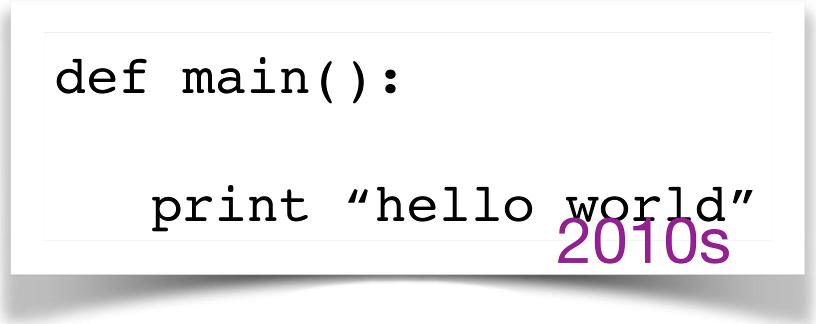
Copy my code and adapt for this slightly different problem.



```
public static void main(String argv[]) {
   System.out.println(“hello world”)
}
```

2000s

If it doesn’t work, add print statements.

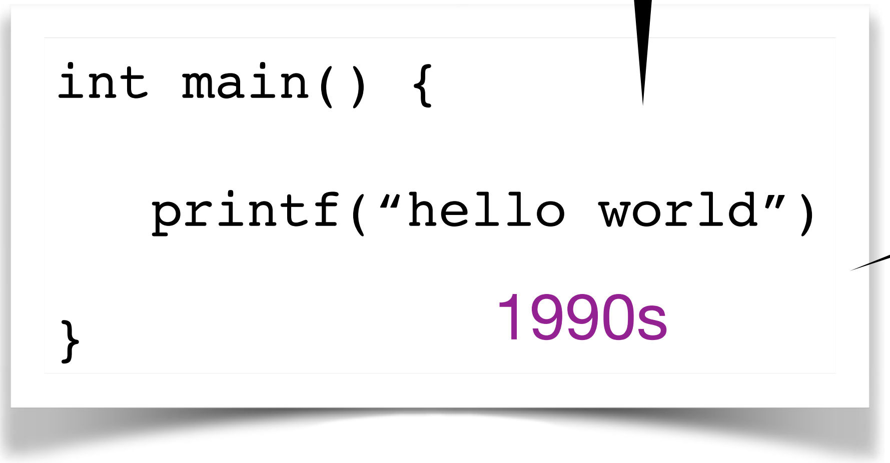


```
def main():
   print “hello world”
```

2010s

Programming 101, the Old Way

Choose a fashionable language.

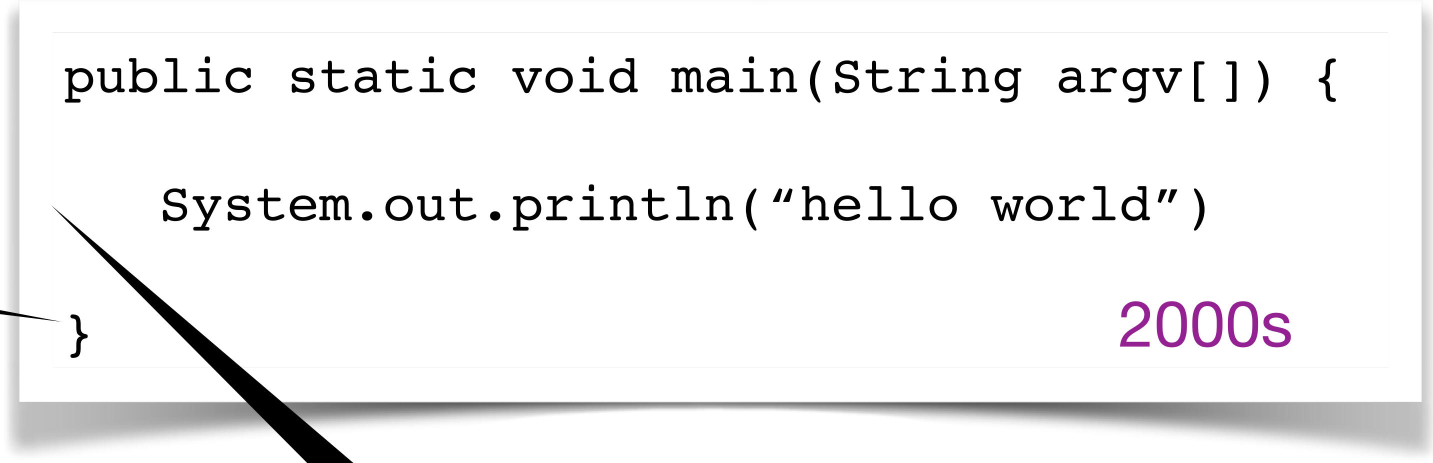


```
int main() {
    printf(“hello world”)
}
```

1990s

Present one syntactic mechanism after another.

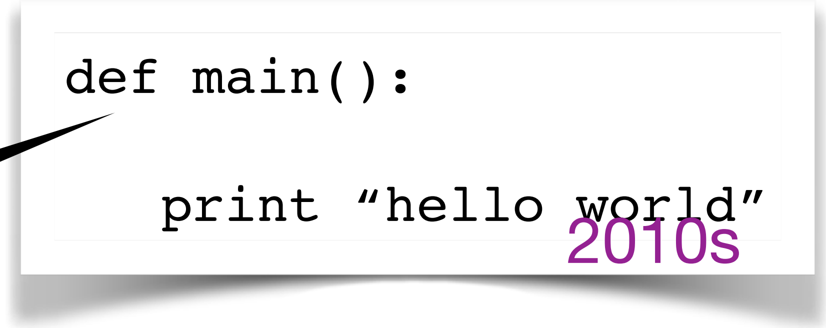
Copy my code and adapt for this slightly different problem.



```
public static void main(String argv[]) {
    System.out.println(“hello world”)
}
```

2000s

If it doesn’t work, add print statements.



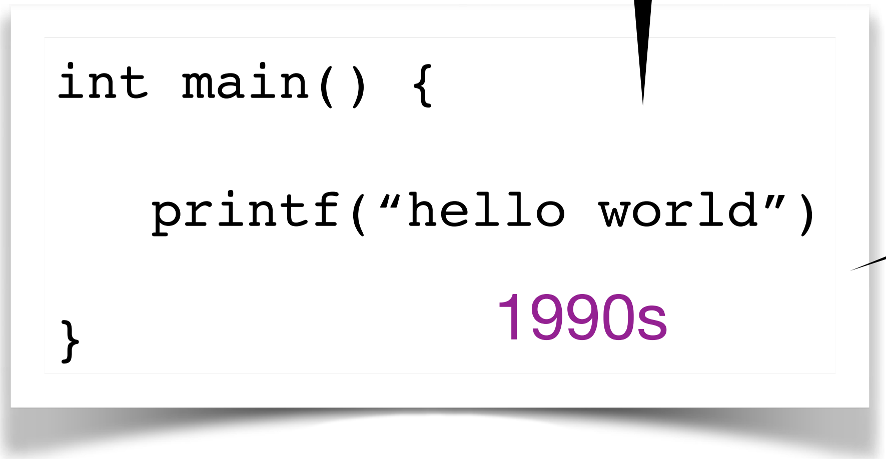
```
def main():
    print “hello world”
```

2010s

Truly advanced? Use a debugger.

Programming 101, the Old Way

Choose a fashionable language.

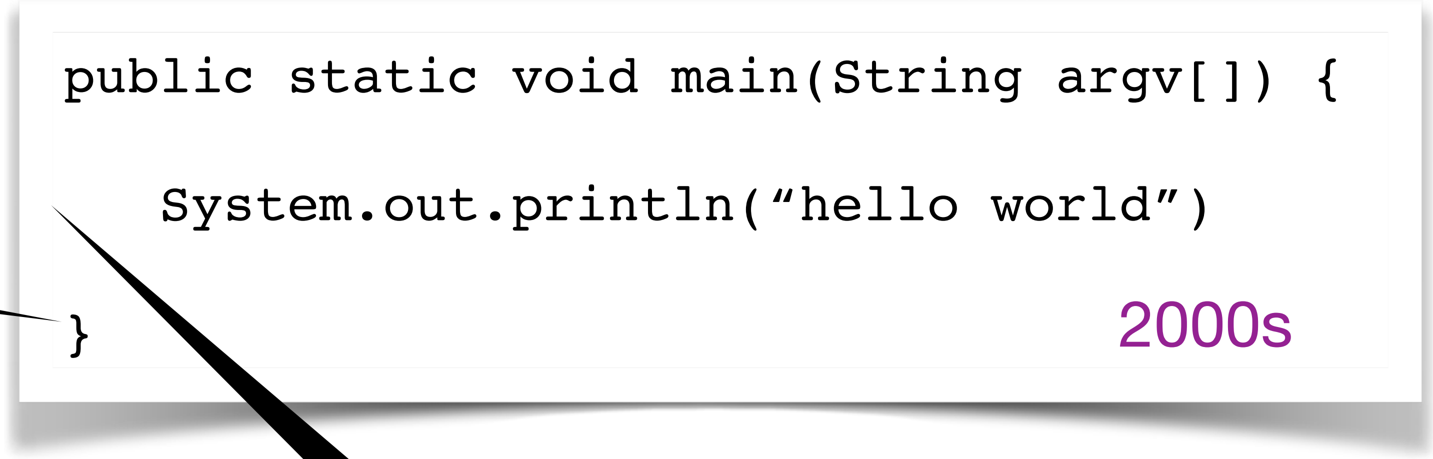


```
int main() {

   printf(“hello world”)

}
```

1990s

Present one syntactic mechanism after another.

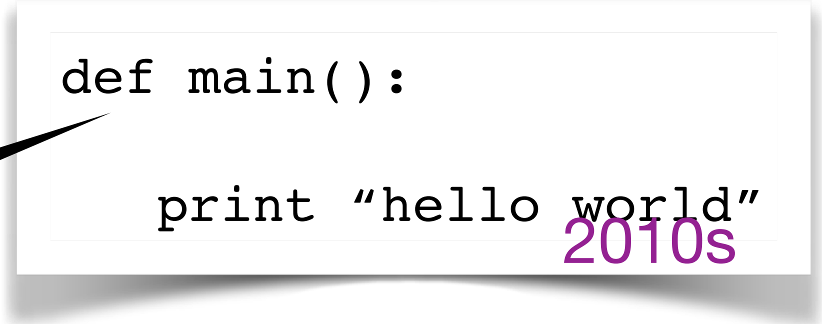
Copy my code and adapt for this slightly different problem.



```
public static void main(String argv[]) {

   System.out.println(“hello world”)

}
```

2000s

If it doesn’t work, add print statements.



```
def main():

   print “hello world”
```

2010s

Truly advanced? Use a debugger.

And after all that,
the code gets autograded
and no teaching assistant
looks at it.
Time to throw it away.

Fundamentals I

Programming, the Technical Skill

Social Interaction about Programming

Fundamentals I

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- drive course development by increasing the complexity of data

Social Interaction about Programming

Fundamentals I

PL: teaching language

Programming, the Technical Skill

Social Interaction about Programming

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- drive course development by increasing the complexity of data

Fundamentals I

PL: teaching language

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- drive course development by increasing the complexity of data

Social Interaction about Programming

- programming is thinking, thinking is best done with others
- practice proper pair programming
- confront students with their code from a couple of weeks ago
- ask students to react to code criticisms by teaching assistants

Fundamentals I, the Technical Skills

Programming, the Technical Skill

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

choose a data representation to represent “the problem” & its result
data description aka data definition
data examples

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

choose a data representation to
represent “the problem” & its result
data description aka data definition
data examples

state what goes in and what comes out
state purpose in your own words
“type signature” (in an untyped PL)
a “*what* does it compute” comment

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

choose a data representation to
represent “the problem” & its result
data description aka data definition
data examples

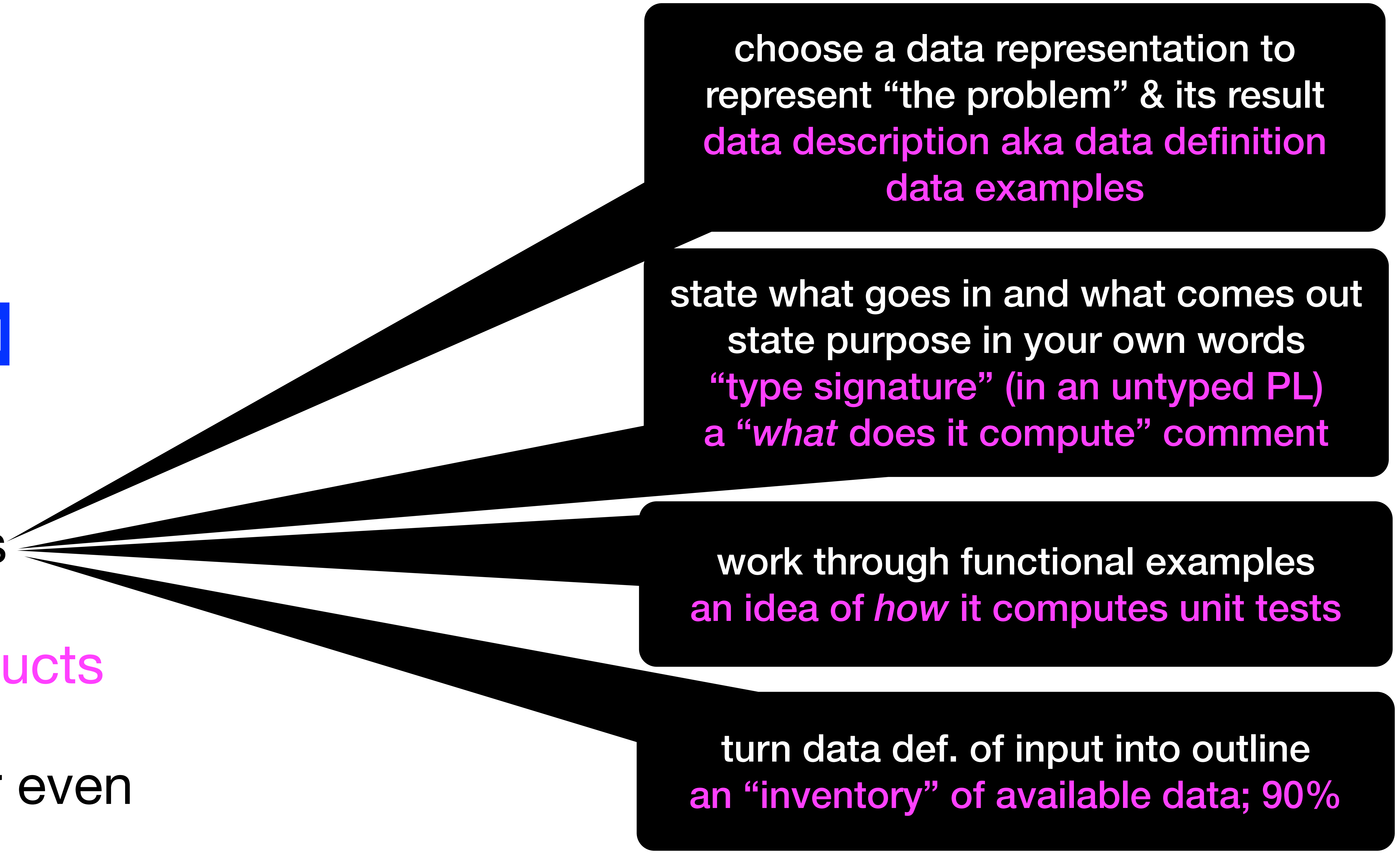
state what goes in and what comes out
state purpose in your own words
“type signature” (in an untyped PL)
a “*what* does it compute” comment

work through functional examples
an idea of *how* it computes unit tests

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



choose a data representation to
represent “the problem” & its result
data description aka data definition
data examples

state what goes in and what comes out
state purpose in your own words
“type signature” (in an untyped PL)
a “*what* does it compute” comment

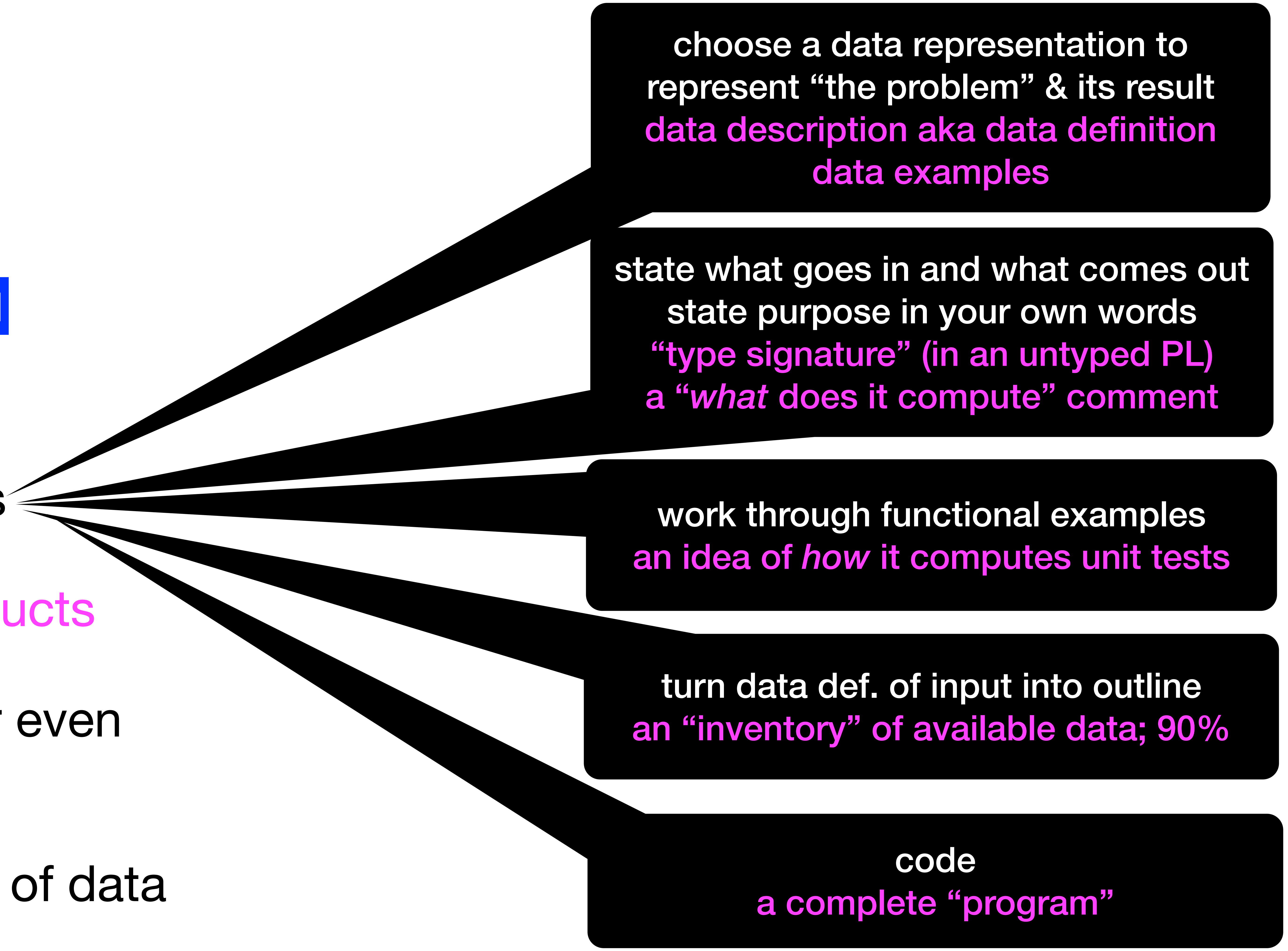
work through functional examples
an idea of *how* it computes unit tests

turn data def. of input into outline
an “inventory” of available data; 90%

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



choose a data representation to represent "the problem" & its result
data description aka data definition
data examples

state what goes in and what comes out
state purpose in your own words
"type signature" (in an untyped PL)
a "*what* does it compute" comment

work through functional examples
an idea of *how* it computes unit tests

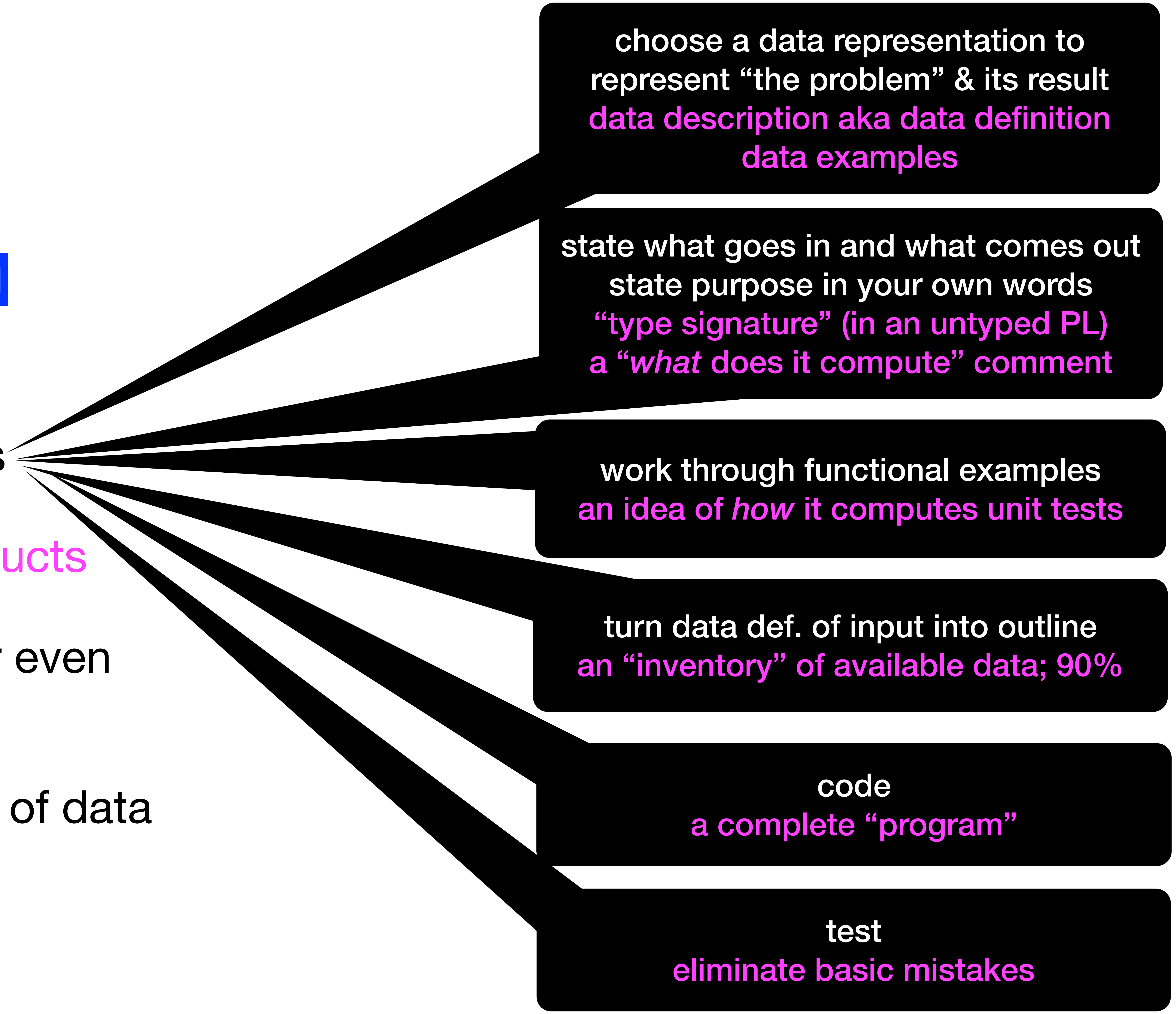
turn data def. of input into outline
an "inventory" of available data; 90%

code
a complete "program"

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



choose a data representation to
represent “the problem” & its result
data description aka data definition
data examples

state what goes in and what comes out
state purpose in your own words
“type signature” (in an untyped PL)
a “*what* does it compute” comment

work through functional examples
an idea of *how* it computes unit tests

turn data def. of input into outline
an “inventory” of available data; 90%

code
a complete “program”

test
eliminate basic mistakes

Fundamentals I, the Technical Skills

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)

/* Shape is one of
  - Square(posnum)
  - Circle(posnum)
  - Triangle(posnum, posnum)
correspond to the respective
geometric shapes. */
```

Data Examples

```
let sq = Square(4)
let cr = Circle(3)
let tr = Triangle(2,1)
```

Fundamentals I, the Technical Skills

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Programming, the Technical Skill

An instructor or teaching assistant can inspect these intermediate results and intervene *before* the student goes off track.

- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)

/* Shape is one of
  - Square(posnum)
  - Circle(posnum)
  - Triangle(posnum, posnum)
correspond to the respective
geometric shapes. */
```

Data Examples

```
let sq = Square(4)
let cr = Circle(3)
let tr = Triangle(2,1)
```

Fundamentals I, the Technical Skills

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Purpose, Signature, Stub

```
// determine the area of `s`
// Shape -> PosNumber
def area(s):
   0
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

Purpose: do student/devs understand the problem?
Signature: don't you wish all untyped code had those?

- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Purpose, Signature, Stub

```
// determine the area of `s`
// Shape -> PosNumber
def area(s):
   0
```

Fundamentals I, the Technical Skills

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Examples

```
let sq = Square(4)
let cr = Circle(3)
let tr = Triangle(2,1)
```

Worked Functional Examples

```
// area(tr)
//   = 1/2 * tr.base * tr.height
//   = 1/2 * 2 * 1
//   = 1

checkExpect(area(tr),1)
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process

It is a bit more than test-driven development. The teaching assistant can check whether students can work through or whether they are guessing.

- the simplest problems
- increase the complexity of data

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Data Examples

```
let sq = Square(4)
let cr = Circle(3)
let tr = Triangle(2,1)
```

Worked Functional Examples

```
// area(tr)
//   = 1/2 * tr.base * tr.height
//   = 1/2 * 2 * 1
//   = 1

checkExpect(area(tr),1)
```

Fundamentals I, the Technical Skills

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
   - Square(posnum)
   - Circle(posnum)
   - Triangle(posnum, posnum. */
```

Function Outline/Inventory of data

```
def area(s):
   condition:
    s is Square: .. s.side ..
    s is Circle: .. s.radius ..
    s is Triangle:
          .. s.base .. s.height ..
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process

A program must compute its outputs from the given data and nothing else. Scales to *all* forms of data.

the simplest problems

- increase the complexity of data

Problem A shape is either a square, a circle, or a triangle. Design a program that computes the area of one of these shapes.

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
   - Square(posnum)
   - Circle(posnum)
   - Triangle(posnum, posnum. */
```

Function Outline/Inventory of data

```
def area(s):
   condition:
     s is Square: .. s.side ..
     s is Circle: .. s.radius ..
     s is Triangle:
           .. s.base .. s.height ..
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

The Code

```
// determine the area of `s`
// Shape -> PosNumber
def area(s):
  condition:
   s is Square: sq(s.side)
   s is Circle: pi * sq(s.radius)
   s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- practice good habits for even the simplest problems
- increase the complexity of data

Coding means filling in a last few gaps in the outline.

The Code

```
// determine the area of `s`
// Shape -> PosNumber
def area(s):
  condition:
    s is Square: sq(s.side)
    s is Circle: pi * sq(s.radius)
    s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
  - Square(posnum)
  - Circle(posnum)
  - Triangle(posnum, posnum. */

// determine the area of `s`
// Shape -> PosNumber
def area(s):
  condition:
   s is Square: sq(s.side)
   s is Circle: pi * sq(s.radius)
   s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Test failed.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
-
- practice good habits for even the simplest problems
- increase the complexity of data

Testing reveals typos and simple mistakes.

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
  - Square(posnum)
  - Circle(posnum)
  - Triangle(posnum, posnum. */

// determine the area of `s`
// Shape -> PosNumber
def area(s):
  condition:
   s is Square: sq(s.side)
   s is Circle: pi * sq(s.radius)
   s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Test failed.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
  - Square(posnum)
  - Circle(posnum)
  - Triangle(posnum, posnum. */

// determine the area of `s`
// Shape -> PosNumber
def area(s):
  condition:
   s is Square: sq(s.side)
   s is Circle: pi * sq(s.radius)
   s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Coverage incomplete.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process

And yes, incomplete coverage is taught as “it is a bug”.

- practice good habits for even the simplest problems

- increase the complexity of data

Data Representation

```
struct Square(side)
struct Circle(radius)
struct Triangle(base,height)
/* Shape is one of
   - Square(posnum)
   - Circle(posnum)
   - Triangle(posnum, posnum. */

// determine the area of `s`
// Shape -> PosNumber
def area(s):
   condition:
     s is Square: sq(s.side)
     s is Circle: pi * sq(s.radius)
     s is Triangle: s.base * s.height

checkExpect(area(tr),1)
```

Coverage incomplete.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

Atomic.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

Atomic.

Enumeration description.

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Atomic.

Enumeration description.

Structure description.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Atomic.

Enumeration description.

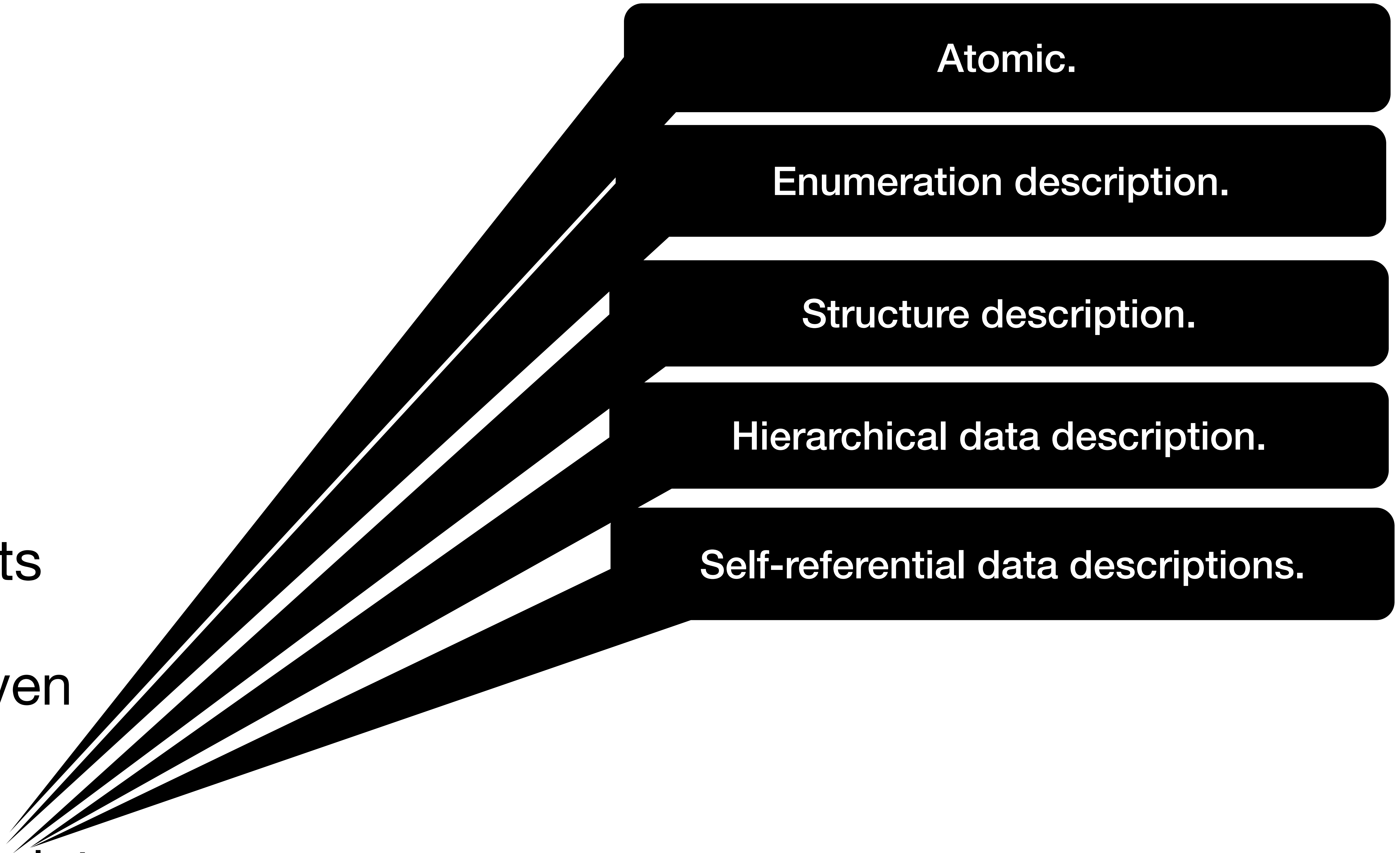
Structure description.

Hierarchical data description.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



Atomic.

Enumeration description.

Structure description.

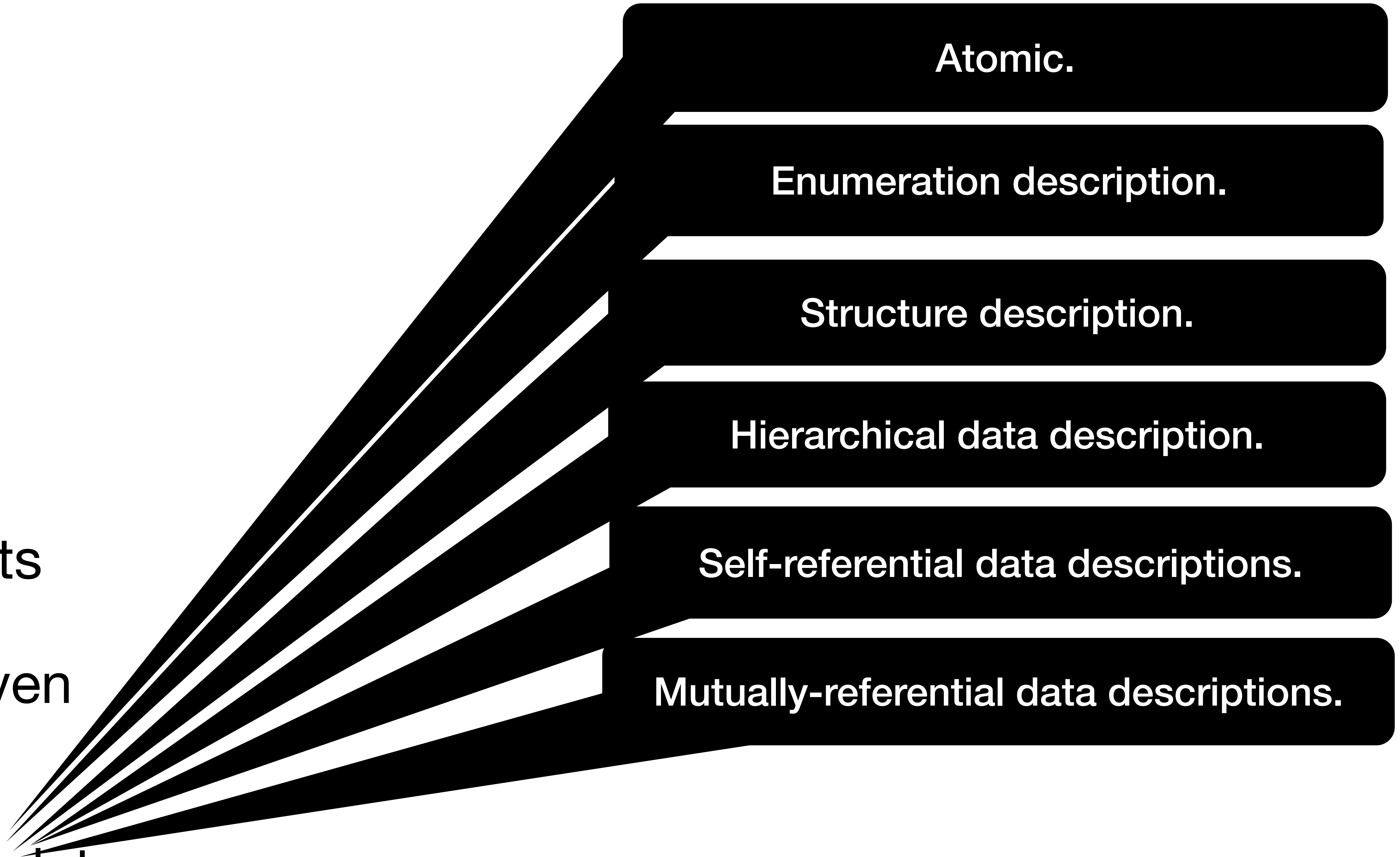
Hierarchical data description.

Self-referential data descriptions.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



Atomic.

Enumeration description.

Structure description.

Hierarchical data description.

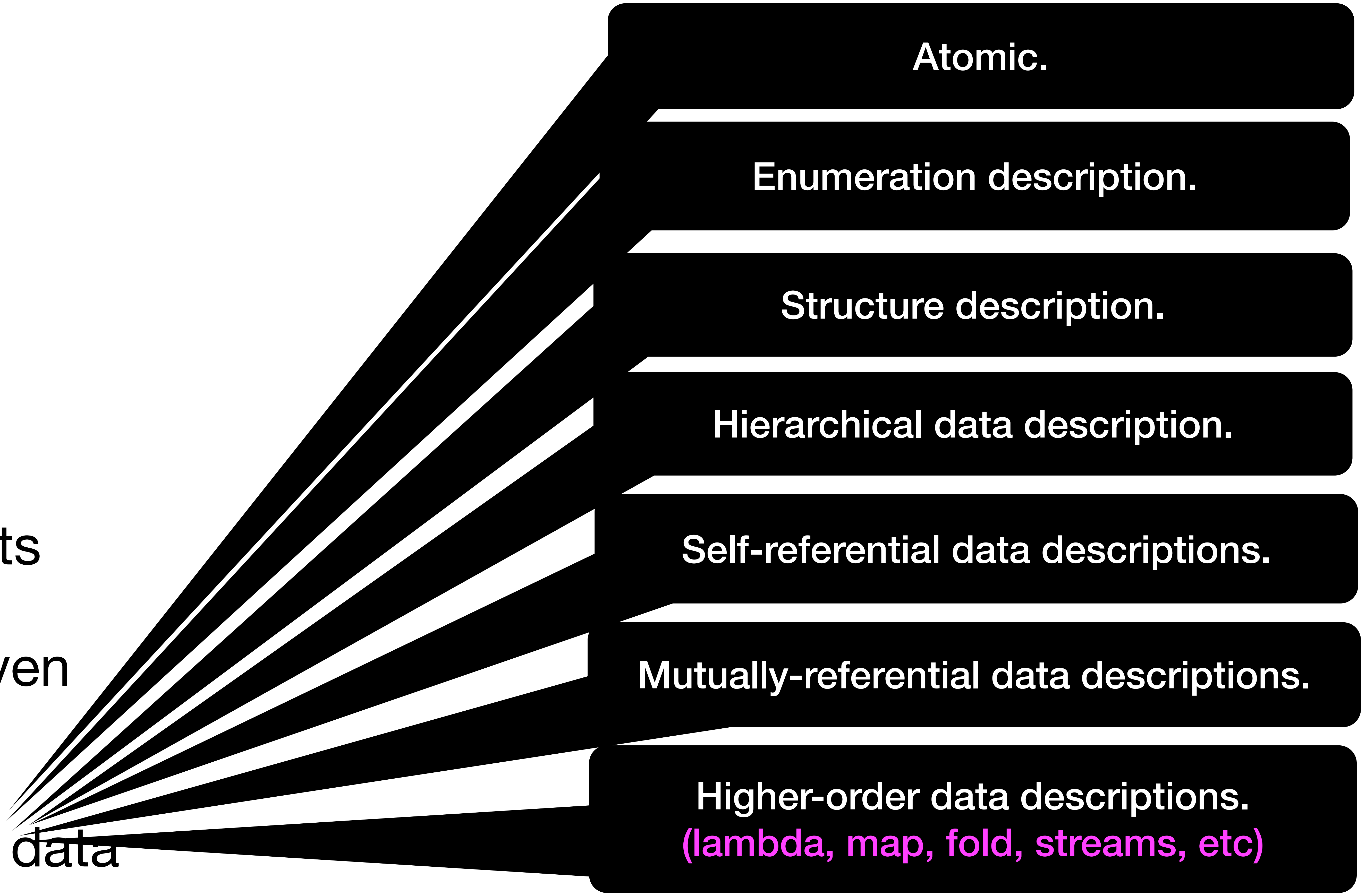
Self-referential data descriptions.

Mutually-referential data descriptions.

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



Atomic.

Enumeration description.

Structure description.

Hierarchical data description.

Self-referential data descriptions.

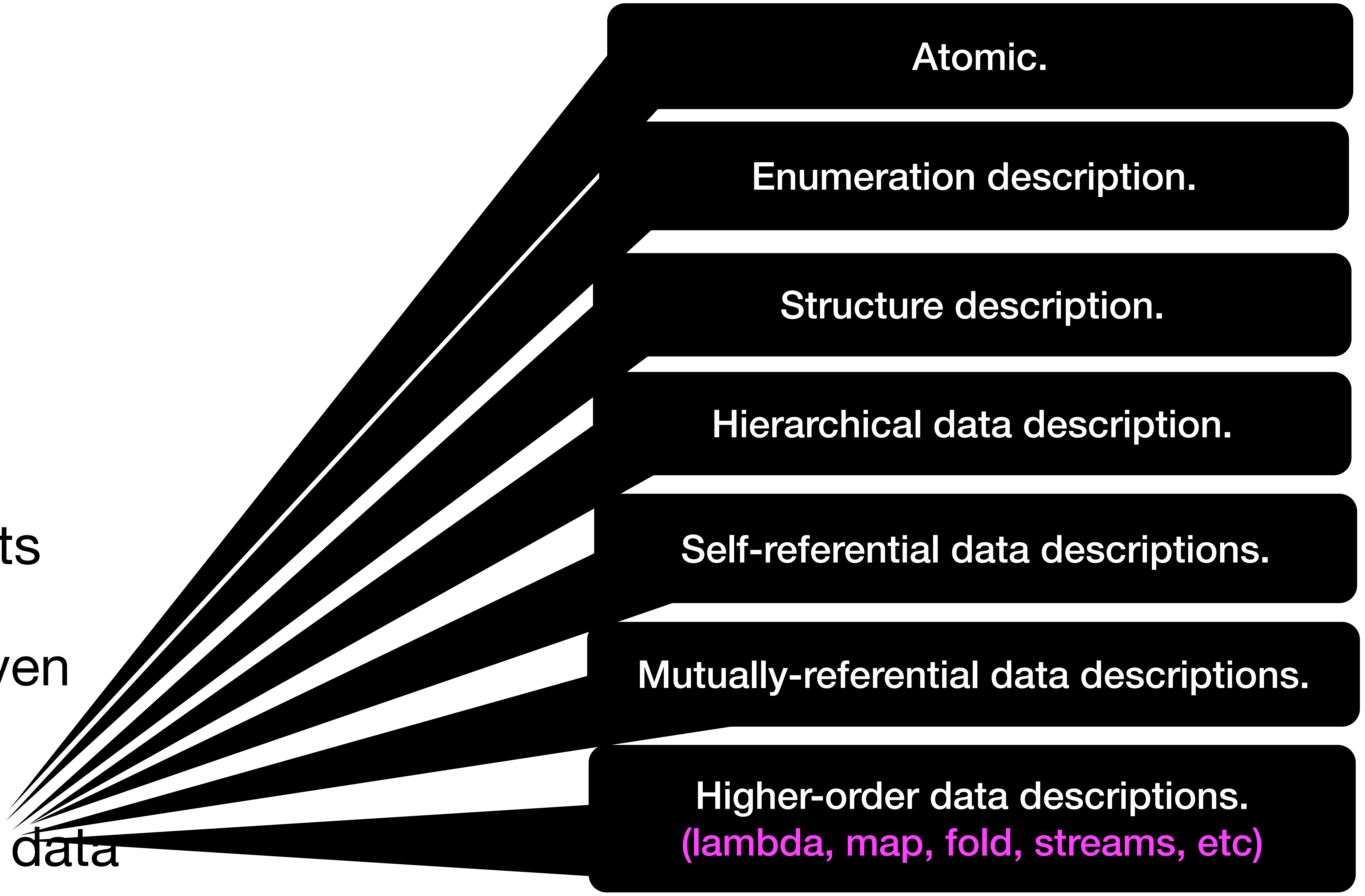
Mutually-referential data descriptions.

Higher-order data descriptions.
(lambda, map, fold, streams, etc)

Fundamentals I, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data



Atomic.

Enumeration description.

Structure description.

Hierarchical data description.

Self-referential data descriptions.

Mutually-referential data descriptions.

Higher-order data descriptions.
(lambda, map, fold, streams, etc)

with accumulators

generative recursion

Fundamentals II, the Technical Skills

Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

Fundamentals II, the Technical Skills

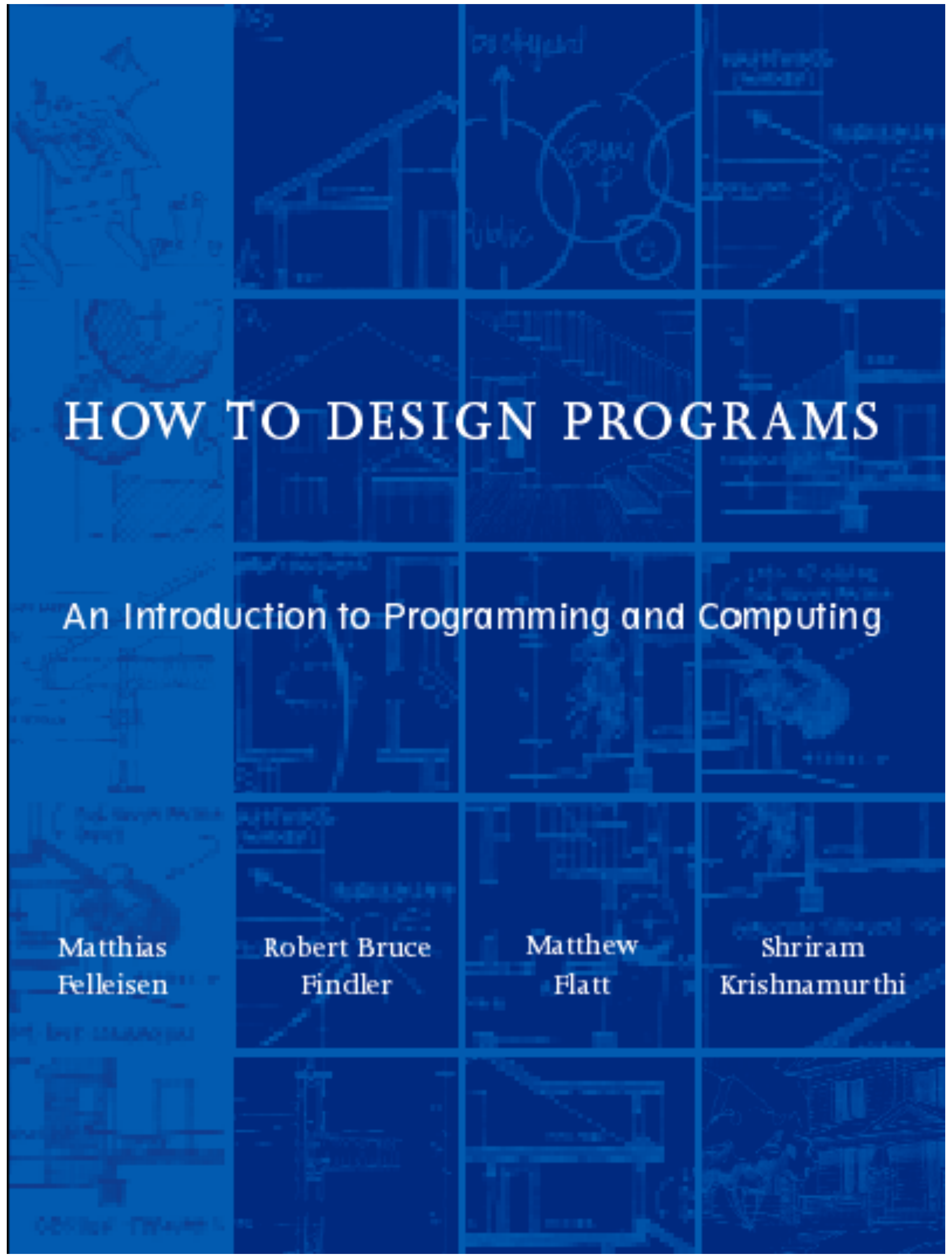
Programming, the Technical Skill

- break down the process
- study intermediate products
- practice good habits for even the simplest problems
- increase the complexity of data

+ classes and objects
+ types

… but otherwise, it repeats the basics:
develop code systematically

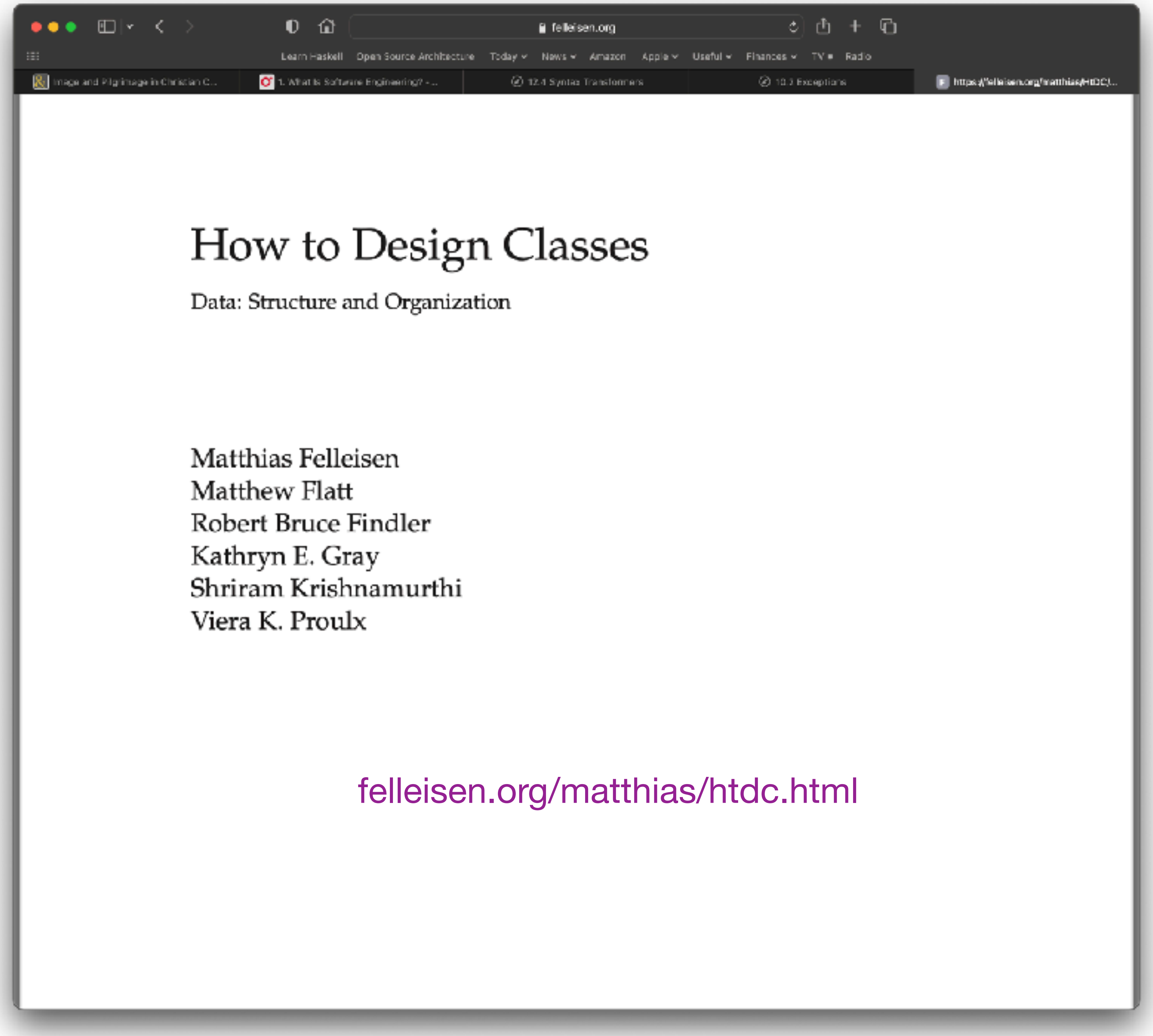
Fundamentals I & II, the Technical Skills



HOW TO DESIGN PROGRAMS

An Introduction to Programming and Computing

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi



How to Design Classes

Data: Structure and Organization

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

felleisen.org/matthias/htdc.html

Fundamentals I, the so-called "soft" skills

Share a screen, speak aloud what they think, question everything, teach each other.

Change partners because that’s life (and good for dissolving ill-matched pairs).

Social Interaction about Programming

- programming is thinking, thinking is best done with others
- practice proper pair programming
- confront students with their code from a couple of weeks ago
- ask students to react to code criticisms by teaching assistants

Fundamentals I, the so-called "soft" skills

Social Interaction about Programming

- programming is thinking, thinking is best done with others
- practice proper pair programming
- confront students with their code from a couple of weeks ago
- ask students to react to code criticisms by teaching assistants

Graded are the building blocks of an evolving semester-long project.

Add features (new callbacks). Rewrite code using new PL concepts (h-o. functions).

Fundamentals I, the so-called "soft" skills

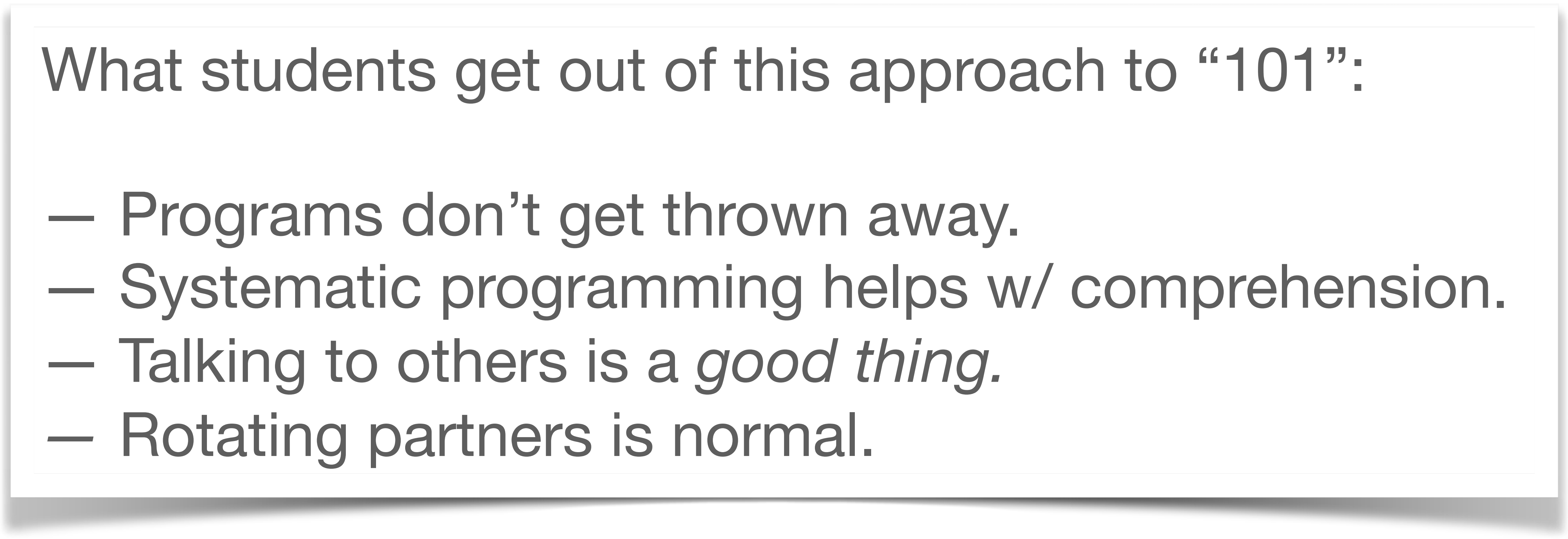
Social Interaction about Programming

- programming is thinking, thinking is best done with others
- practice proper pair programming
- confront students with their code from a couple of weeks ago
- ask students to react to code criticisms by teaching assistants

Week n: TAs leave comments on parts of a building block.

Week n+2: Students must react to the comments.

Fundamentals I, the so-called "soft" skills



What students get out of this approach to “101”:

— Programs don’t get thrown away.
— Systematic programming helps w/ comprehension.
— Talking to others is a *good thing.*
— Rotating partners is normal.

Social Interaction about Programming

- programming is thinking, thinking is best done with others

- practice proper pair programming

- confront students with their code from a couple of weeks ago

- ask students to react to code criticisms by teaching assistants

Software Development (“Hell”)

(not software engineering)

Software Development: Its Context

Year 5: Co-op 3; electives in AI, Big Data, Compilers, …

Year 4: Co-op 2; electives in AI, Big Data, Compilers, …

Year 3: Software Development

Year 2: Fundamentals III; opt.: Algorithms, Co-op 1

Year 1: Fundamentals I & II; Discrete; optionally: Logic.

Software Development: Overview

Goal: distributed board game, autonomous players

Software Development: Overview

Goal: distributed board game, autonomous players

Choose and explore a programming language & eco. system.

~2 weeks

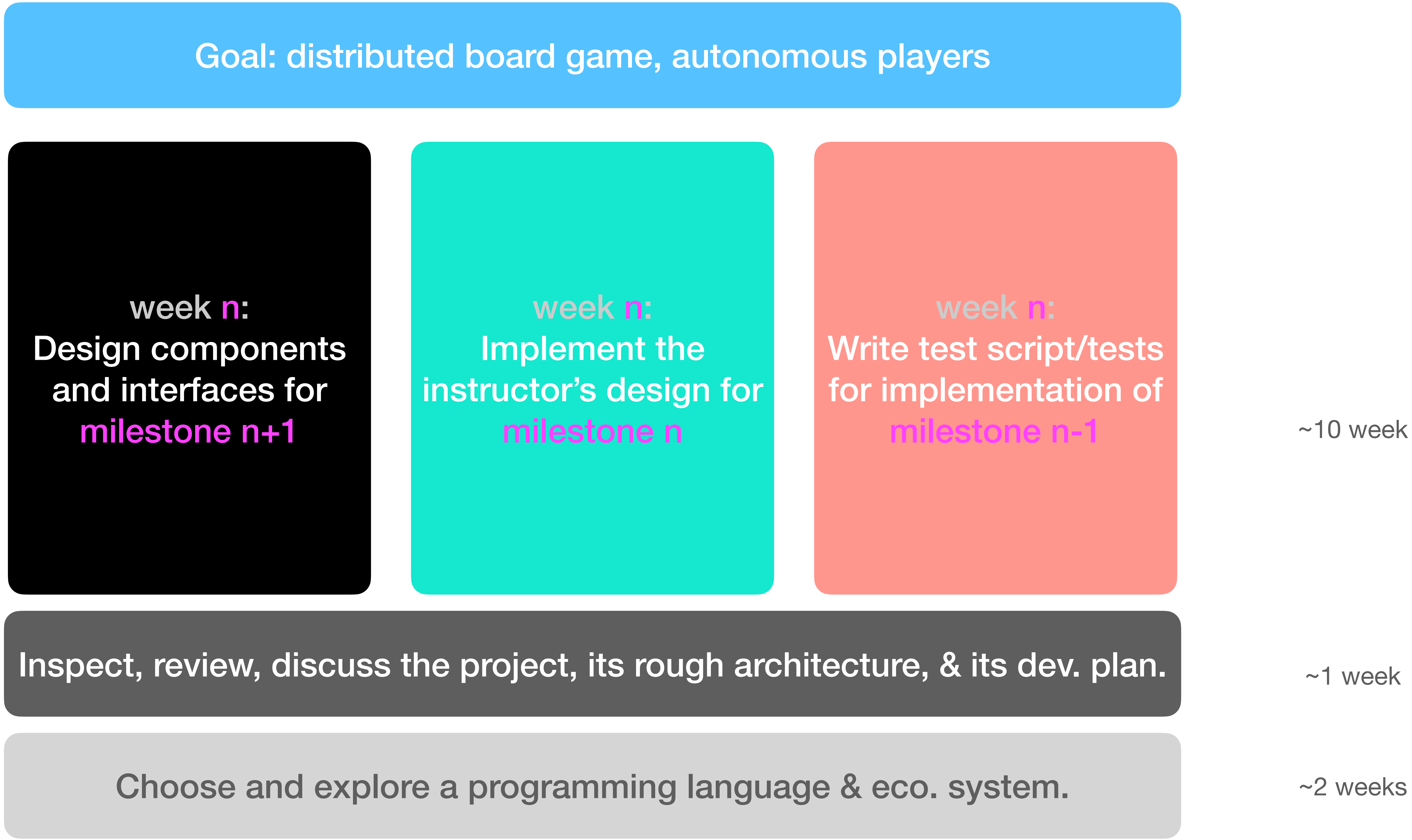
Software Development: Overview

Goal: distributed board game, autonomous players

Inspect, review, discuss the project, its rough architecture, & its dev. plan. ~1 week

Choose and explore a programming language & eco. system. ~2 weeks

Software Development: Overview



Goal: distributed board game, autonomous players

week n:
Design components and interfaces for milestone n+1

week n:
Implement the instructor's design for milestone n

week n:
Write test script/tests for implementation of milestone n-1

~10 week

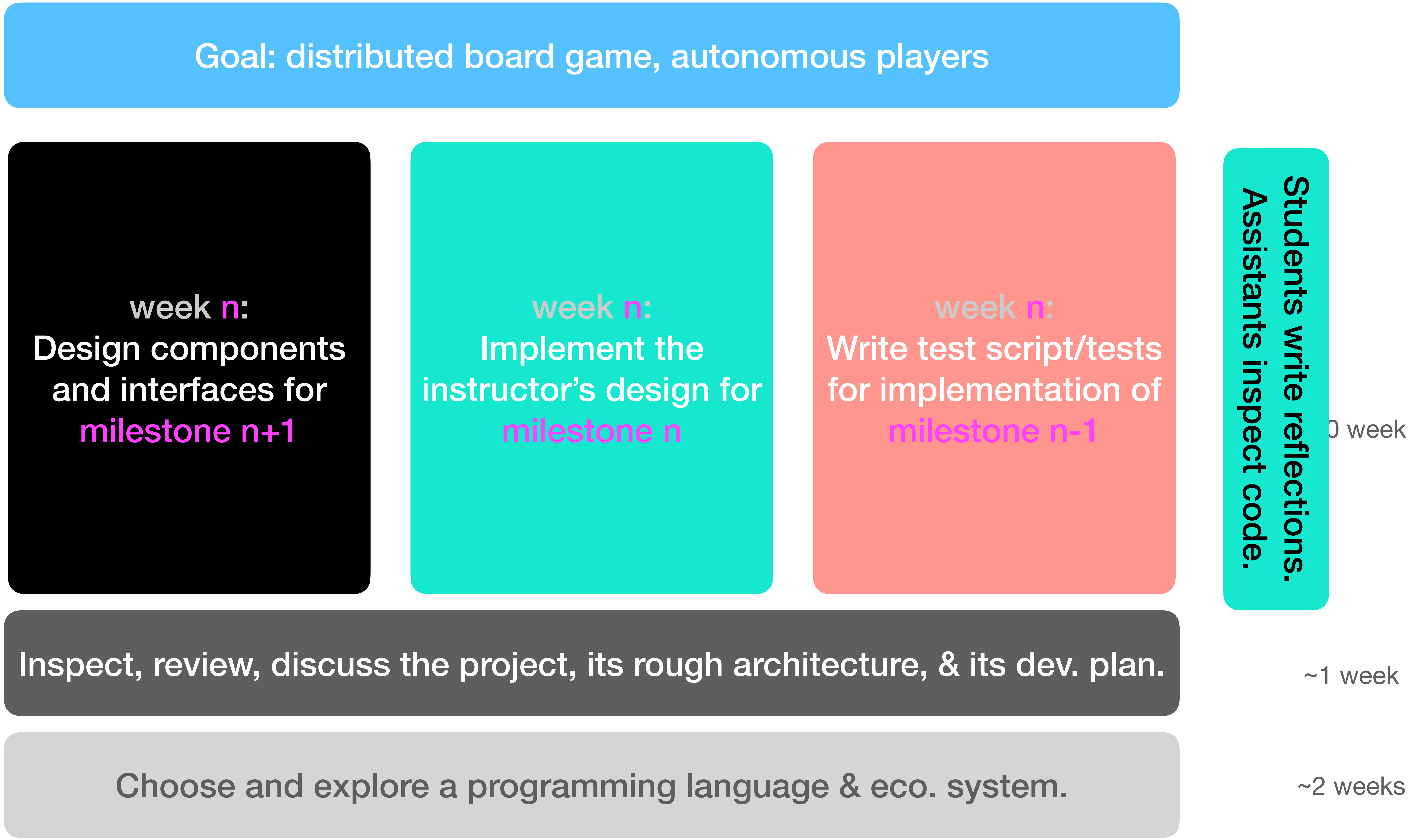
Inspect, review, discuss the project, its rough architecture, & its dev. plan.

~1 week

Choose and explore a programming language & eco. system.

~2 weeks

Software Development: Overview



Goal: distributed board game, autonomous players

week n:
Design components and interfaces for milestone n+1

week n:
Implement the instructor’s design for milestone n

week n:
Write test script/tests for implementation of milestone n-1

Students write reflections.
Assistants inspect code.

0 week

Inspect, review, discuss the project, its rough architecture, & its dev. plan.

~1 week

Choose and explore a programming language & eco. system.

~2 weeks

Software Development: Overview

Goal: distributed board game, autonomous players

week n:
Design components and interfaces for milestone n+1

week n:
Implement the instructor's design for milestone n

week n:
Write test script/tests for implementation of milestone n-1

Students write reflections.
Assistants inspect code.

0 week

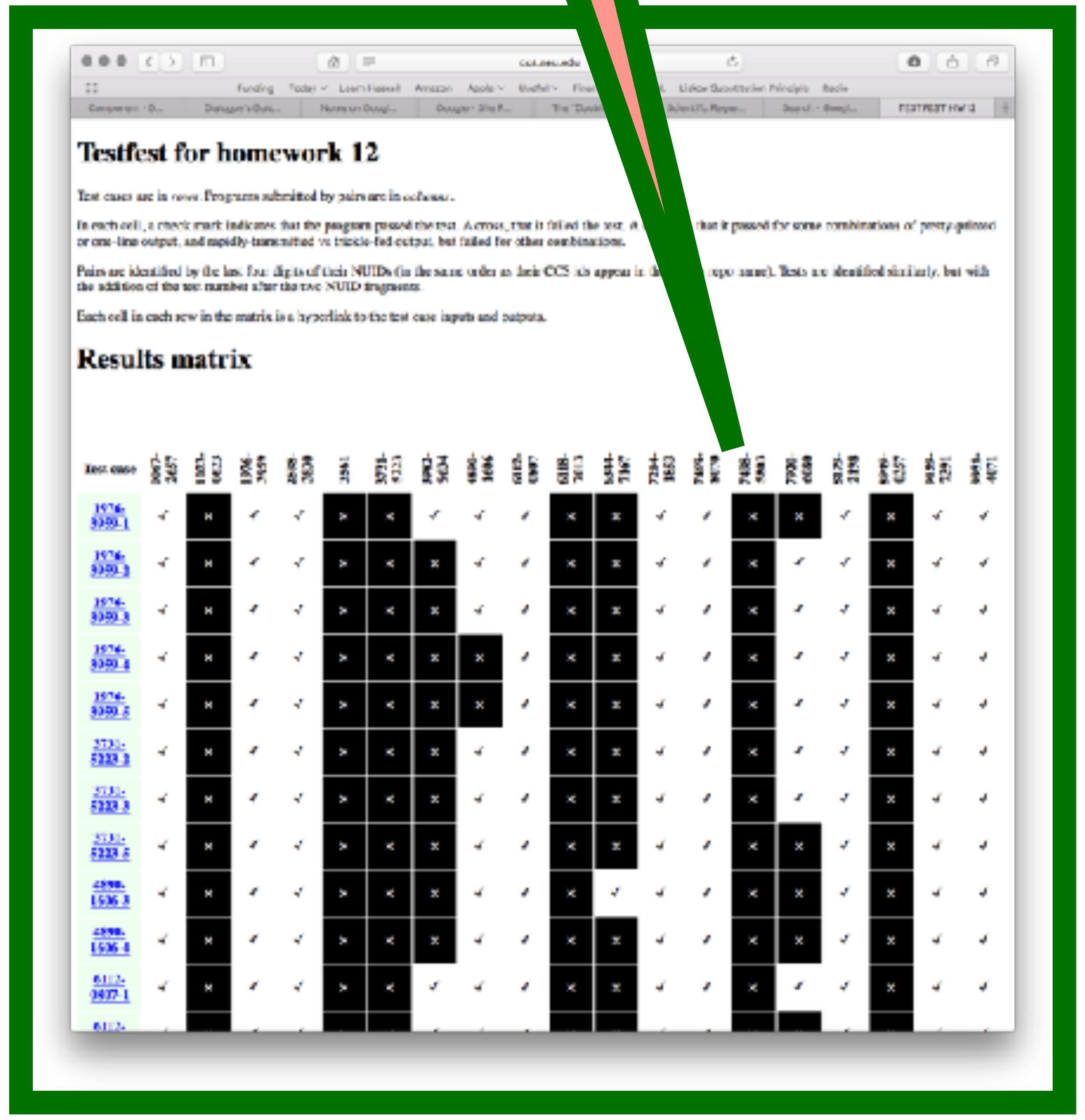
Inspect, review, discuss the project, its rough architecture, & its dev. plan.

~1 week

Choose and explore a programming language & eco. system.

~2 weeks

test fests: run everyone's tests against everyone's



Testfest for homework 12

Test cases are in rows. Programs submitted by pairs are in columns.

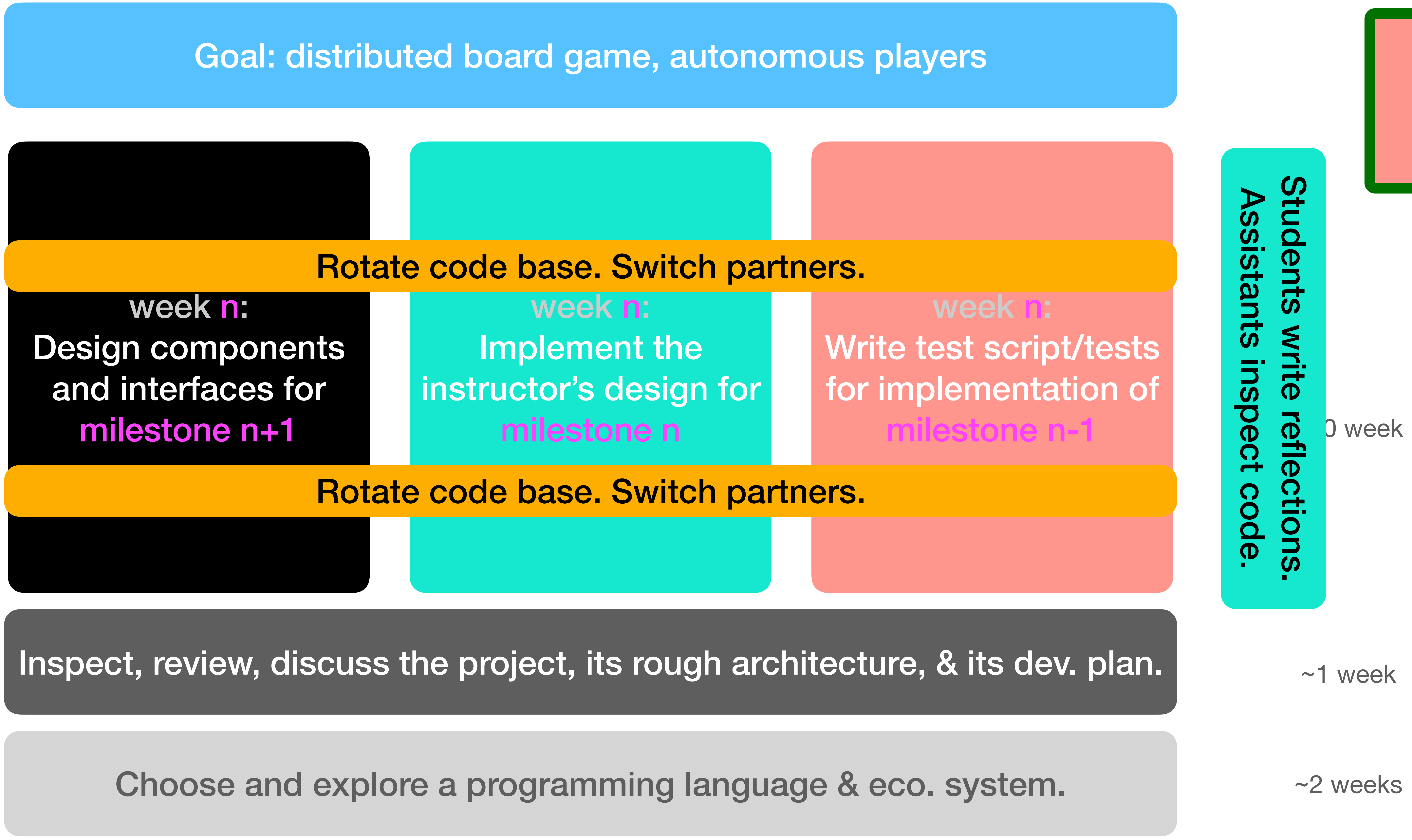
In each cell, a check mark indicates that the program passed the test. A cross, that it failed the test. A [illegible] that it passed for some combinations of pretty-printed or one-line output, and rapidly-transmitted vs trickle-fed output, but failed for other combinations.

Pairs are identified by the last four digits of their NUIDs (in the same order as their CCS ids appear in [illegible] repo name). Tests are identified similarly, but with the addition of the test number after the two NUID fragments.

Each cell in each row in the matrix is a hyperlink to the test case inputs and outputs.

Results matrix

Software Development: Overview



Goal: distributed board game, autonomous players

Rotate code base. Switch partners.

week n:
Design components and interfaces for milestone n+1

week n:
Implement the instructor's design for milestone n

week n:
Write test script/tests for implementation of milestone n-1

Rotate code base. Switch partners.

Students write reflections.
Assistants inspect code.

0 week

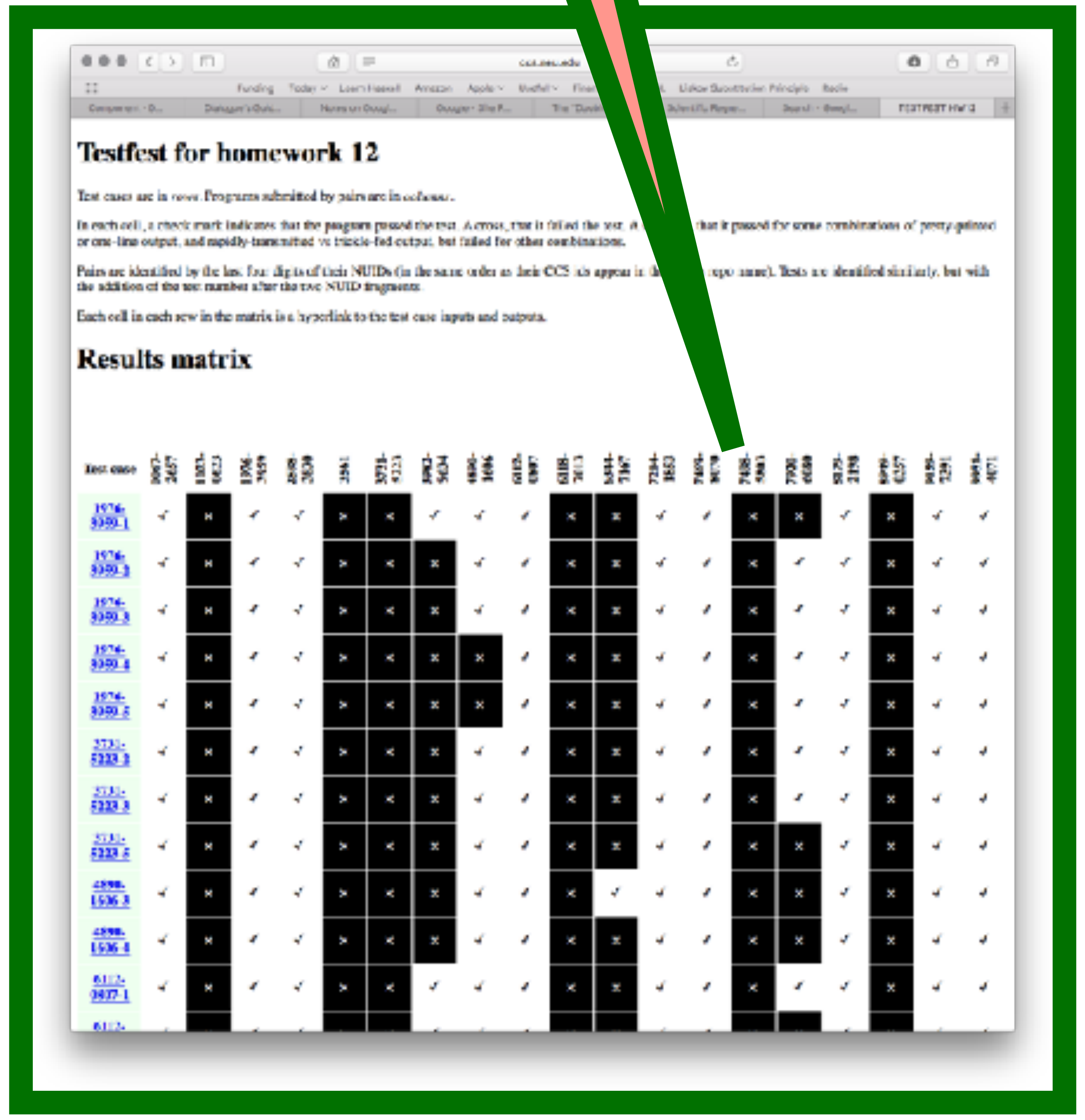
Inspect, review, discuss the project, its rough architecture, & its dev. plan.

~1 week

Choose and explore a programming language & eco. system.

~2 weeks

test fests: run everyone's tests against everyone's



Testfest for homework 12

Test cases are in rows. Programs submitted by pairs are in columns.

In each cell, a check mark indicates that the program passed the test. A cross, that it failed the test. A [illegible] that it passed for some combinations of pretty-printed or one-line output, and rapidly-transmitted vs trickle-fed output, but failed for other combinations.

Pairs are identified by the last four digits of their NUIDs (in the same order as their CCS ids appear in [illegible] repo name). Tests are identified similarly, but with the addition of the test number after the two NUID fragments.

Each cell in each row in the matrix is a hyperlink to the test case inputs and outputs.

Results matrix

Software Development: Its Goals

To each student: choose your favorite programming language

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers
- inspecting code as a panelist with the goal of finding design flaws and bugs
- reflecting on code; writing about code

Software Development: Coding Details

Students pick emotionally. Fashion rules.

(Self-selection suggests quality of code is somewhat related to choice of PL.)

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- "grace under pressure" systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Students pick emotionally. Fashion rules.
(Self-selection suggests quality of code is somewhat related to choice of PL.)

Students write external and/or internal specifications per milestone.
Teaching assistants check minimal standards.
Learning from compare and reflect.

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Students pick emotionally. Fashion rules.
(Self-selection suggests quality of code is somewhat related to choice of PL.)

Students write external and/or internal specifications per milestone.
Teaching assistants check minimal standards.
Learning from compare and reflect.

Coding a non-trivial component per week and presenting them is intentional.
Partner and code-base rotation add stress.
Teaching assistants check minimal standards using the ideas from Fundamentals I through III.

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Students pick emotionally. Fashion rules.
(Self-selection suggests quality of code is somewhat related to choice of PL.)

Students write external and/or internal specifications per milestone.
Teaching assistants check minimal standards.
Learning from compare and reflect.

Coding a non-trivial component per week and presenting them is intentional.
Partner and code-base rotation add stress.
Teaching assistants check minimal standards using the ideas from Fundamentals I through III.

The *remote-proxy* pattern is the the only new design technique they encounter.

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Student: “We don’t know how to write unit tests for this function. It’s too long."

Staff: “*Fundamentals* teach you to work through examples first; write tests; keep methods short (‘atomic’ xor ‘composite’).”

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Student: “We don’t know how to write unit tests for this function. It’s too long."

Staff: “*Fundamentals* teach you to work through examples first; write tests; keep methods short (‘atomic’ xor ‘composite’).”

Code: `int distance;`
Student: “I think it is the distance between the left of the car and the lane marker on the street.”
Staff: “*Fundamentals* teach about data representation. If ou don’t remember now, how will the maintain of the code?”

Software Development: Coding Details

Programming, the Technical Skill

- get to know a PL eco. sys. *in depth*
- designing components & interfaces
- “grace under pressure” systematic program development
- a first taste: a systematically developed distributed system with some failure tolerance

Student: “We don’t know how to write unit tests for this function. It’s too long."

Staff: “*Fundamentals* teach you to work through examples first; write tests; keep methods short (‘atomic’ xor ‘composite’).”

Code: `int distance;`
Student: “I think it is the distance between the left of the car and the lane marker on the street.”
Staff: “*Fundamentals* teach about data representation. If ou don’t remember now, how will the maintain of the code?”

Student: “We didn’t have time to write unit tests, because we had to do so much debugging.”

Staff: “Fundamentals I, II, and III teach you to write unit tests to reduce debugging time.”

Software Development: “Soft” Skills

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

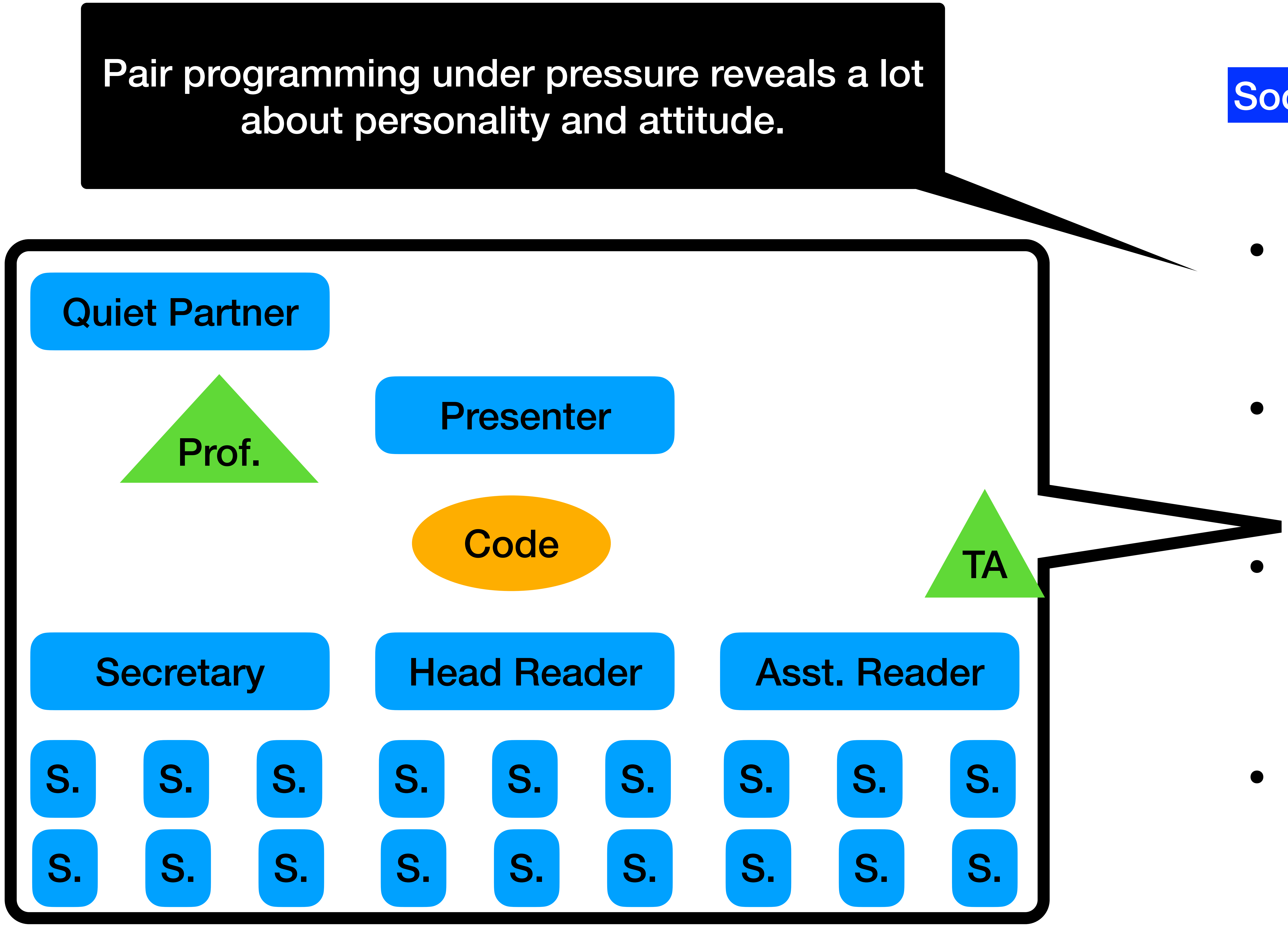
Software Development: “Soft” Skills

Pair programming under pressure reveals a lot about personality and attitude.

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

Software Development: “Soft” Skills



Pair programming under pressure reveals a lot about personality and attitude.

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

Software Development: Soft Skills

Pair programming under pressure reveals a lot about personality and attitude.

There is nothing like a formal code review, eye-to-eye, that brings out what it means to “code as if the next guy to take on the code matters.”

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

Software Development: Soft Skills

Pair programming under pressure reveals a lot about personality and attitude.

There is nothing like a formal code review, eye-to-eye, that brings out what it means to “code as if the next guy to take on the code matters.”

Describe issues with presented code.

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

Software Development: Soft Skills

Pair programming under pressure reveals a lot about personality and attitude.

There is nothing like a formal code review, eye-to-eye, that brings out what it means to “code as if the next guy to take on the code matters.”

Describe issues with presented code.

Every milestone comes with a self-evaluation: “Method m must perform three tasks: t1, t2, t3. Does your implementation of m reflect this specification? How? Where? Cite git lines.”

Social Interaction about Programming

- more pair programming; on-boarding new partners; learning to be onboarded
- presenting in public to a panel composed of peers (“egoless”)
- inspecting code in public as a panelist with the goal of finding design flaws and bugs (“egoless”)
- reflecting on code; writing about code

Software Development: Teaching It.

Programming, the Technical Skill

- Instructor must develop a new project for every semester.
- Instructor must code and practice the “classroom gospel” of coding.
- Instructor must explore design alternatives for in-class use and grading purposes.
- Instructor must write extremely hardened test scripts (and unit tests).

Social Interaction about Programming

- Instructor must manage a highly unusual classroom set-up (read, observe, control).
- Instructor must deal with student problems (“couple counseling” vs “divorces”).
- Instructor must be the “first egoless programmer”.

Software Development: Teaching It.

It’s not easy. But it is our *moral* obligation, and it is extremely rewarding.

Programming, the Technical Skill

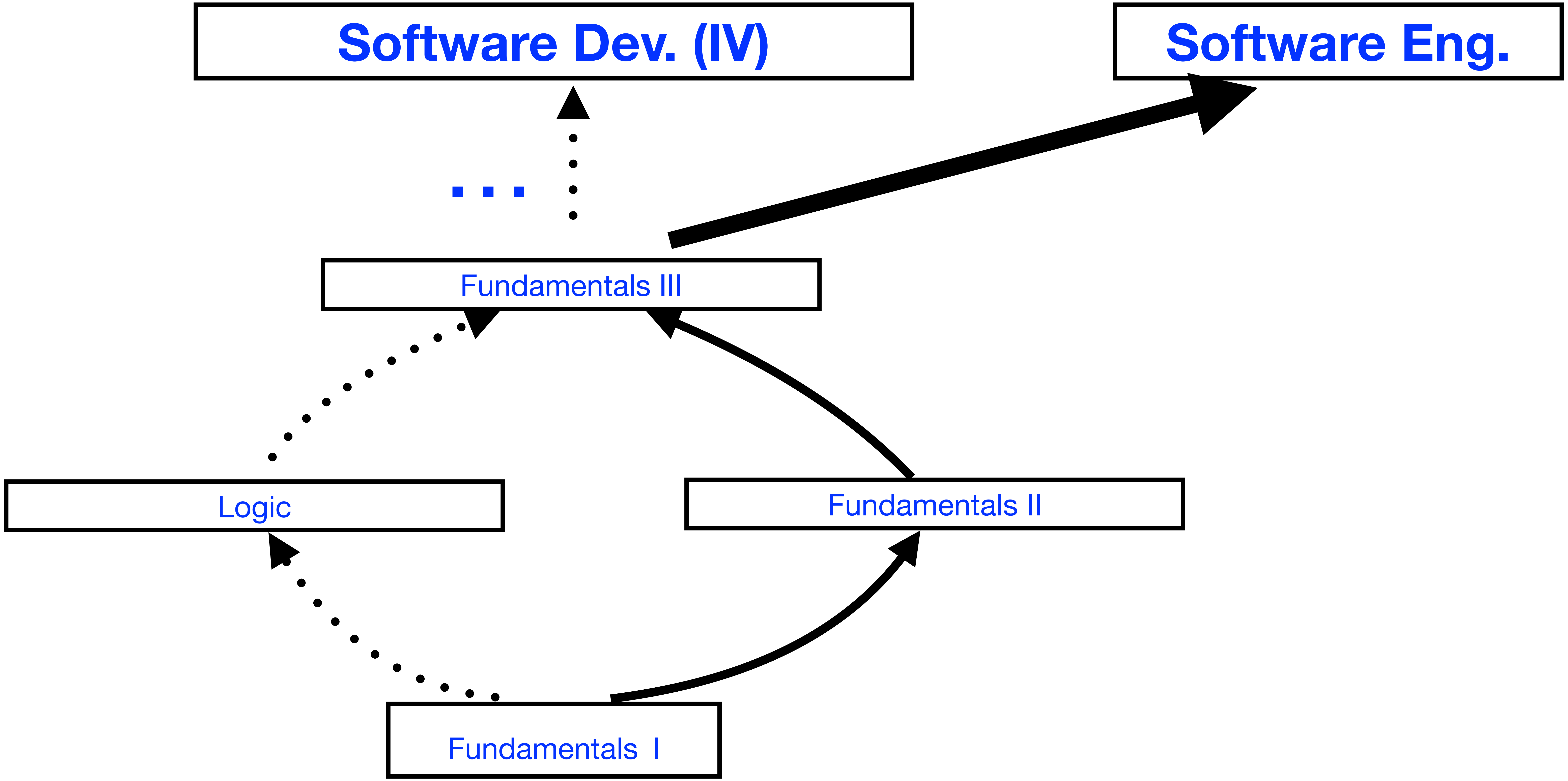
- Instructor must develop a new project for every semester.
- Instructor must code and practice the “classroom gospel” of coding.
- Instructor must explore design alternatives for in-class use and grading purposes.
- Instructor must write extremely hardened test scripts (and unit tests).

Social Interaction about Programming

- Instructor must manage a highly unusual classroom set-up (read, observe, control).
- Instructor must deal with student problems (“couple counseling” vs “divorces”).
- Instructor must be the “first egoless programmer”.

Warning

Warning: Past Reality vs Present Reality



Software Dev. (IV)

Software Eng.

...

Fundamentals III

Logic

Fundamentals II

Fundamentals I

Take Aways

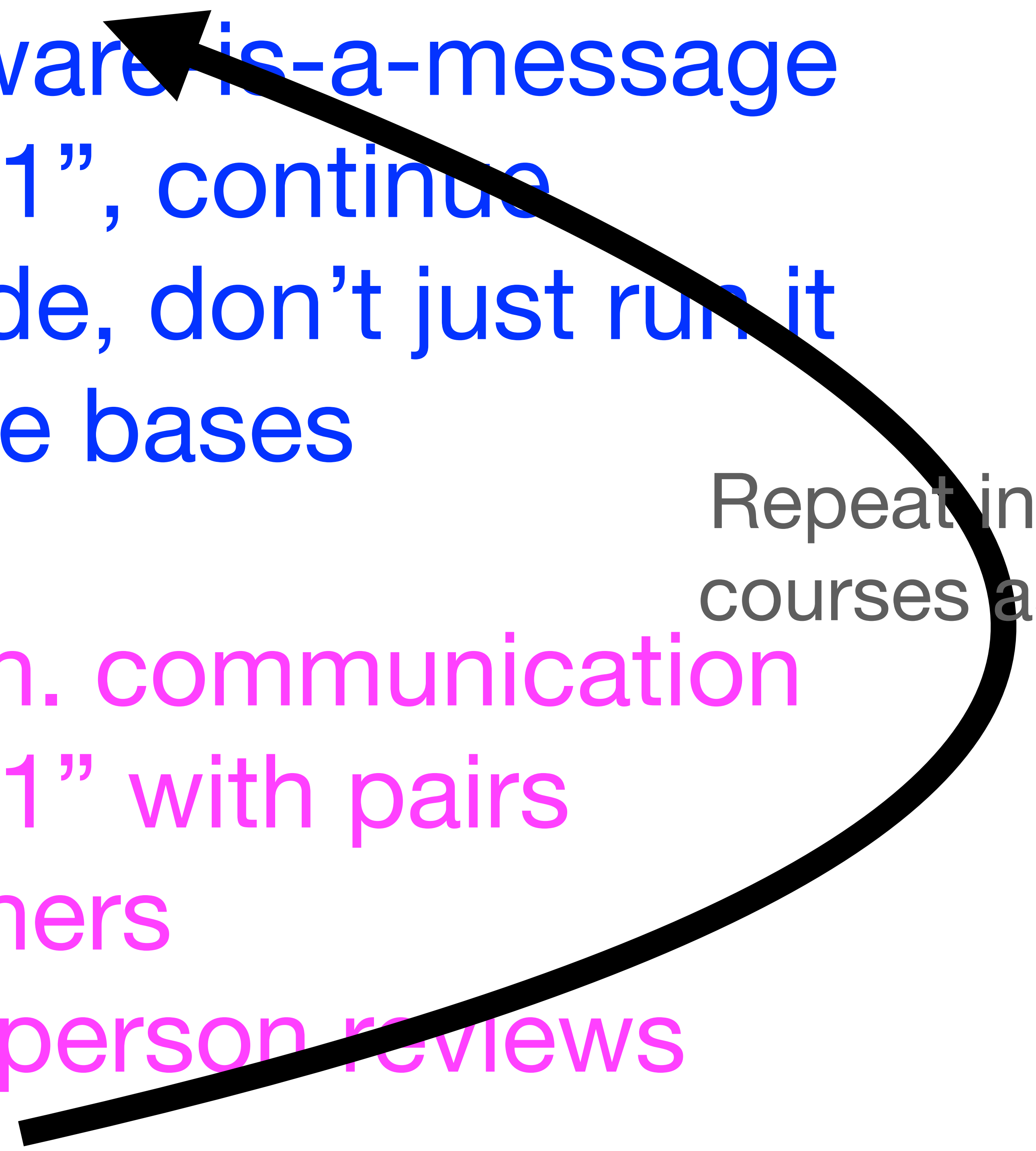
Challenges, and a Solution

Challenges, and a Solution

- teach software-is-a-message
- start in “101”, continue
- inspect code, don’t just run it
- switch code bases

- teach techn. communication
- start in “101” with pairs
- rotate partners
- grow to in-person reviews

Challenges, and a Solution



- teach software-is-a-message
- start in “101”, continue
- inspect code, don’t just run it
- switch code bases

- teach techn. communication
- start in “101” with pairs
- rotate partners
- grow to in-person reviews

Repeat in as many courses as feasible

Challenges, and a Solution

- teach software-is-a-message
- start in “101”, continue
- inspect code, don’t just run it
- switch code bases

- teach techn. communication
- start in “101” with pairs
- rotate partners
- grow to in-person reviews

Repeat in as many courses as feasible

What will *you* do?

Thanks for listening.