Chapter 8

Chapter 8 is a sort of catch-all chapter — we'll review a few things and talk about design.

Scope review

- What is scope?
- Where are each of the following in scope?
  - a local variable or parameter name
  - an instance field name
  - a static field name
  - a local declared in a `for` loop header

What is a side-effect?

How are parameters passed to methods?

(Rest assured that we’ll have an exercise on these topics later this week!)

There are four kinds of classes: *instantiable*, *tester*, *application*, and *utility*.

Instantiable classes model program-domain entities.

(What is a program-domain entity?)

What are some instantiable classes we've seen so far?

Application classes contain main methods and rely on instantiable classes to implement program tasks.

What are some application classes we’ve seen so far?

Tester classes contain methods to systematically exercise every part of another class' interface.

Utility classes cannot be instantiated; rather, they contain useful class methods.

What are some utility classes we've seen so far?

Instantiable classes model program-domain entities.

(So: what is a program-domain entity, anyway?)

A class should correspond to *a single concept.*

(What is a concept?)

Some classes correspond to tangible real-world things: Clicker, Employee, CashRegister.

Some classes correspond to abstract concepts.

(What are some example abstract concepts?)

Some classes are actors, which manipulate instances of other concepts.

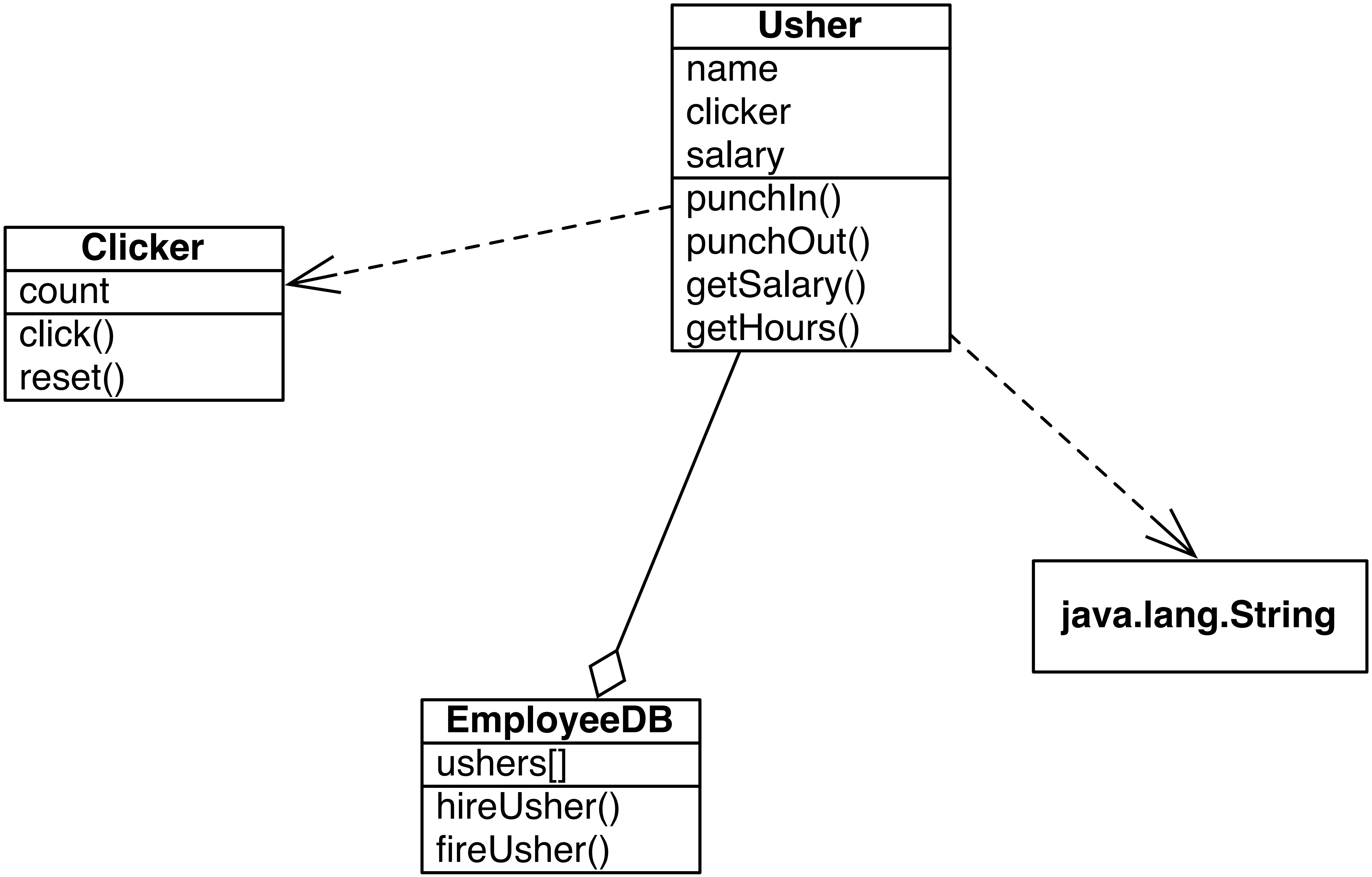
(What are some examples?)

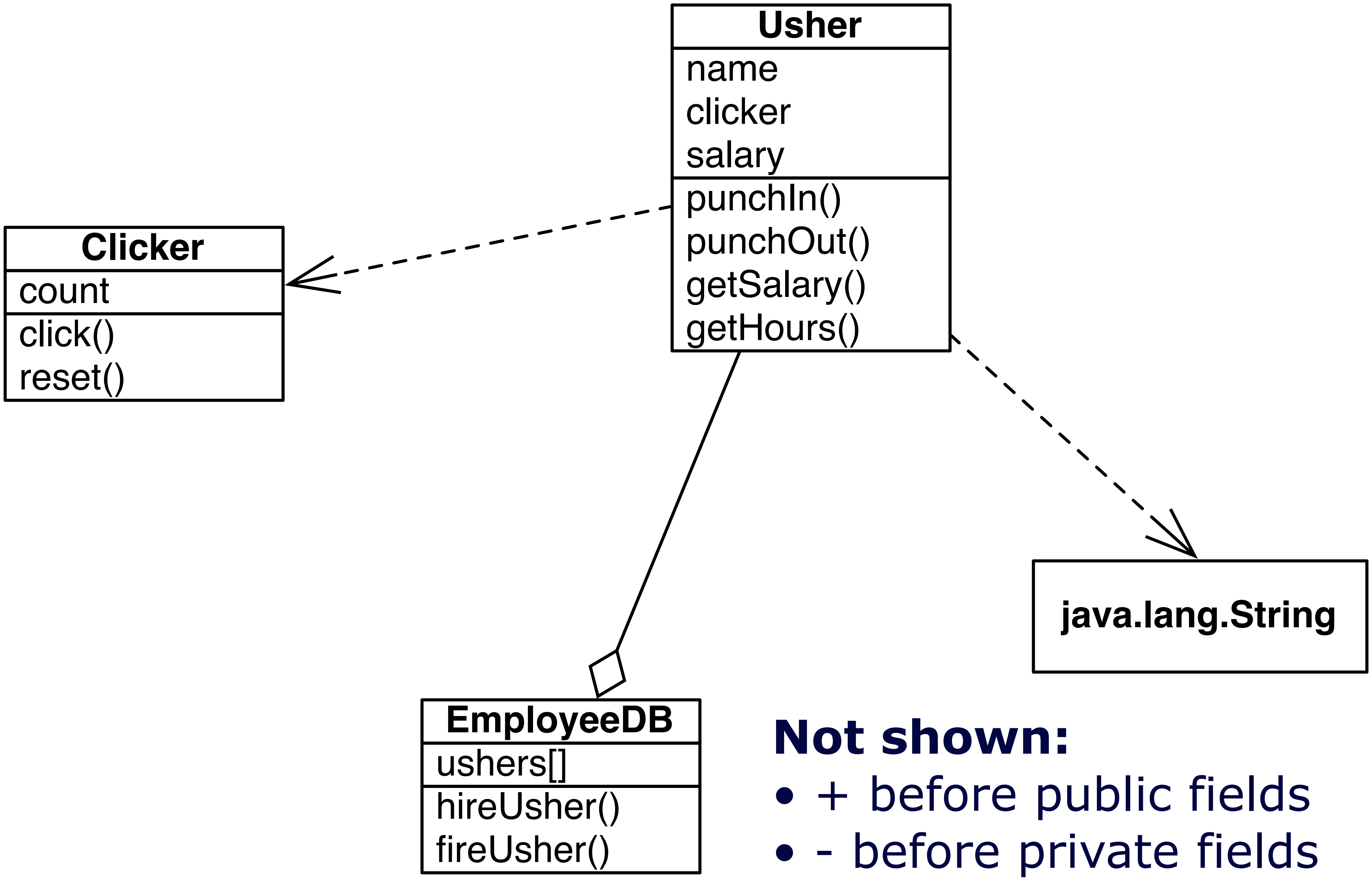
Given a problem description, you should be able to *choose classes* by identifying relevant concepts.

Is there a good, standard way to talk about these designs?

UML

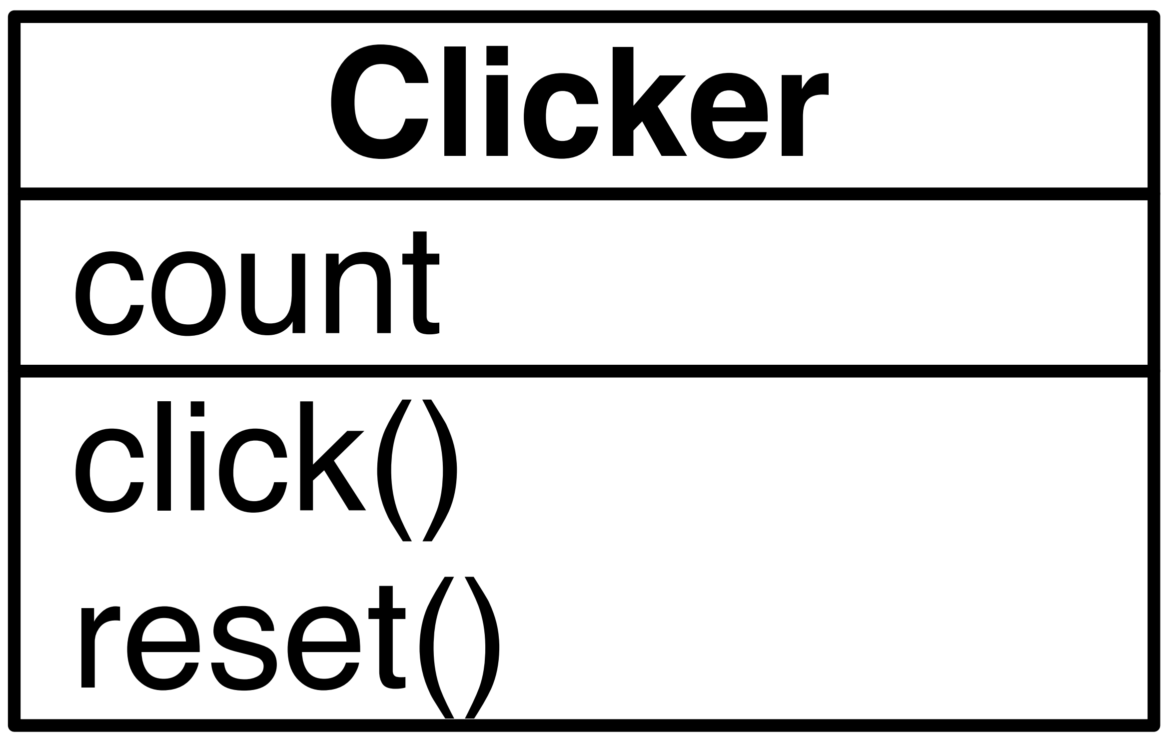
- Simple way to show classes and relationships between classes
- Describes all fields and methods declared in a class
- Describes relationships (dependence, aggregation, subtyping) between classes and interfaces

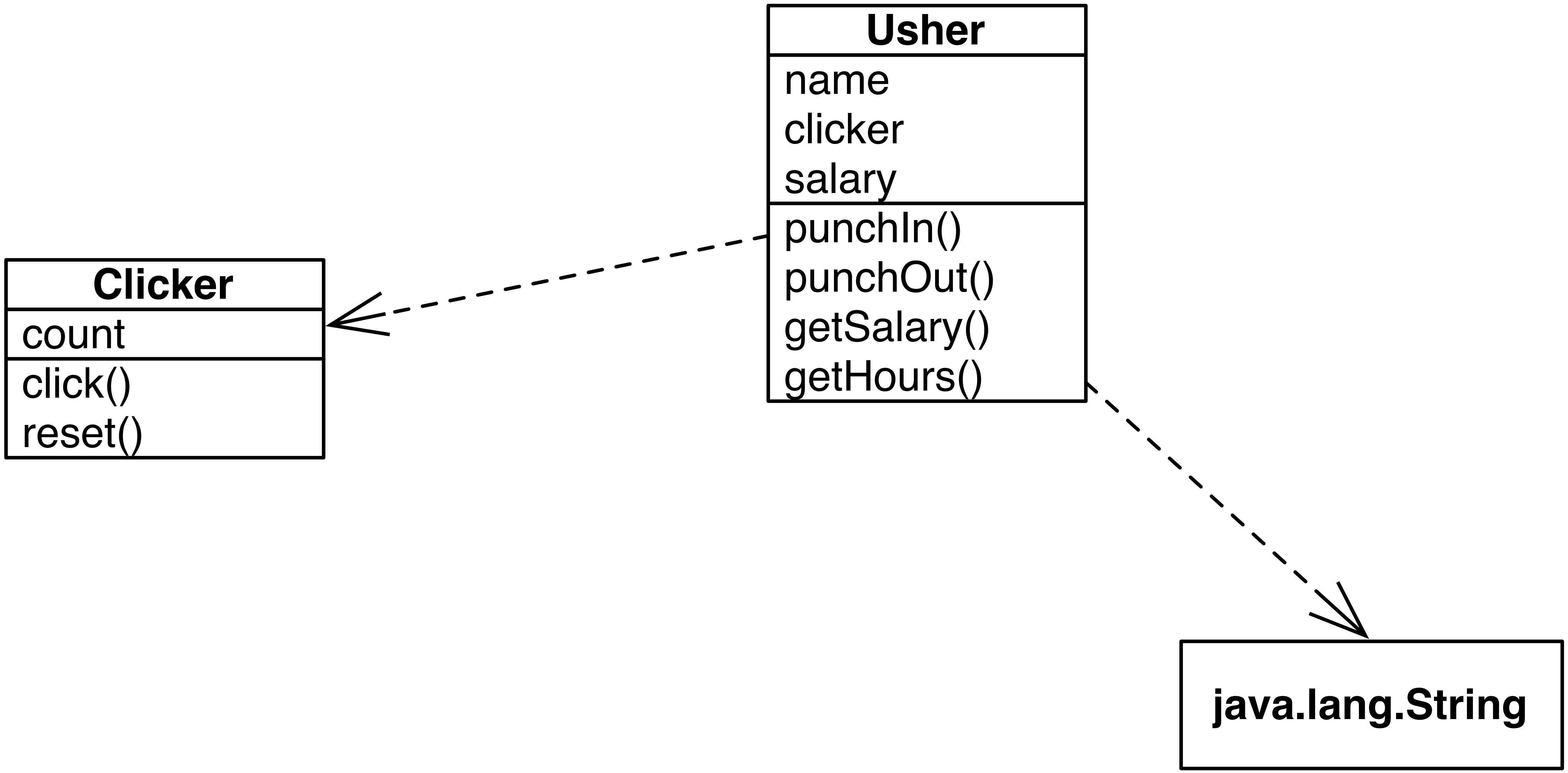

Usher
name
clicker
salary
punchIn()
punchOut()
getSalary()
getHours()
Clicker
count
click()
reset()
EmployeeDB
ushers[]
hireUsher()
fireUsher()
java.lang.String

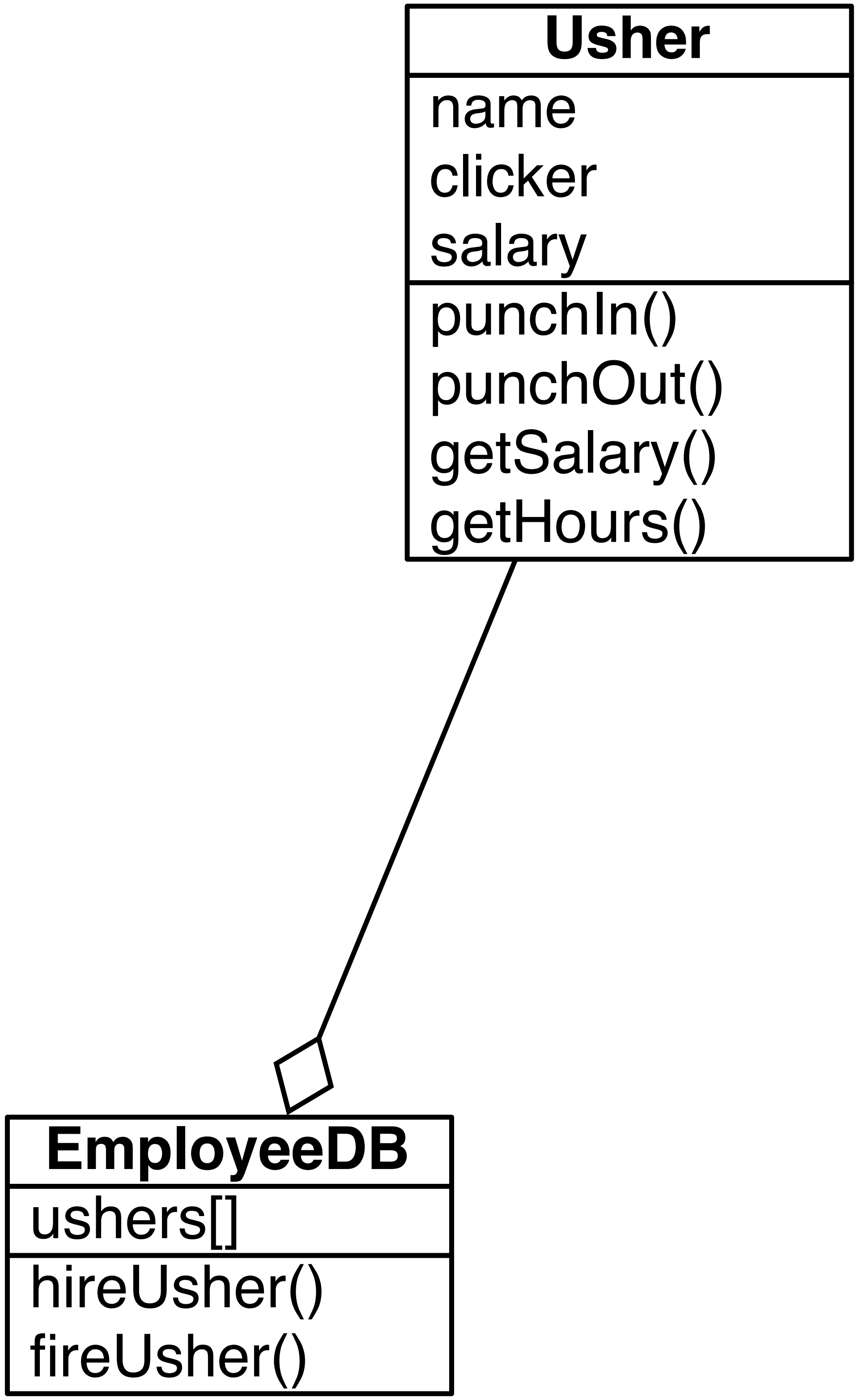


Not shown:

- + before public fields
- - before private fields
- static fields and methods should be <u>underlined</u>

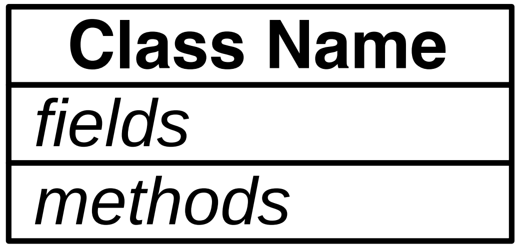

Clicker
count
click()
reset()

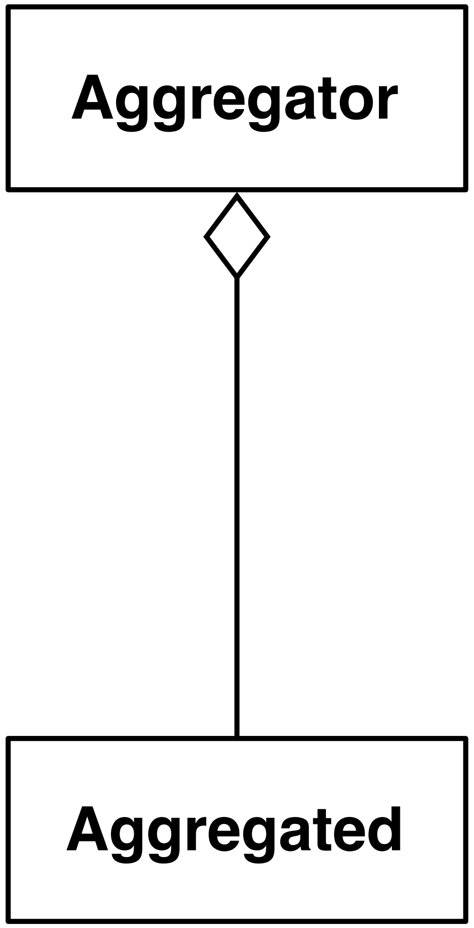

Usher
name
clicker
salary
punchIn()
punchOut()
getSalary()
getHours()
Clicker
count
click()
reset()
java.lang.String

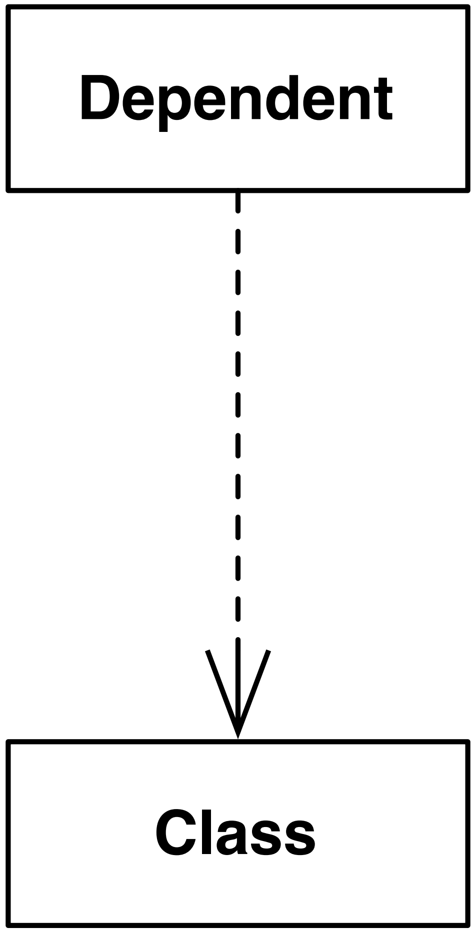

Usher
name
clicker
salary
punchIn()
punchOut()
getSalary()
getHours()
EmployeeDB
ushers[]
hireUsher()
fireUsher()

Basic UML

(subject to revision)







(See Chapter 17.1 for more!)

A *cohesive* class is one that only implements a single concept.

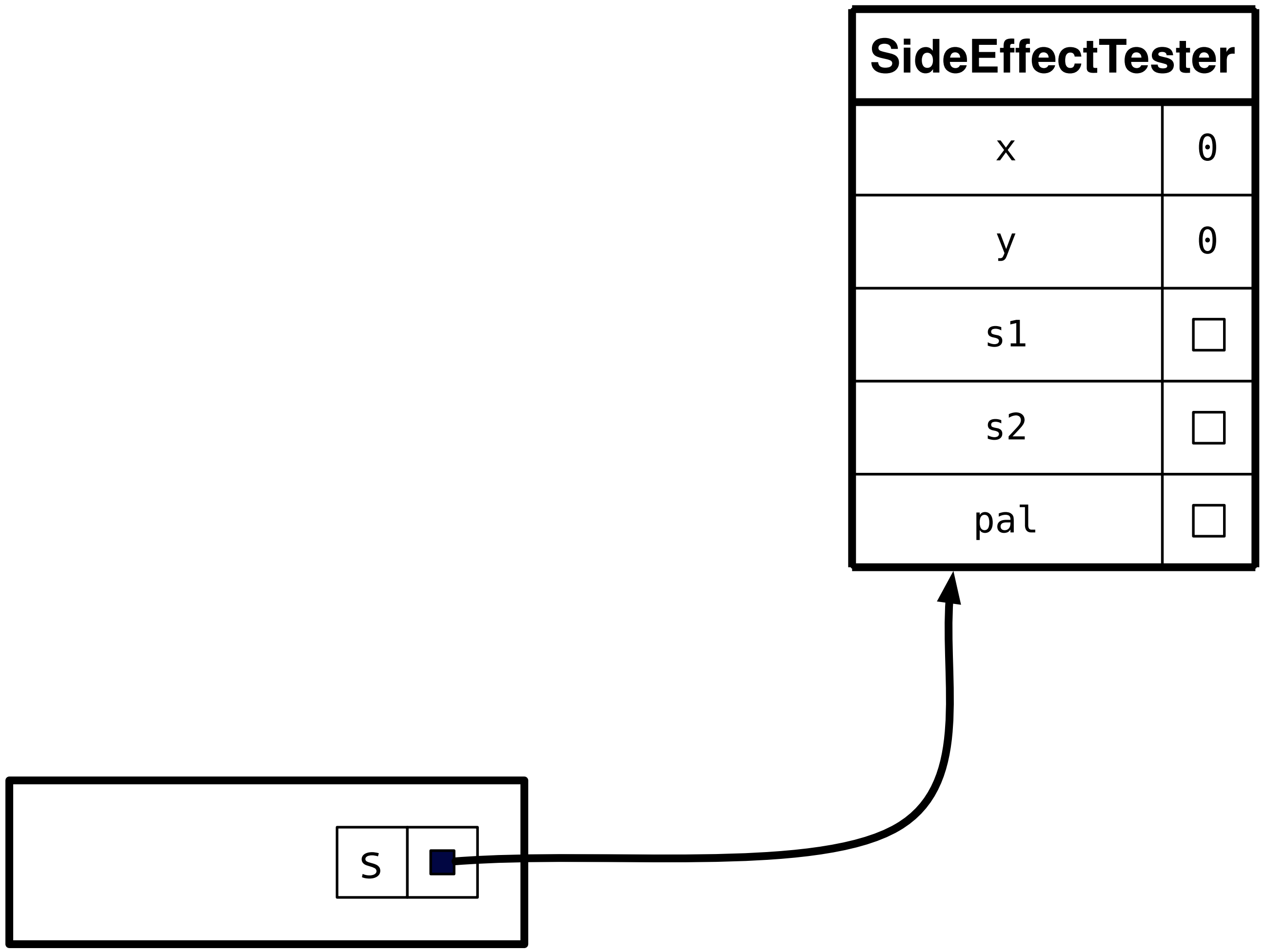
Cohesive or not?

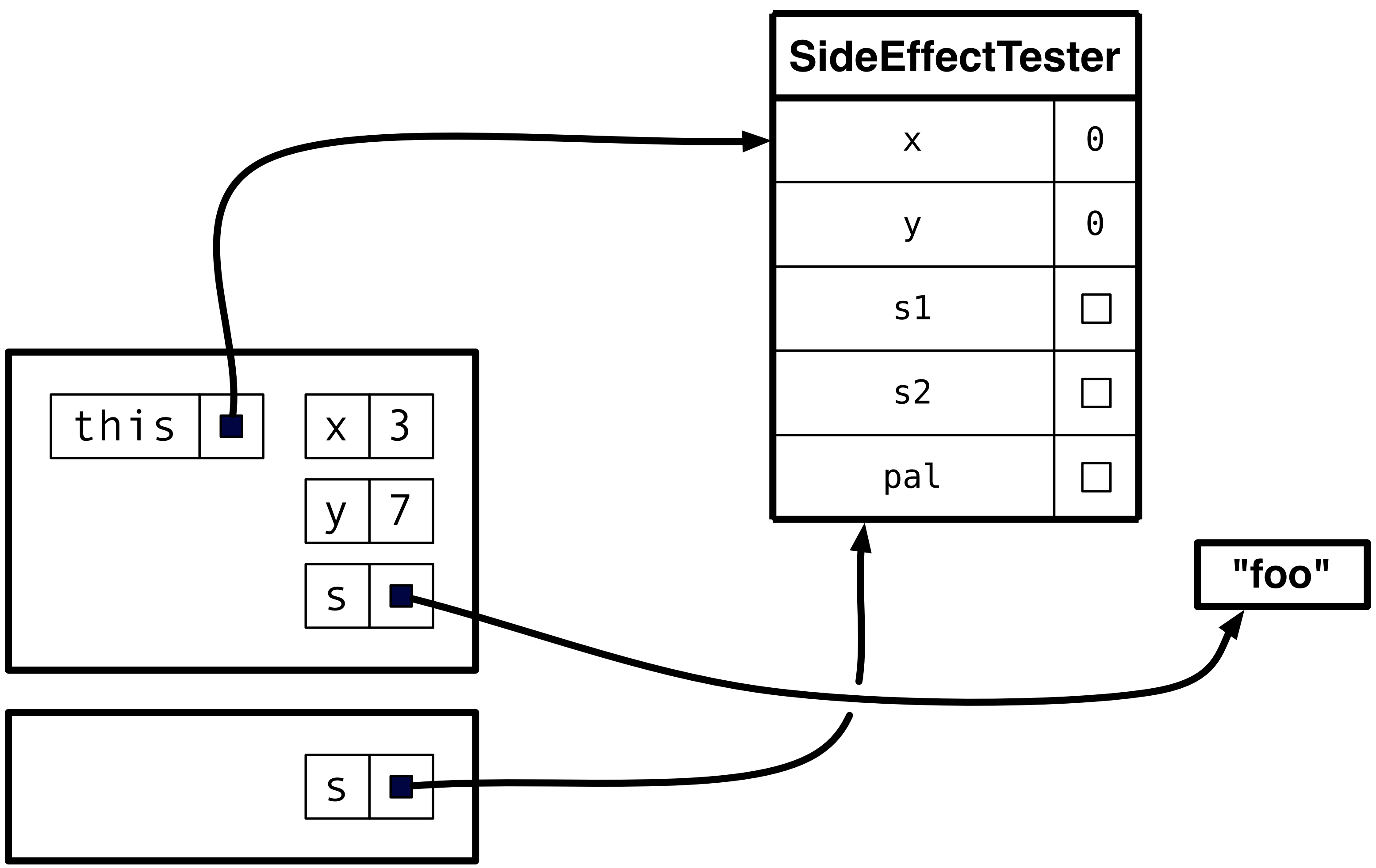
- Coin class
- CashRegister class that knows about U.S. currencies (or “all major currencies”)
- Temperature class that includes getCelsius(), getFahrenheit() methods
- Email client that can install software on your computer
- Color class that knows about *color spaces*

Homework exercise: Find one example of a real-world product that is *not* cohesive and email it to the class list.

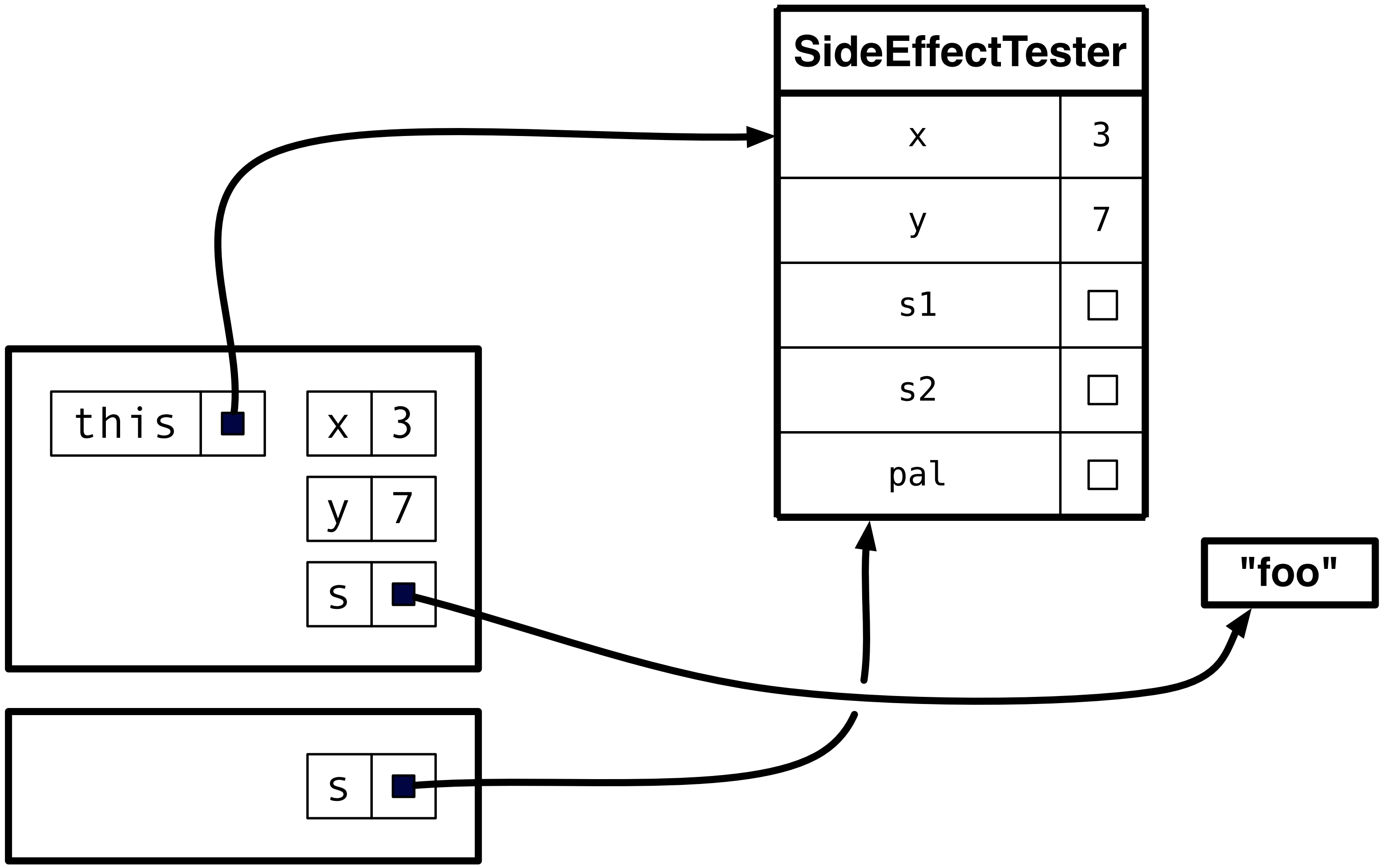
SideEffectTester: constructors

```
s = new SideEffectTester(3, 7, "foo");
```

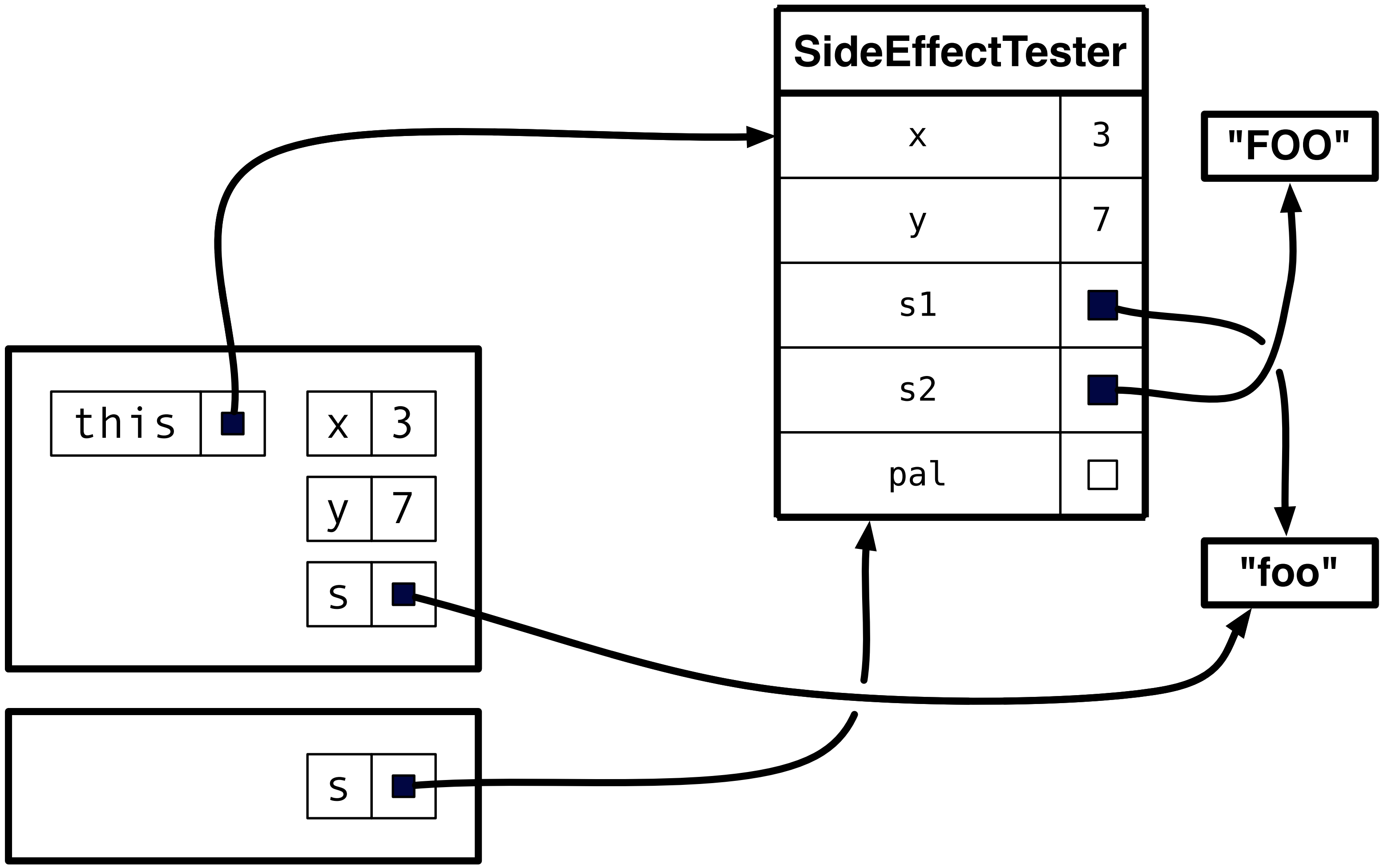



SideEffectTester
x
0
y
0
s1
s2
pal
"foo"
this
x
3
y
7
s
s

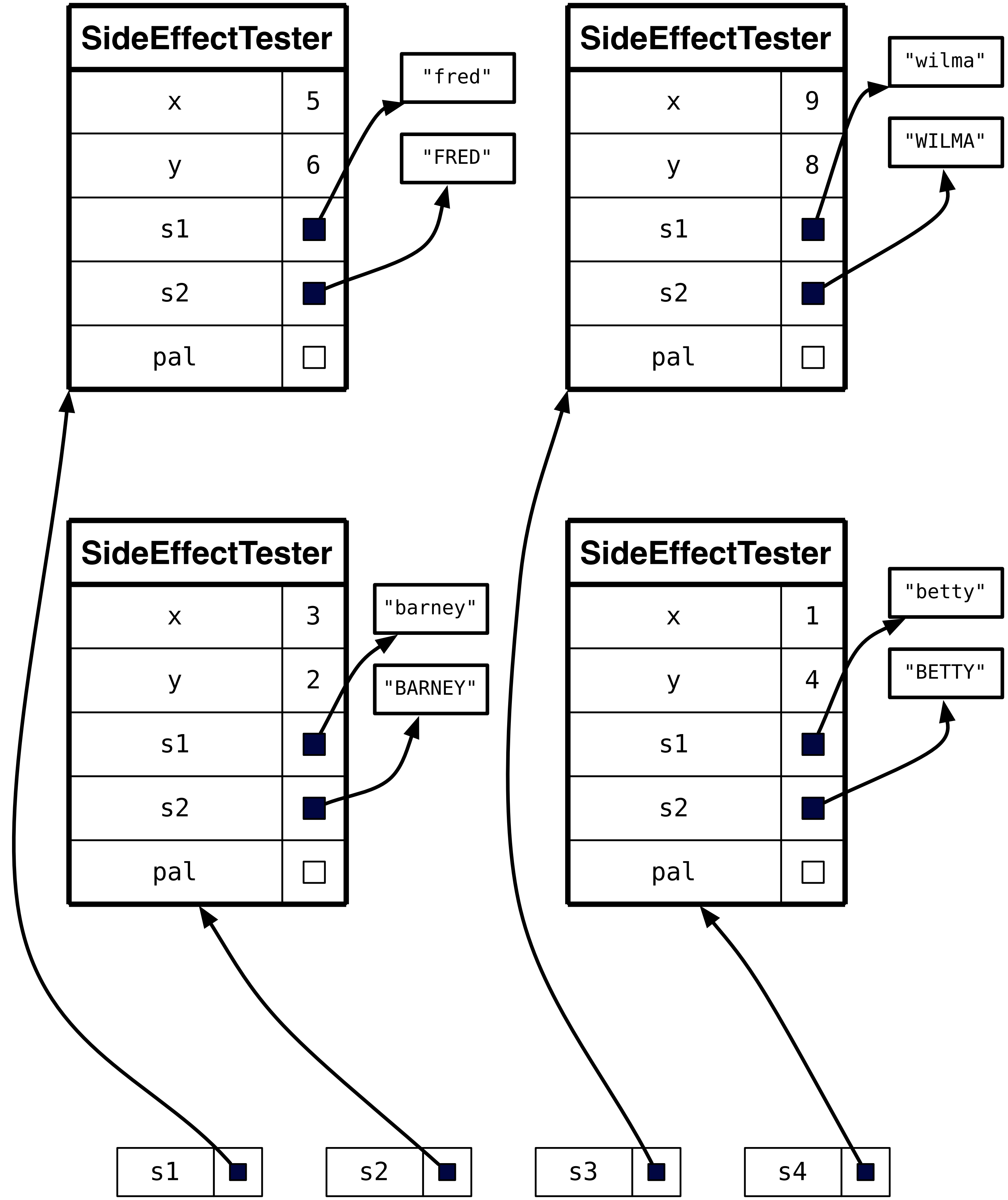
```
this.x = x;
this.y = y;
```



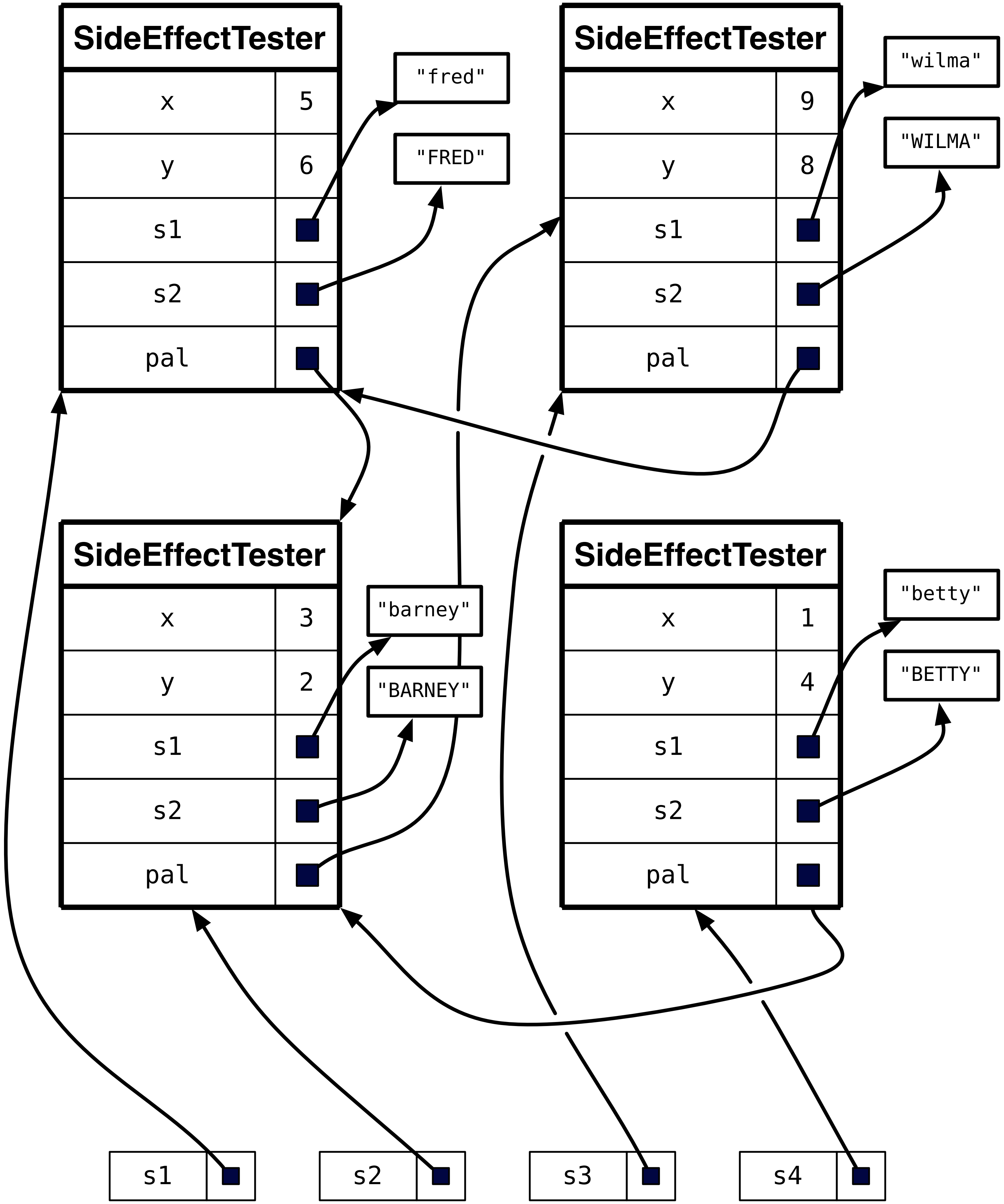
```
s1 = s;
s2 = s.toUpperCase();
```



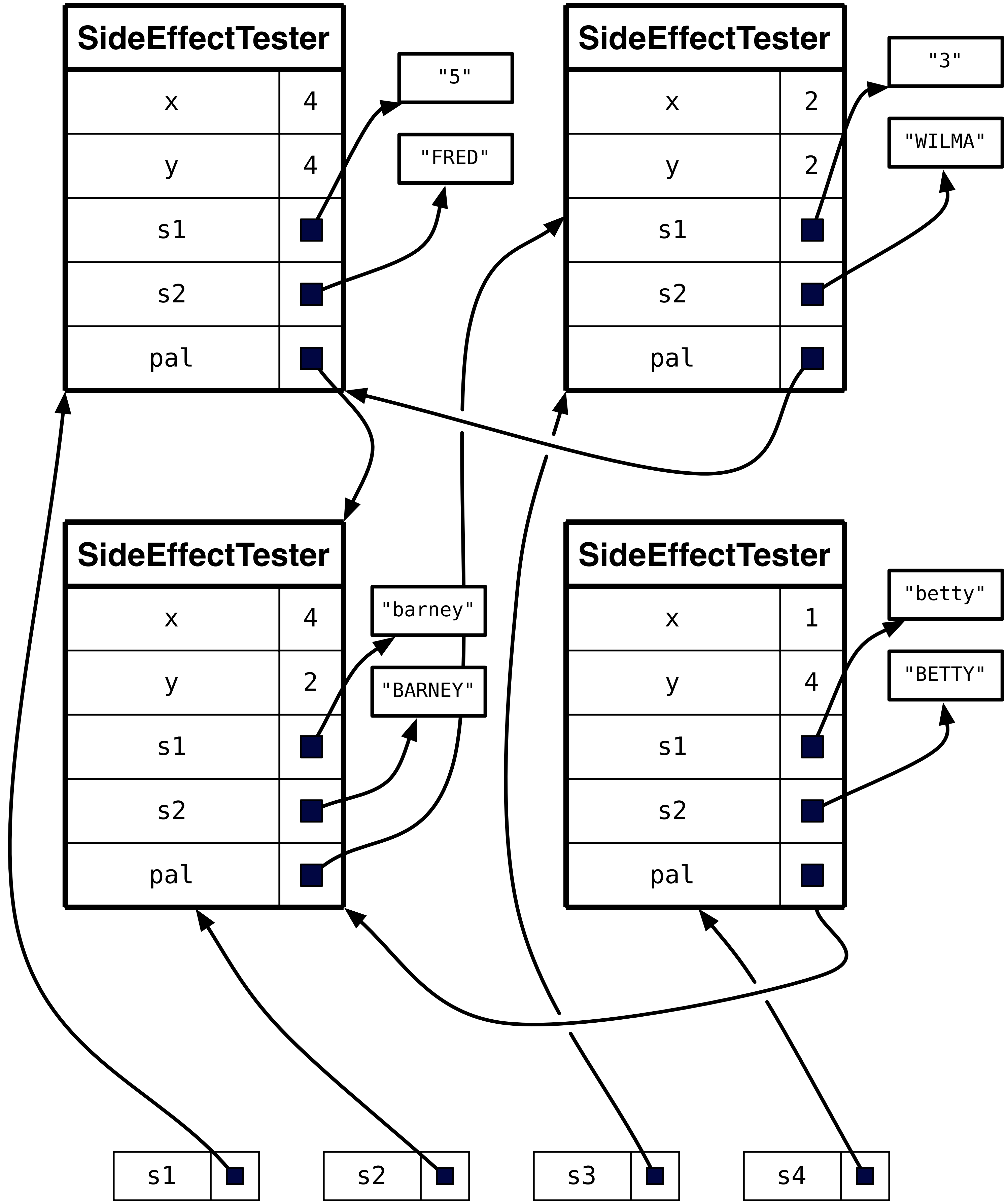
SideEffectTester solution



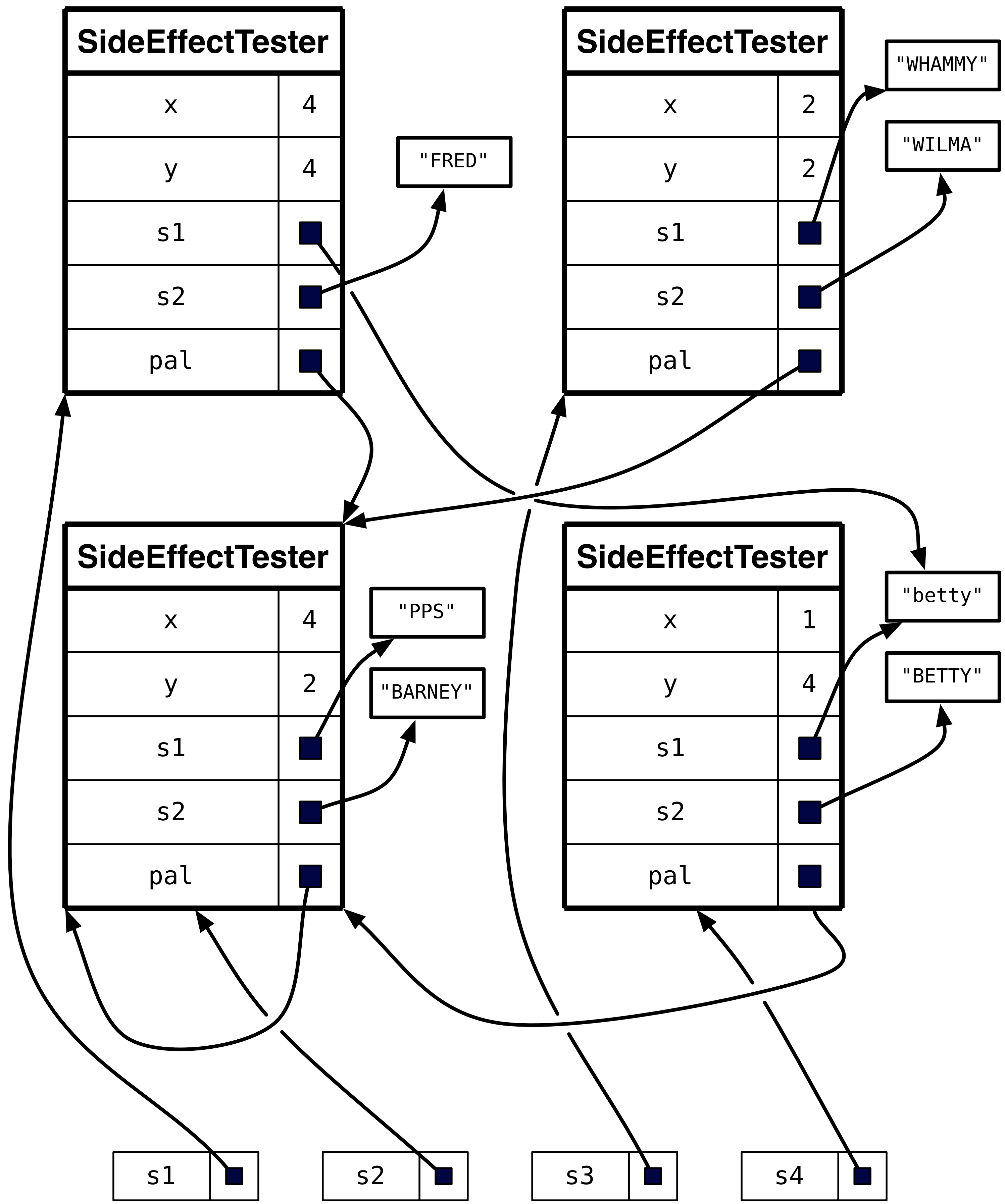
after constructors



after rubbish() calls



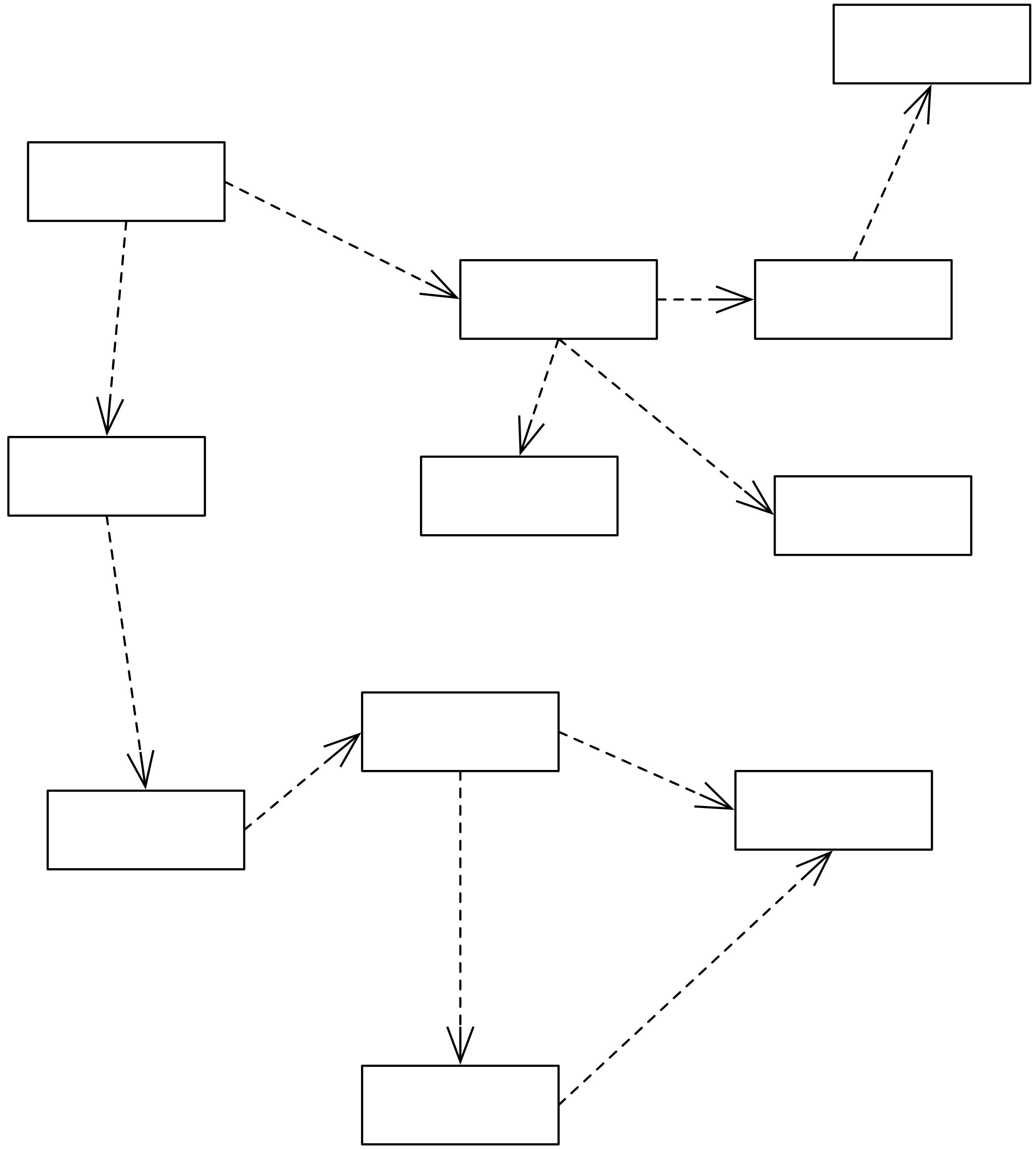
after mangle() calls

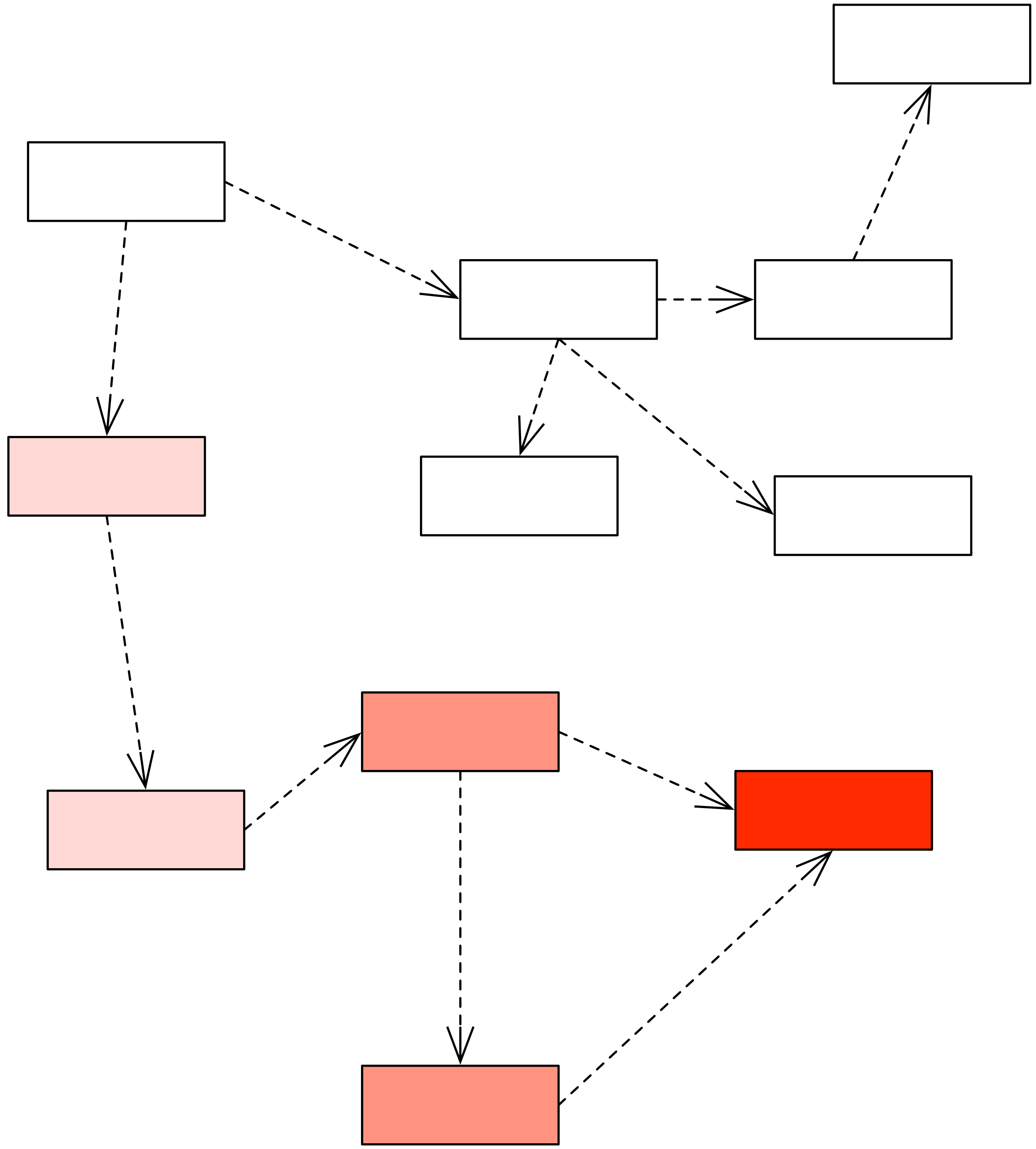


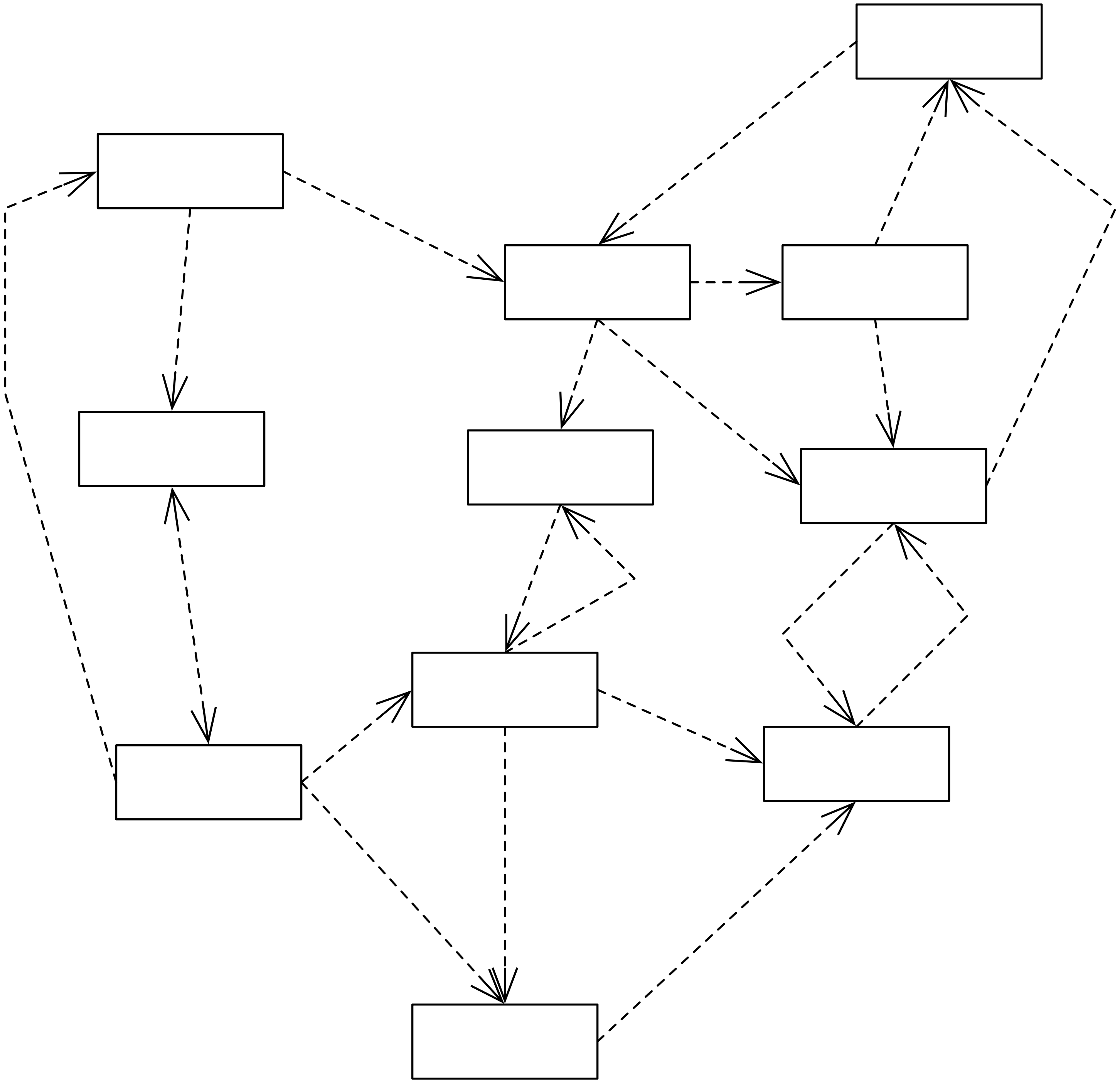
after kruftulate() calls

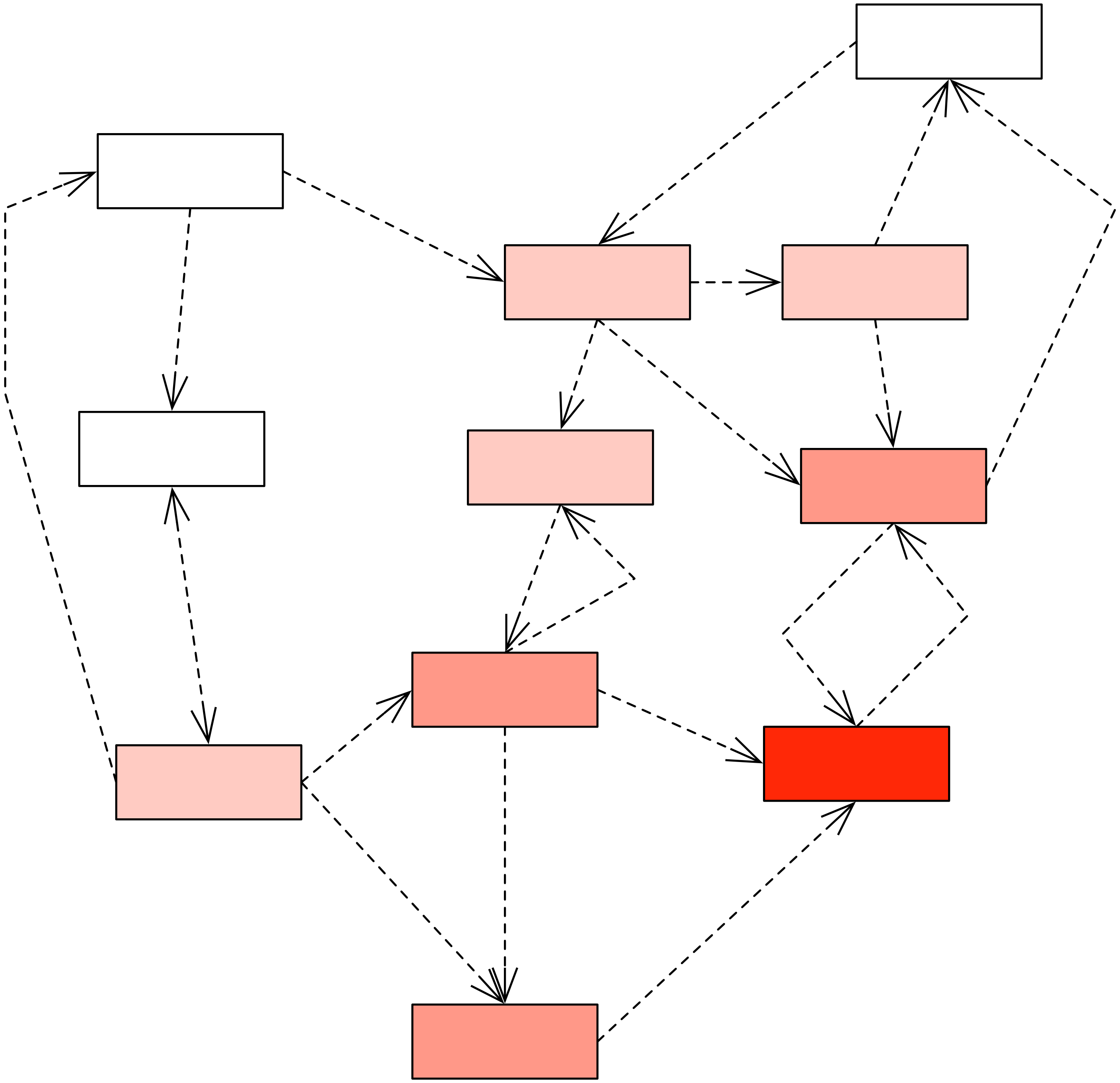
Two classes are *coupled* if they must know about one another.

Alternatively, a class is *coupled* to another if it depends on that class.









Avoid excessive and bidirectional coupling.

The vending machine

- Design the classes for a vending machine
  - Holds cans of soft drink, keeps track of inventory
  - Accepts fifty cents in currency
  - Activates product selector when paid

How do we evaluate a design?

The vending machine

- Design the classes for a vending machine
  - Holds cans of soft drink, keeps track of inventory
  - Accepts fifty cents in currency
  - Activates product selector when paid

OK, but what about....

- Debit cards?
- Different product types?
- Temperature-controlled discounts?
- Is our design general enough to handle these cases?

What makes for a good design?

Good designs enable future improvements.

Wrap-up

- Read Alistair Cockburn's "coffee machine problem" article
  - http://tinyurl.com/faq7r
  - Inspiration for vending machine problem
- Why doesn't Cockburn like UML? Is this an inherent limitation of UML as we've used it?

Any questions?

So, how do we come up with a good design?

...and meet changing requirements?

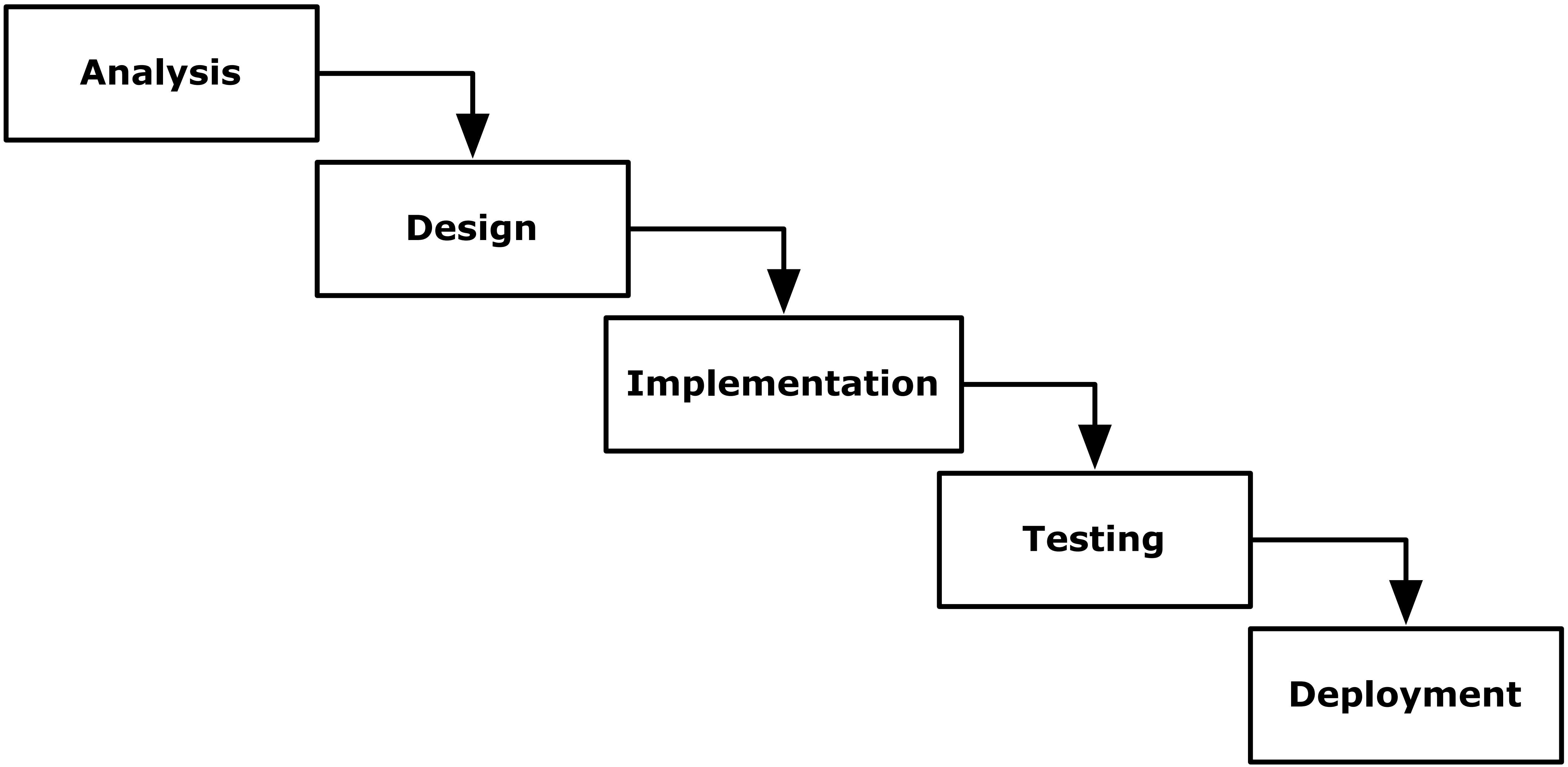
...and wind up producing a program that works?

The *software lifecycle* is the "big picture" of software development.

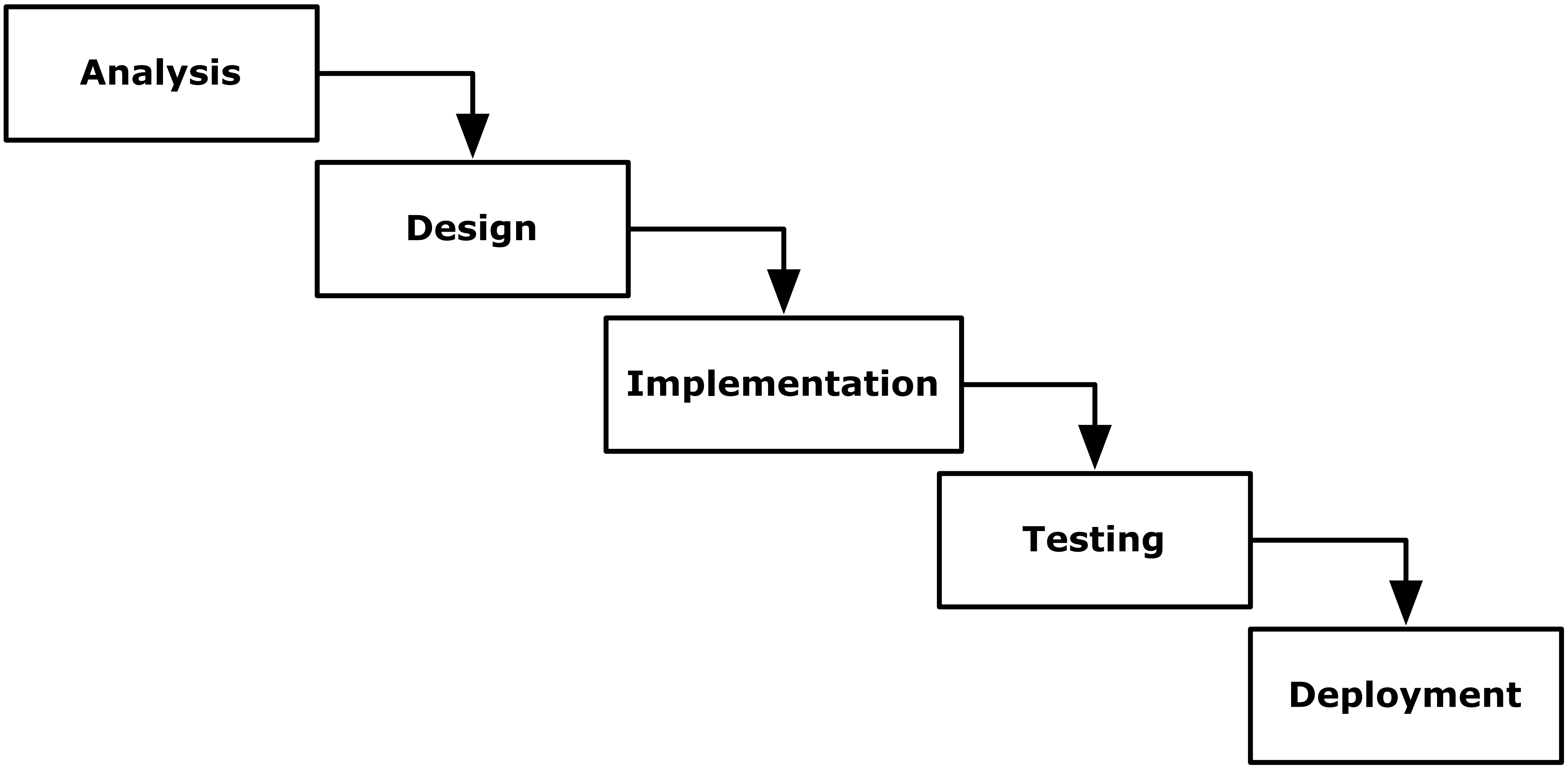
The software lifecycle

- How do we go from an idea to a design to a working program?
- How do we think about this process?

The “waterfall model” is an old way of looking at software.

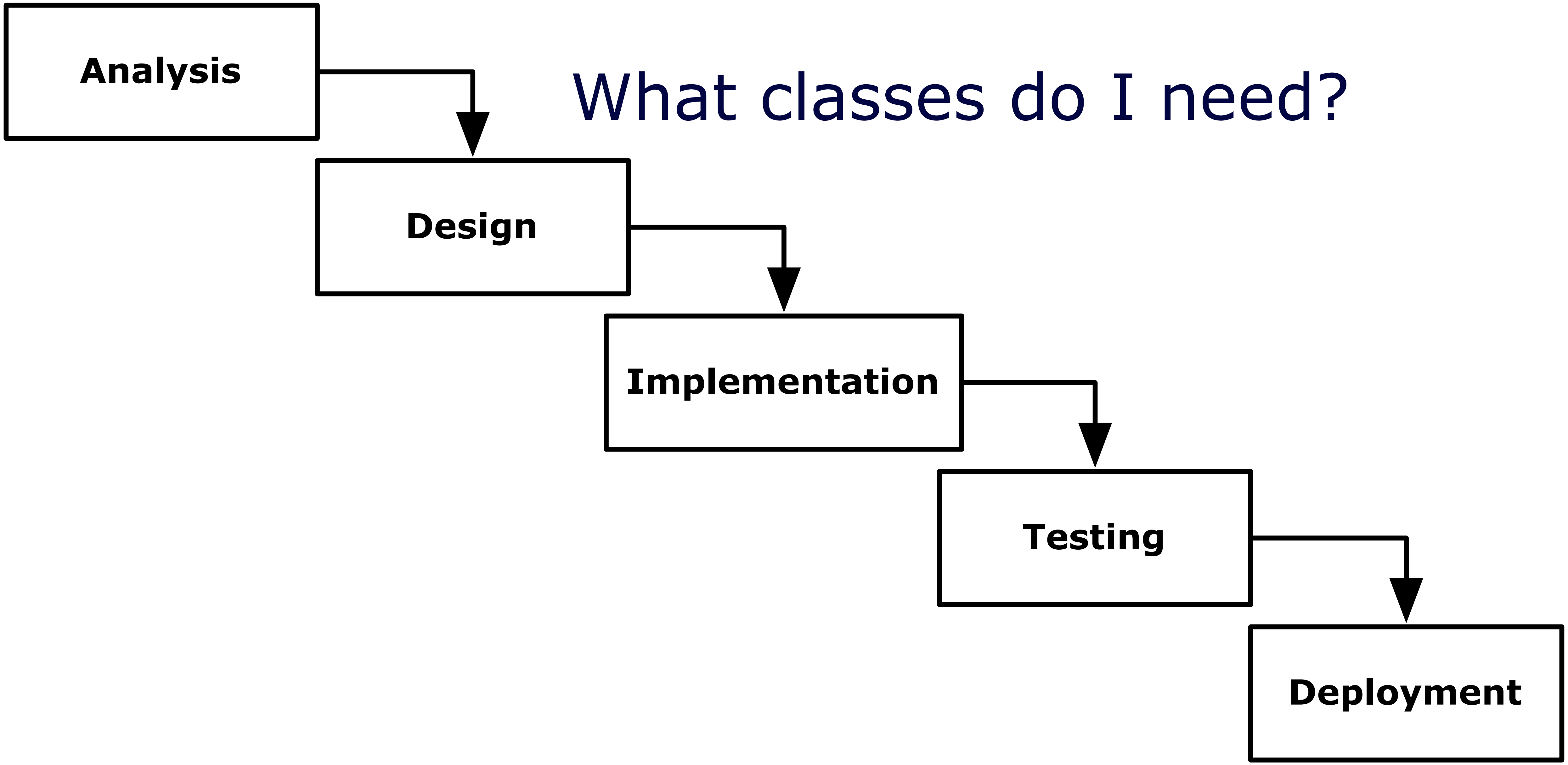

Analysis
Design
Implementation
Testing
Deployment

What are the requirements?



What are the requirements?

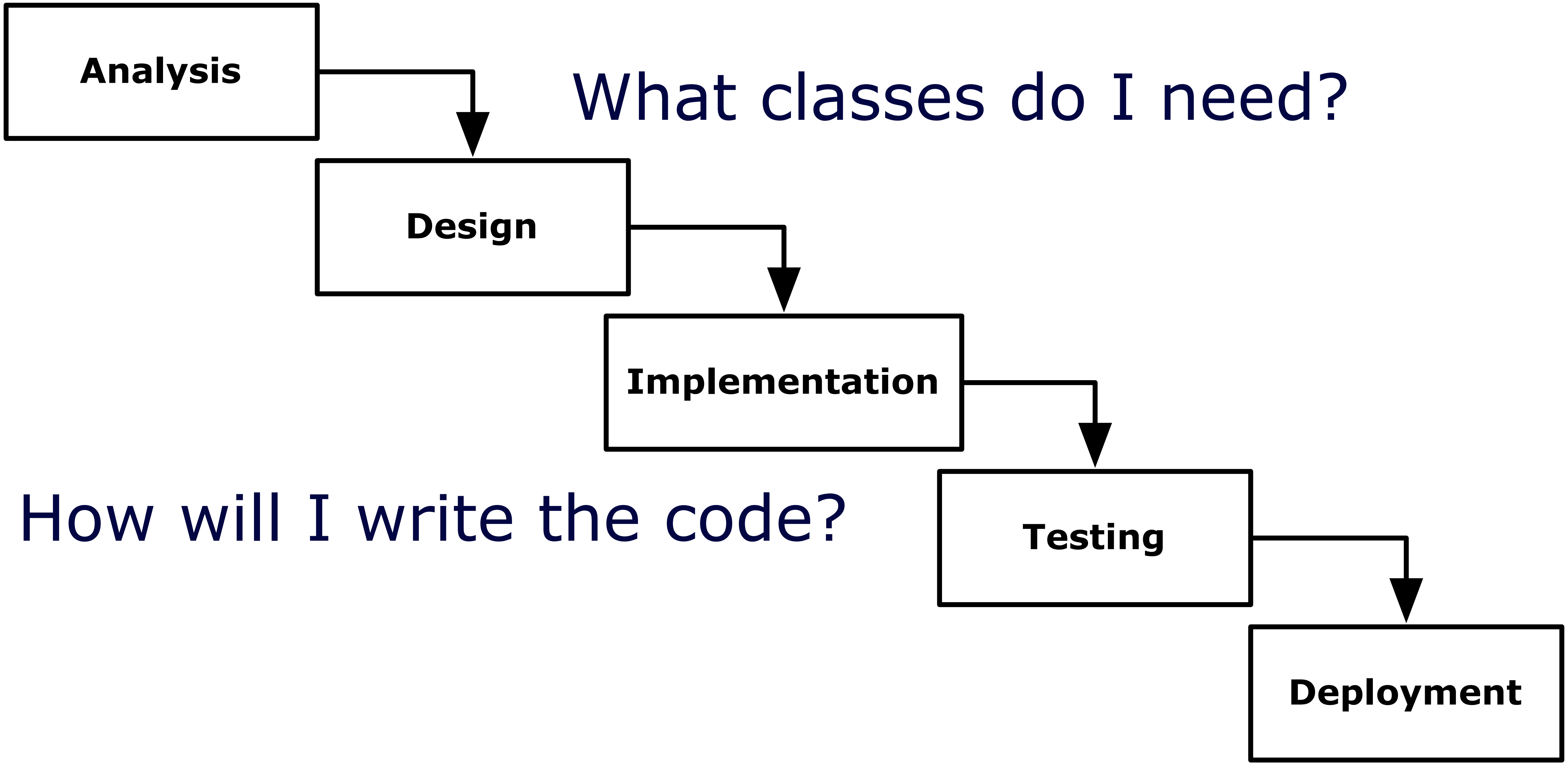
What classes do I need?

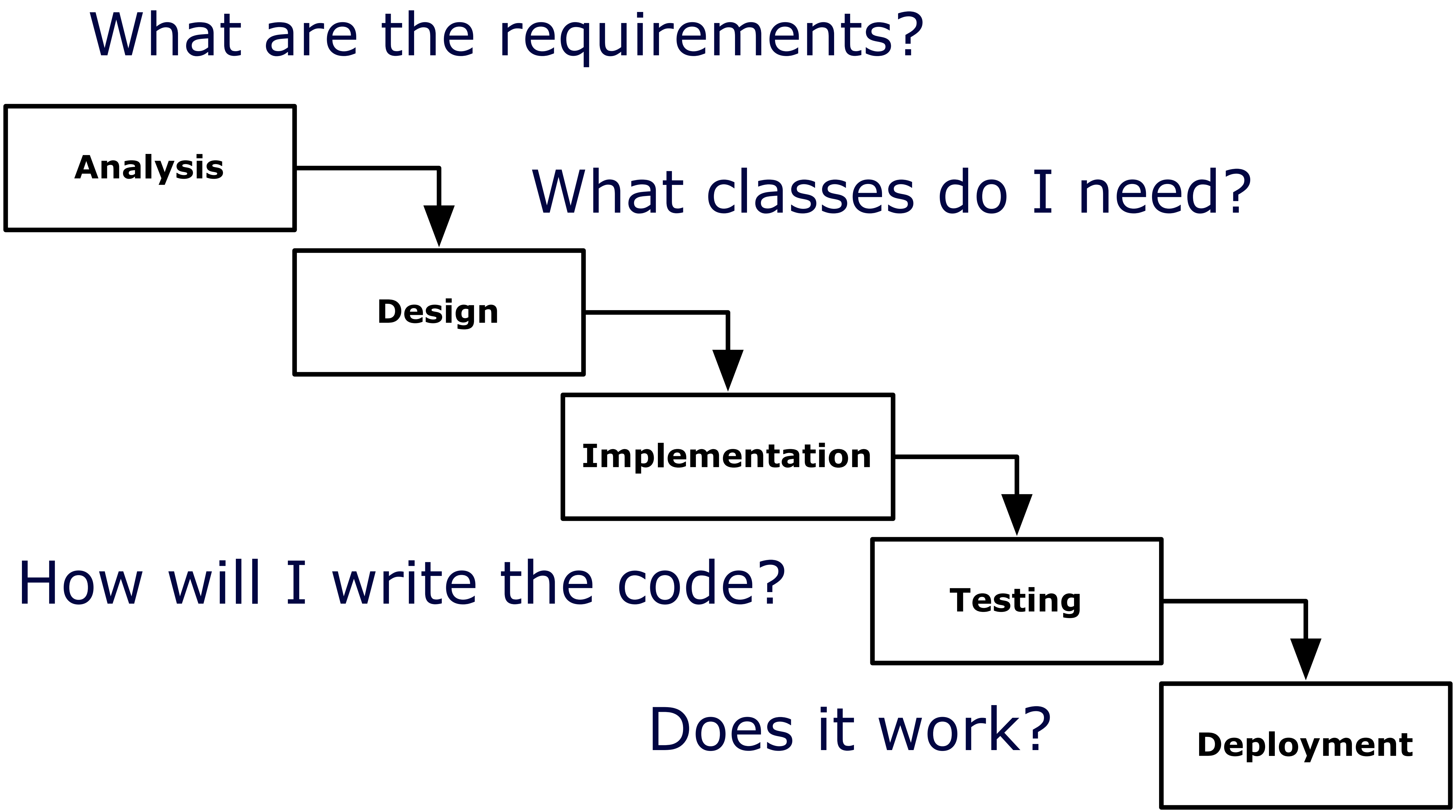


What are the requirements?

What classes do I need?

How will I write the code?




What are the requirements?
Analysis
What classes do I need?
Design
Implementation
How will I write the code?
Testing
Does it work?
Deployment

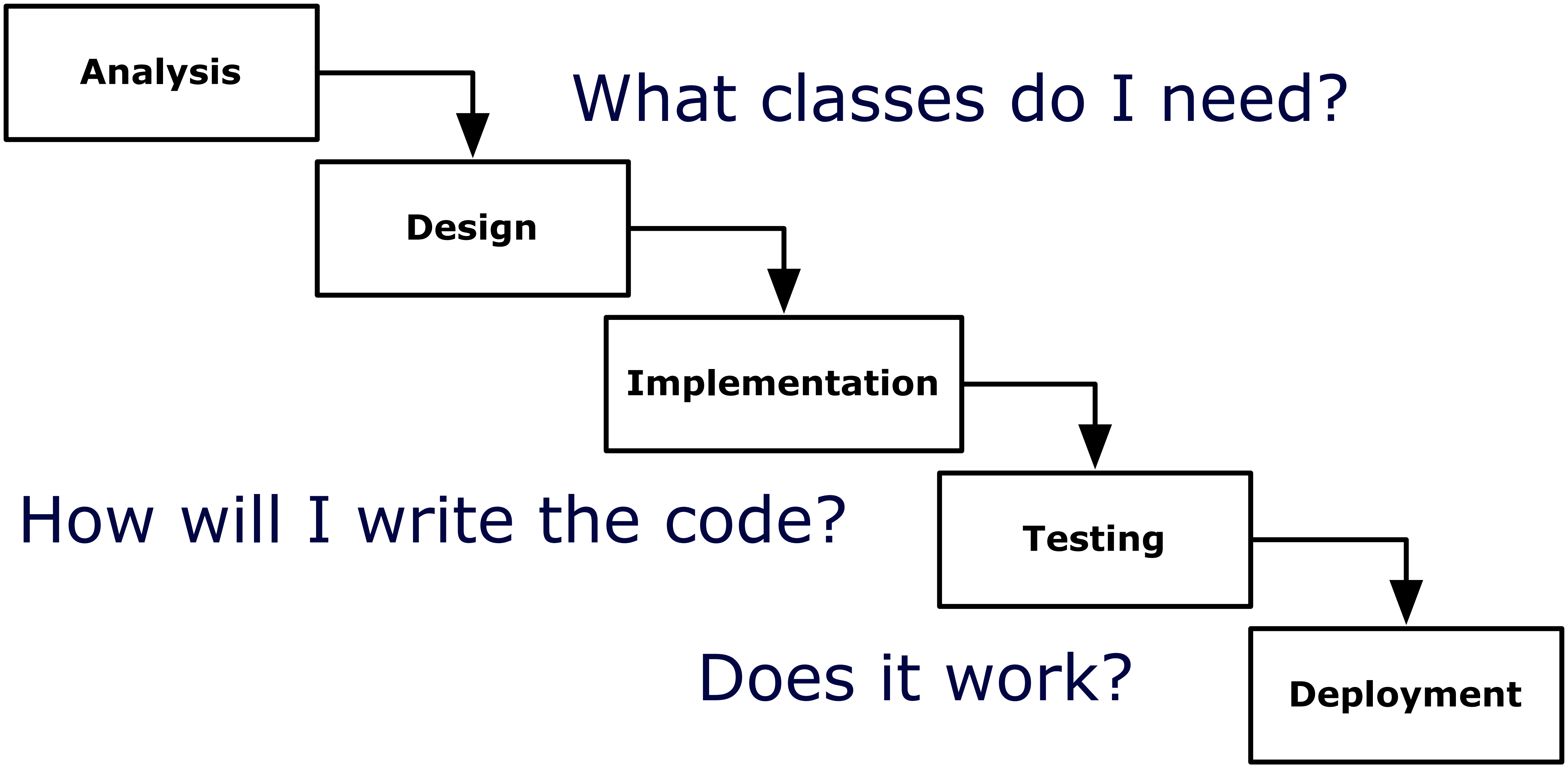
What are the requirements?

What classes do I need?

How will I write the code?

Does it work?

Done!

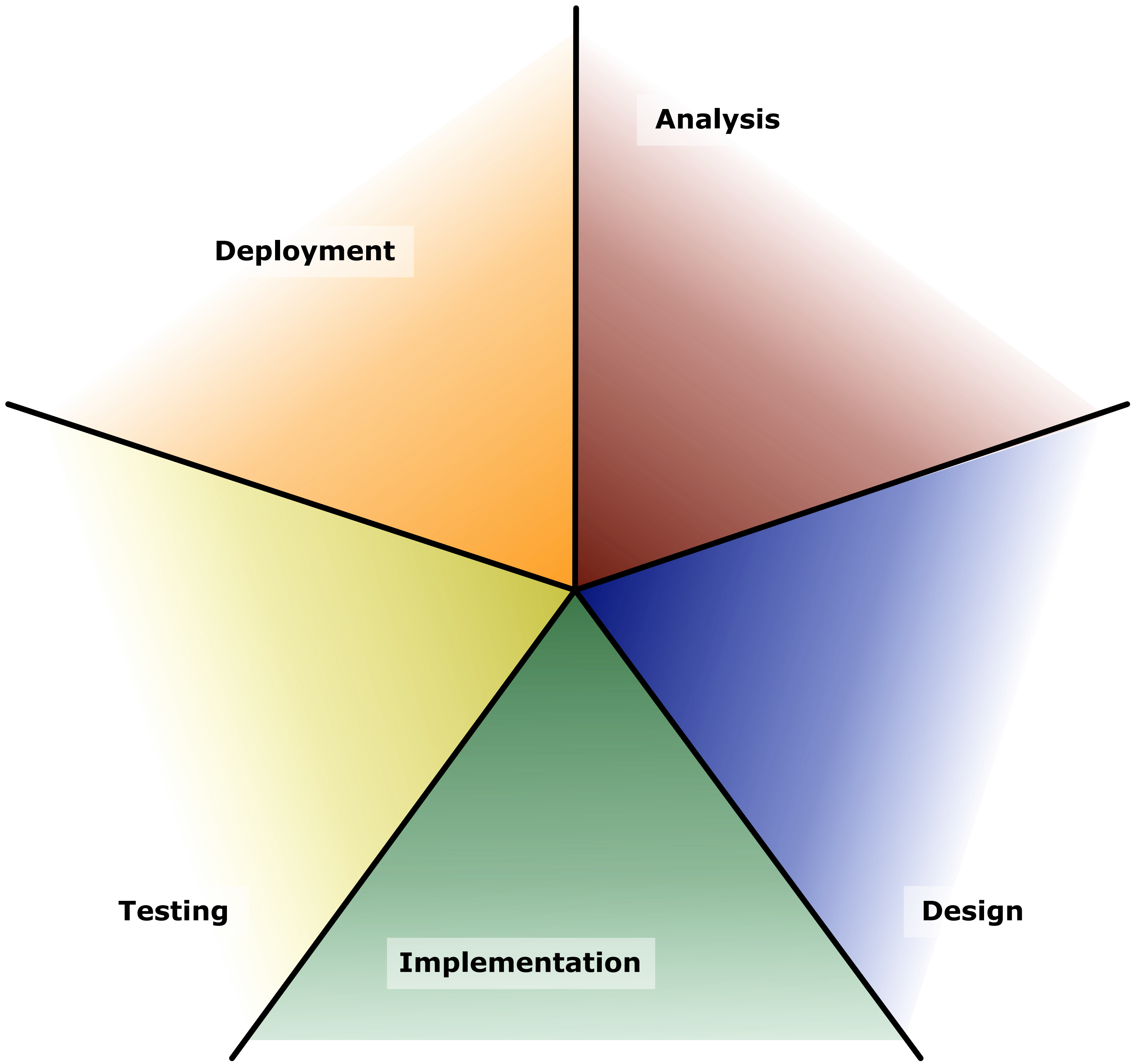


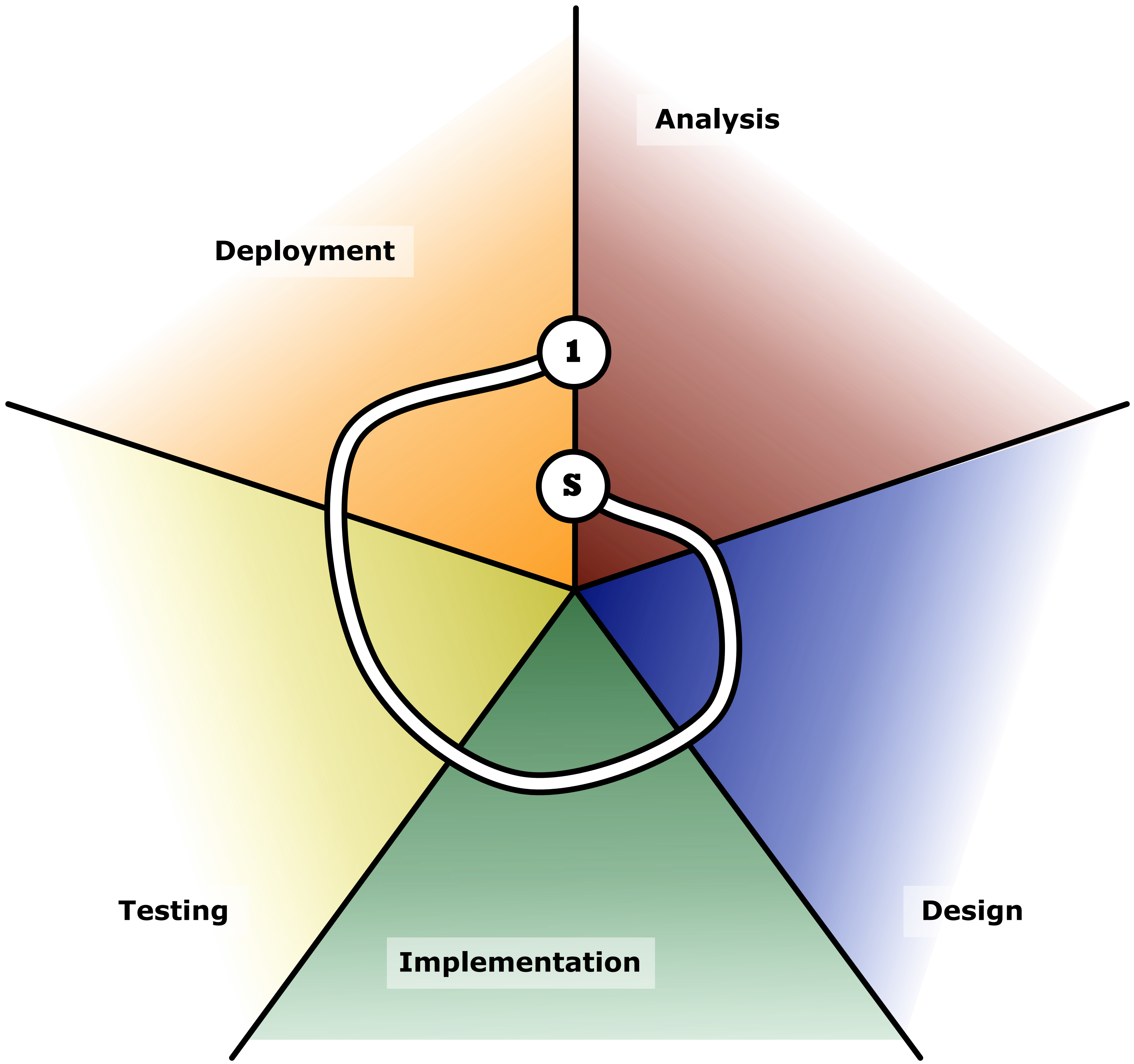
What's wrong with this picture?

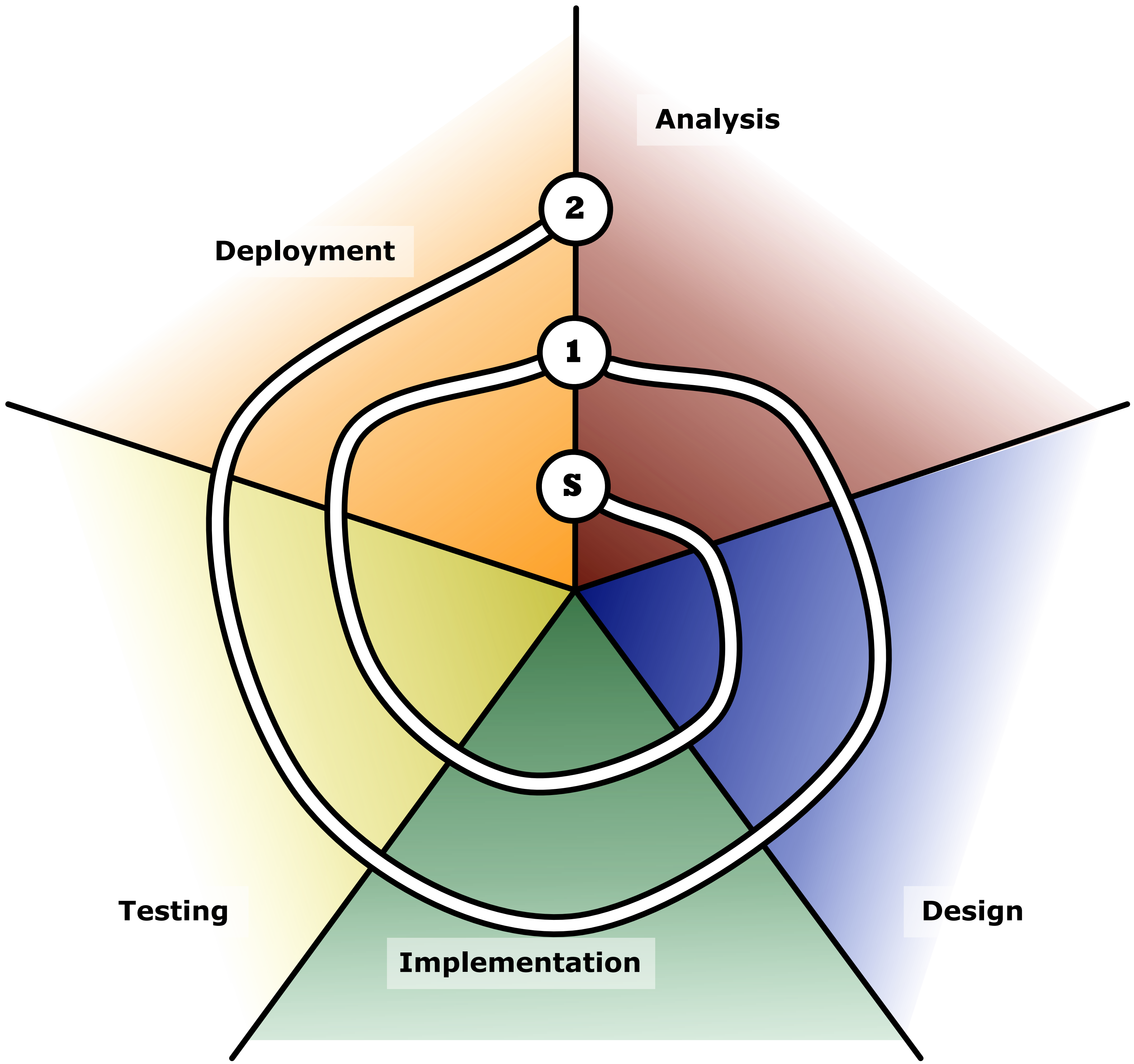
(Indeed, the waterfall model was initially proposed as a straw man.)

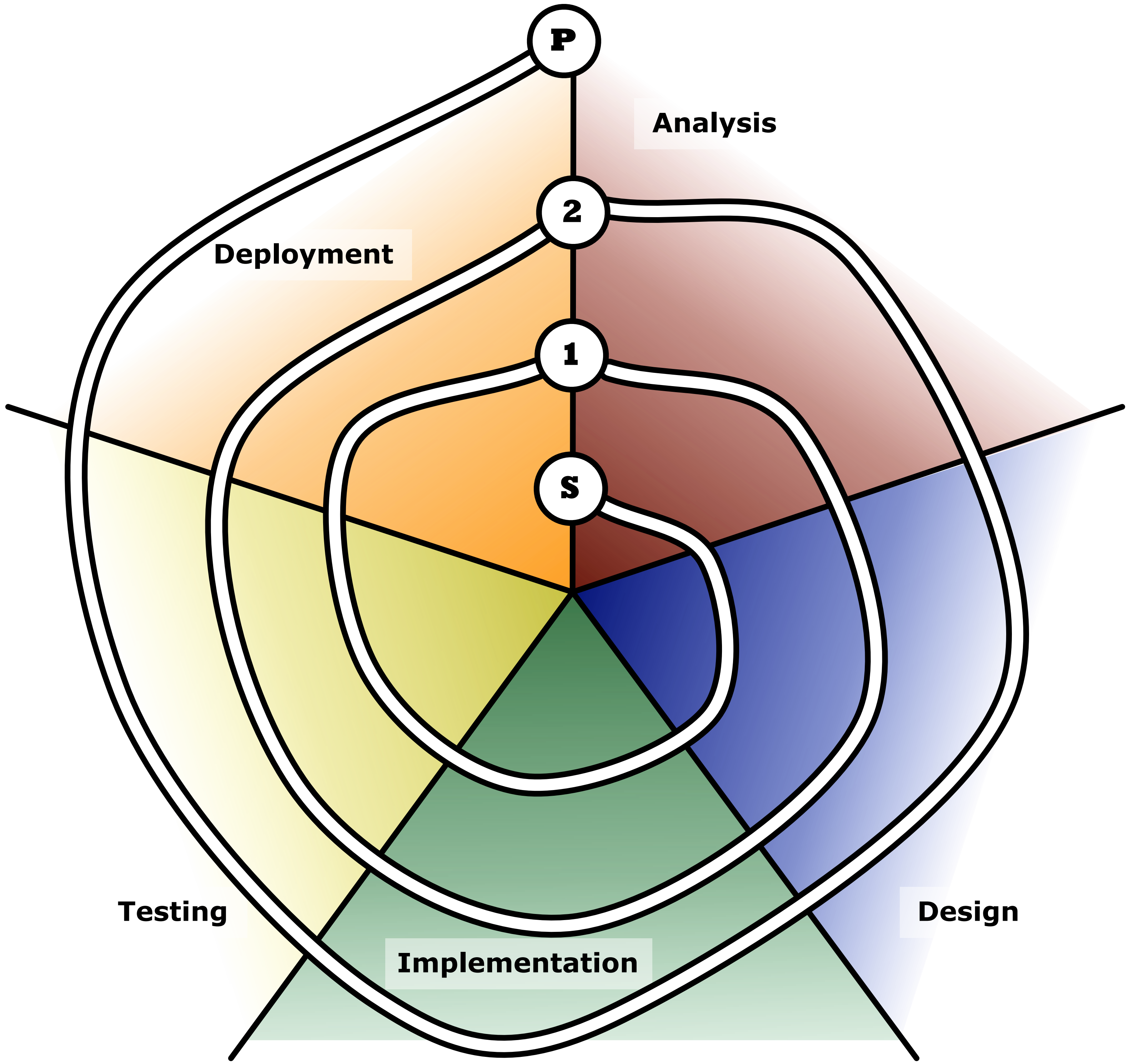
The *spiral model* applies the waterfall model iteratively, developing several prototypes.

(Hopefully, these eventually result in a finished product.)


Analysis
Deployment
Testing
Implementation
Design


Analysis
Deployment
1
S
Testing
Design
Implementation


Analysis
Deployment
2
1
S
Testing
Implementation
Design


P
Analysis
2
Deployment
1
S
Testing
Design
Implementation

What’s wrong with this picture?

Any questions?

Extreme programming seeks to solve the problems of traditional s/w dev.

“Extreme?”

Ideas

- Customers make business decisions.
  - Tell “user stories” about how they’d like to use the program.
- Programmers make technical decisions.
- Make simple plans; fix these when inconsistent with state of world.

XP ideas, cont'd

- Frequent releases
- Test-driven development
- Pair programming
- Refactoring
  - See book by Martin Fowler -- v. useful!

XP is not a panacea, but it is a collection of good ideas (indeed, some are great).