

# Beyond OOP: A Case Study

DConf 2018

Luís Marques  
[`<luis@luismarques.eu>`](mailto:<luis@luismarques.eu>)

# Status Update

- DHDL transpiler proof-of-concept working
  - Compiles plain circuits to D
  - Working on tighter D integration without requiring full D compiler reimplementation
- Original plan was to first write a digital logic simulator (Simbool), *then* do DHDL
  - I was confused about good design

# Beyond OOP

- I have no *a priori* preference for OOP or non-OOP
  - I just want to know the truth
  - I'm ready to change my mind
  - Constructive feedback is always welcome

# Beyond OOP

- Bad code and bad designs are not a personal failing
  - I've produced my own share of them
  - Programming is hard
  - Great software is more important than great code

# Beyond OOP

- This talk argues that OOP is wrong *in the general case*
  - “An aspirin cures everything” is wrong in the general case
  - It’s still the right treatment in specific cases
  - “Of course aspirin doesn’t cure everything. Just use the most appropriate tool for the job”.
  - But no one ever points out a broader theory that tries to explain what “medicine” best treats what “ailment”



Search Stories ▾ by Popularity ▾ for All time ▾

942 results (0.008 seconds) 

## Goodbye, **Object Oriented** Programming

377 points | ingve | 2 years ago | 340 comments | (<https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53>)

## Rob Pike on **Object Oriented** programming

231 points | crawshaw | 5 years ago | 169 comments | (<https://plus.google.com/101960720994009339267/posts/hoJdanhKwb>)

## What's wrong with **Object-Oriented** Programming and Functional Programming

214 points | roguelynn | 4 years ago | 142 comments | (<https://yinwang0.wordpress.com/2013/11/09/oop-fp/>)

## **Object oriented** programming with ANSI-C (1993) [pdf]

214 points | geospeck | 6 months ago | 57 comments | (<https://www.cs.rit.edu/~ats/books/ooc.pdf>)

## Dimple: An **object-oriented** API for business analytics powered by D3

150 points | based2 | 2 years ago | 31 comments | (<http://dimplejs.org/>)

## **Object Oriented** Programming is Inherently Harmful

149 points | idoco | 3 years ago | 190 comments | ([http://harmful.cat-v.org/software/OO\\_programming/](http://harmful.cat-v.org/software/OO_programming/))

## Rich Hickey's Keynote: A deconstruction of **object-oriented** time [pdf]

147 points | swannodette | 9 years ago | 43 comments | (<http://wiki.jvmlangsummit.com/images/a/ab/HickeyJVMSummit2009.pdf>)

## Golang **Object Oriented** Design

143 points | JanLaussmann | 5 years ago | 73 comments | (<http://nathany.com/good/>)

## Applying the Unix Philosophy to **Object-Oriented** Design

125 points | sudonim | 5 years ago | 58 comments | (<http://blog.codeclimate.com/blog/2012/11/28/your-objects-the-unix-way/>)

## Rooby: a Ruby-like **object oriented** language written in Go

122 points | type0 | a year ago | 66 comments | (<https://github.com/rooby-lang/rooby>)

## Alan Kay on the Meaning of "**Object-Oriented** Programming" (2003)



Search

Stories ▾

by

Popularity ▾

for

All time ▾

878 results (0.002 seconds)



## Don't distract new programmers with **OOP**

248 points | angrycoder | 7 years ago | 154 comments | (<http://prog21.dadgum.com/93.html>)

## Alan Kay on the misunderstanding of **OOP** (1998)

240 points | mmphosis | 2 years ago | 206 comments | (<http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>)

## Golang concepts from an **OOP** point of view

200 points | markkit | 2 years ago | 46 comments | (<https://github.com/luciottato/golang-notes/blob/master/OOP.md>)

## **OOP** Isn't a Fundamental Particle of Computing

194 points | joeyespo | 5 years ago | 158 comments | (<http://prog21.dadgum.com/156.html>)

## Java 9 with GPU processing, Java 10 will be all-**OOP** without primitives

188 points | Mitt | 6 years ago | 135 comments | (<http://www.javaworld.com/javaworld/jw-03-2012/120315-oracle-s-java-roadmap.html>)

## Don't Distract New Programmers with **OOP**

186 points | ColinWright | 4 years ago | 220 comments | (<http://prog21.dadgum.com/93.html?HN2>)

## 99 Bottles of **OOP**

164 points | bdcravens | 2 years ago | 71 comments | (<http://www.sandimetz.com/99bottles>)

## Understanding Javascript **OOP**

150 points | old\_sound | 6 years ago | 19 comments | (<http://killdream.github.com/blog/2011/10/understanding-javascript-oop/>)

## **OOP** practiced backwards is "POO"

140 points | raganwald | 7 years ago | 90 comments | (<http://github.com/raganwald/homoiconic/blob/master/2010/12/oop.md#readme>)

## Alan Kay on the misunderstanding of **OOP**

119 points | chc | 8 years ago | 30 comments | (<http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>)

## **OOP** no longer mandatory in CMU Computer Science Curriculum

# Typical Arguments

- “The problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.” – Joe Armstrong
  - Funny, but is it true?
  - This is an argument about dependencies. If true:
    - *Incidental* fact of current practices/technology?
    - Necessary consequence of OOP?

# Typical Arguments

- “Sure OOP might not be appropriate for some things, but you really want it for GUIs”
  - What about React?
  - Do you need all of the properties of OOP for GUI programming?
  - If not, which subset?
  - Can other paradigms have that subset?

# Typical Arguments

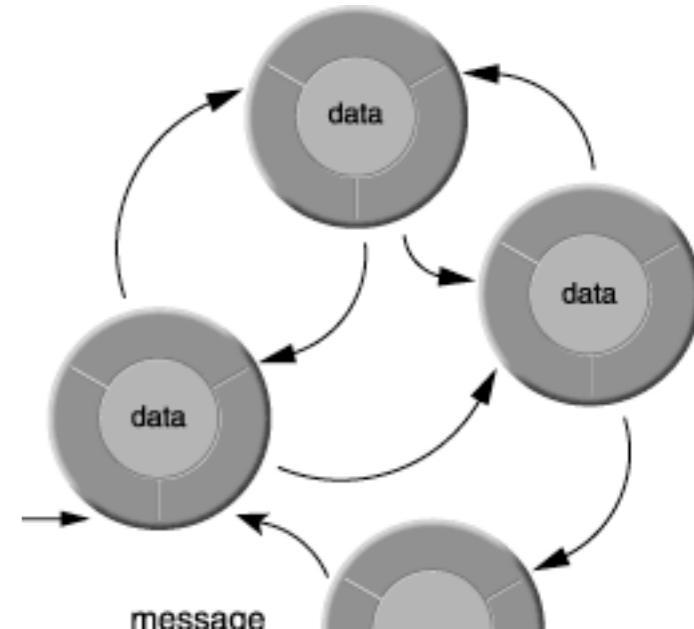
- “Sure OOP might not be worth it for small projects. But for large projects you really need OOP to tackle the complexity”
  - Why can’t you decompose your large project into smaller parts and then use other paradigms?
  - What *exactly* is about OOP that other paradigms lack to be able to tackle that complexity?
  - Exactly what size is that? How do you *know* it?

# Moving the Goalposts?

- 1998:
  - Alice: “Visual Basic 6 is great. Now it’s even object-oriented.”
  - Bob: “Nah, VB 6 doesn’t have inheritance. For OOP you need polymorphism, inheritance and encapsulation.”
- 2018:
  - Alice: “OOP sucks. Inheritance causes lots of problems.”
  - Bob: “Ah yes, everybody knows you should prefer composition over inheritance. It’s not my fault if you don’t practice OOP correctly!”

# OOP

- Some things are arguably not fundamental to OOP.  
Example: classes are just a *mechanism* for defining related objects. Thus the same with inheritance.
- Probably the most fundamental:
  - Messaging
  - Encapsulation
  - What's the problem with this?



# Object-Oriented Design

- There's nothing wrong with objects *per se*
  - An object is just a concept, like an integer or red
  - But there is an expectation on *how* we use objects to solve programming problems
  - What is an object, what is encapsulated, etc.
  - That's what this talk criticises



# Business Value

- What is our fire? (goal, \$)
  - Features
  - Performance
  - Reliability
  - Security
  - ...

# (Business Value)'

- Goal mechanisms:
  - Encapsulation
  - Design by contract
  - Tests
  - Type checking

# **((Business Value)')'**

- Goal mechanisms, mechanisms:
  - Member functions
  - Assertions
  - Subtyping
  - Continuous integration server
- Mechanism → Mechanism → Mechanism → Goal

# Business Value?

- Scott Meyers on encapsulation:
  - Member functions → encapsulation → flexibility and robustness → \$
  - *“I've been battling programmers who've taken to heart the lesson that being object-oriented means putting functions inside the classes containing the data on which the functions operate. After all, they tell me, that's what encapsulation is all about.”* [1]

# Business Value?

- Scott Meyers on encapsulation:
  - ~~Member functions~~ → encapsulation → flexibility and robustness → \$
  - *"I've been battling programmers who've taken to heart the lesson that being object-oriented means putting functions inside the classes containing the data on which the functions operate. After all, they tell me, that's what encapsulation is all about."* [1]
  - Non-member functions improve encapsulation

# Business Value?

- Scott Meyers on encapsulation:
  - Non-member functions → encapsulation → flexibility and robustness → \$
- Other alternatives?
  - Non-member functions → Parnas abstract interface → flexibility and robustness → \$
  - Pure functions → functional programming → flexibility and robustness → \$

# Software Engineering

- Superficial goals and objections:
  - “That’s not object-oriented”
  - “You aren’t doing test-driven design”
  - “You are not following the rational process”
  - “We must use a dependency injection framework”

# Approach Overview

- Identify (possible) contradiction
- Understand contradiction
- Solve contradiction?
- Put it in practice

# Reductio ad absurdum

- Bad code comments:

```
void increaseScore() {  
    goals++; // increase the number of goals  
}
```

- Better code comments:

```
void increaseScore() {  
    goals++; // each goal scores one point  
}
```

# Reductio ad absurdum

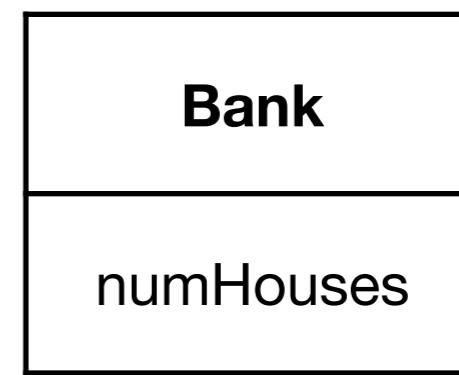
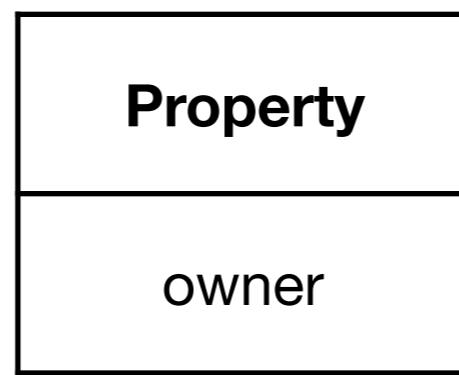
- Is this a contradiction?

```
/**  
Swaps `lhs` and `rhs`. (...)  
  
Params:  
    lhs = Data to be swapped with `rhs`.  
    rhs = Data to be swapped with `lhs`.  
*/  
void swap(T)(T lhs, T rhs)
```

# Identify Contradiction

- Sillion Valley D Meetup (2016-01-28) argument [2]
  - Rich Domain Model (RDM)
    - OOP approach
  - Anemic Domain Model (ADM)
    - Procedural / functional approach

# RDM Monopoly



# RDM Monopoly

```
class Player
{
    Money cash;
    Property[] ownedProperties;
    // (...)

    bool canBuyProperty(Property property)
    {
        return cash >= property.printedPrice;
    }

    void buyProperty(Property property)
    {
        property.owner = this;
        cash -= property.printedPrice;
    }
}

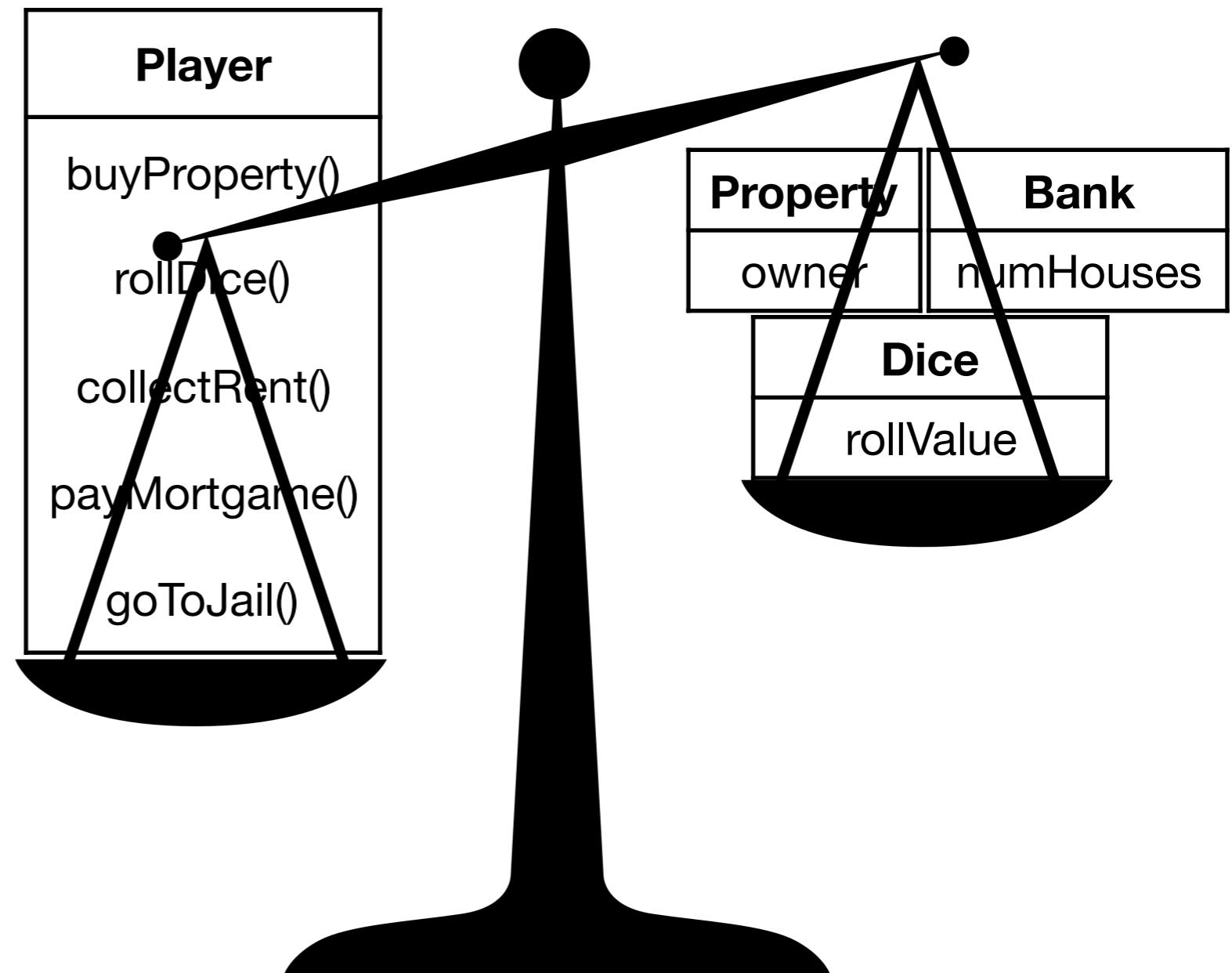
class Property
{
    string name;
    Money printedPrice;
    Player owner;
    // ...
}
```

# Object Responsibilities

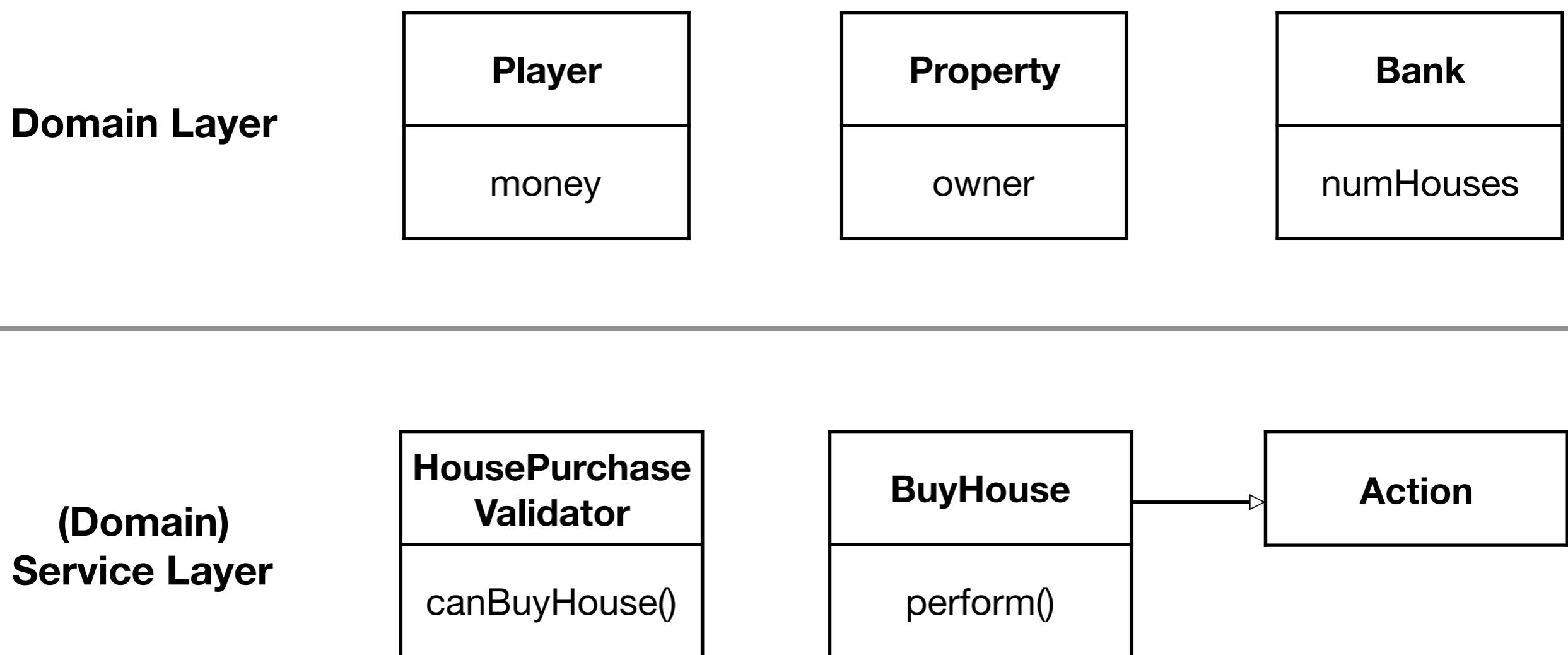
- If a player owns Baltic Avenue, can the player add a house to it?
  - Can she afford it?
  - Is there a house in the bank?
  - Is it either the player's turn or between turns?
  - Does the property already have four houses?
  - Is Baltic Avenue mortgaged?
  - What if Mediterranean Avenue (same group) is mortgaged?
  - What if Baltic Avenue has one house but Mediterranean Avenue has none?

# RDM Monopoly

- Fat objects
- God objects
- No single responsibility



# ADM Monopoly



# ADM Monopoly

```
class OfficialRulesHousePurchaseValidator : HousePurchaseValidator
{
    Game game;
    Player purchaser;
    Property property;

    bool canBuyHouse()
    {
        return game.currentPlayer == purchaser &&
            player.cash >= property.houseCost &&
            (...);
    }
}

class BuyHouse : Action
{
    Property property;
    HousePurchaseValidator validator; // we depend on the abstract validator interface

    override void perform()
    {
        enforce(validator.canBuyHouse);
        (...)
    }
}
```

# Meetup Conclusions

- The anemic approach had advantages:
  - Better separation of concerns (SRP)
  - Increased reusability
  - Easier to test

# Meetup Conclusions

- More detailed description:
  - Core Dump podcast, episode 1 [3]
  - <<http://www.coredump.xyz/1>>



# Identify Contradiction

- People want and believe they can get both OOP (RDM) and SRP
- To get SRP we had to go for fake OOP (ADM)
- Therefore  $\neg ( \text{SRP} \wedge \text{OOP} )$ 
  - Goes against standard assumptions. Contradiction.
- Also, OOP was supposed to bring reusability
  - In that case it was the not-quite-OOP solution that did

# Identify Contradiction

- Where's the contradiction?
  - On the surface: RDM vs SRP
  - But... principle of explosion

# Identify Contradiction

1.  $P \wedge \neg P$   
assumption
2.  $P$   
from (1) by conjunction elimination
3.  $\neg P$   
from (1) by conjunction elimination
4.  $P \vee Q$   
from (2) by disjunction introduction
5.  $Q$   
from (3) and (4) by disjunctive syllogism
6.  $(P \wedge \neg P) \rightarrow Q$   
from (5) by conditional proof (discharging assumption 1)

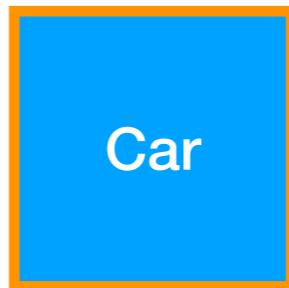
# Identify Contradiction

- $P \wedge \neg P \rightarrow Q$ 
  - roses are red  $\wedge$  roses are not red  $\rightarrow$  ham is tasty
- $\dots \rightarrow \neg (\text{SRP} \wedge \text{RDM}) \rightarrow \neg \text{RDM}$ 
  - Maybe the problem is upstream
  - Maybe it's not related to either SRP or RDMs
- What's the fundamental problem with OOP?

# OOP Problem

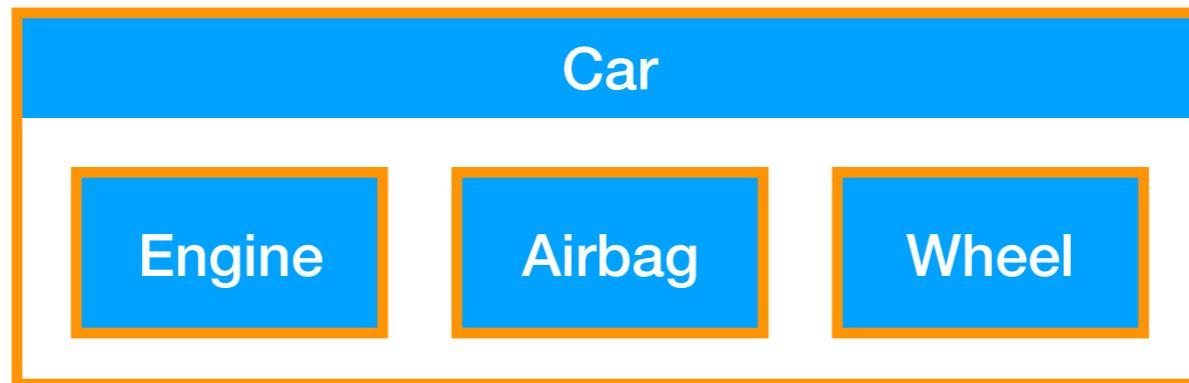
- The solution has to look like the problem
  - The entities always define the abstraction frontier

# Abstraction Frontier



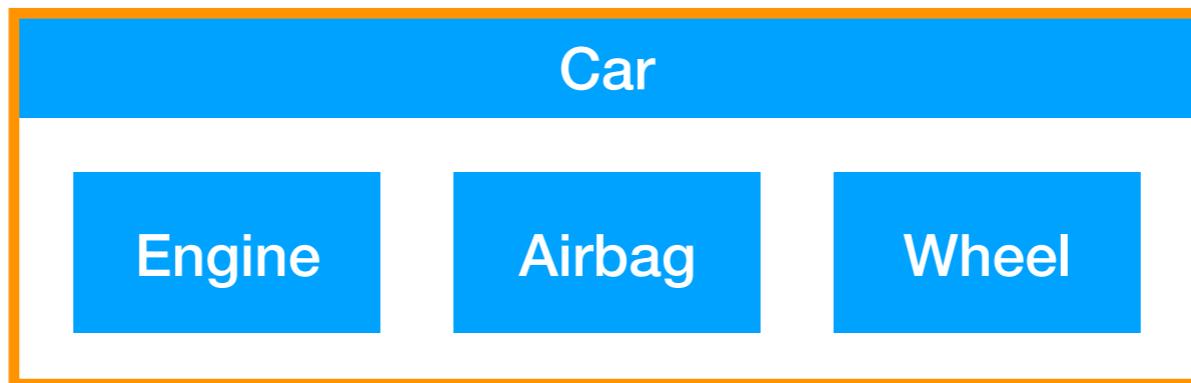
- The car OOP abstraction frontier
  - Things outside the car object know nothing about the internals of the car
  - Things inside the car object know everything about the internals

# Abstraction Frontier



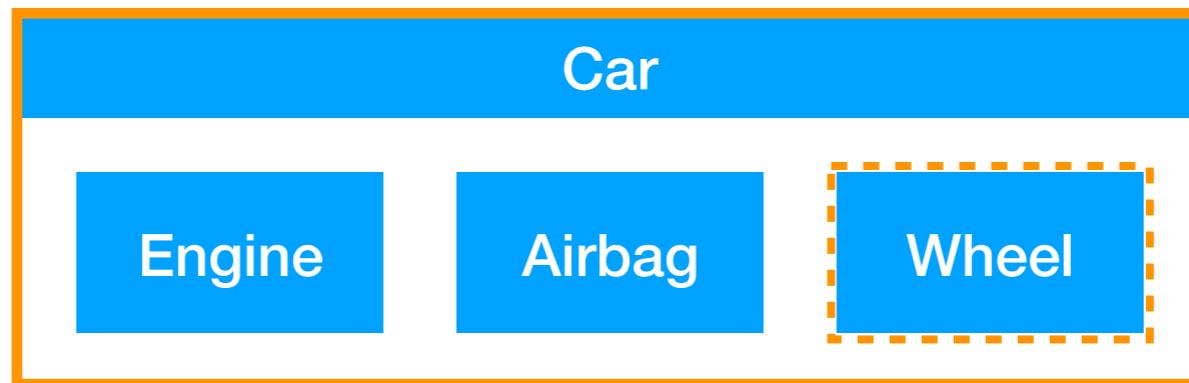
- Object composition is OOP

# Abstraction Frontier



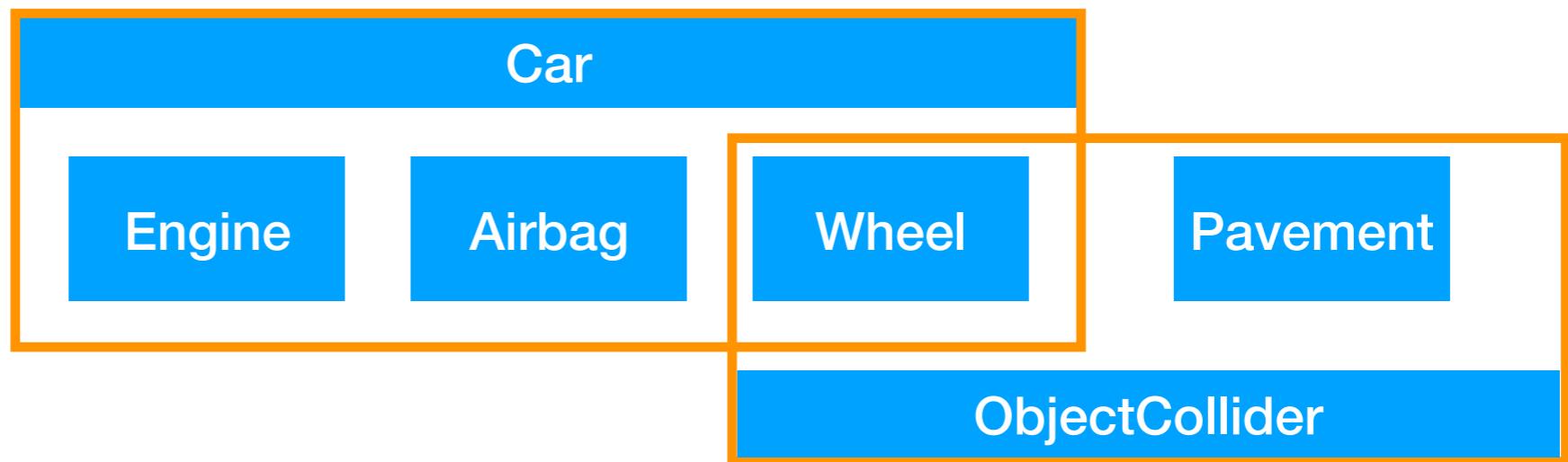
- Object composition is OOP
- Unencapsulated object aggregation is not

# Abstraction Frontier



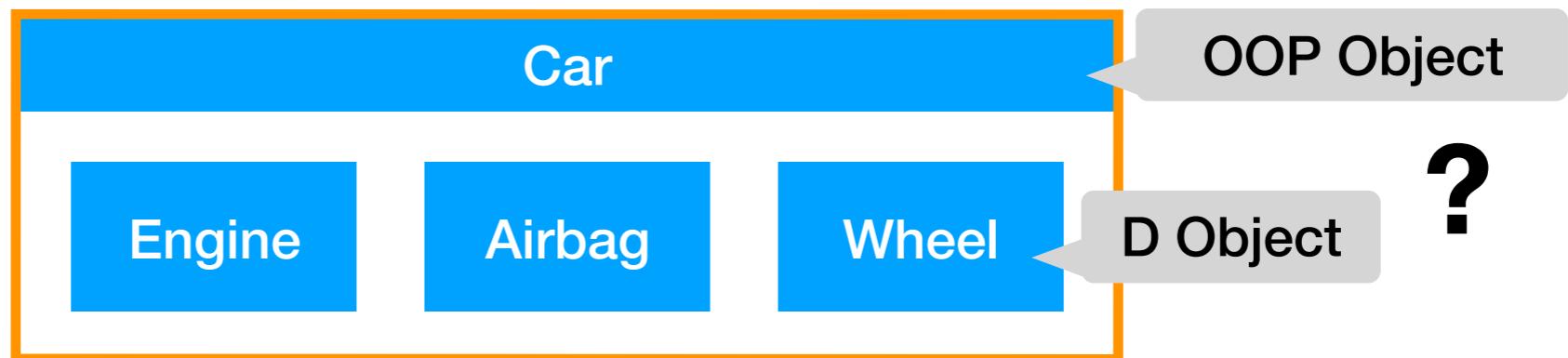
- Object composition is OOP
- Unencapsulated object aggregation is not

# Abstraction Frontier



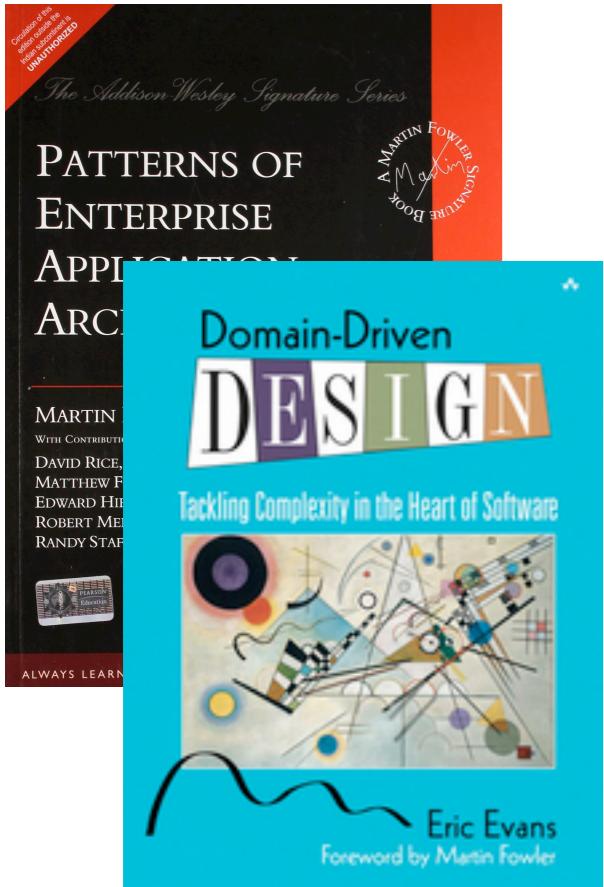
- Object composition is OOP
- Unencapsulated object aggregation is not
- Neither is shared encapsulation

# Abstraction Frontier

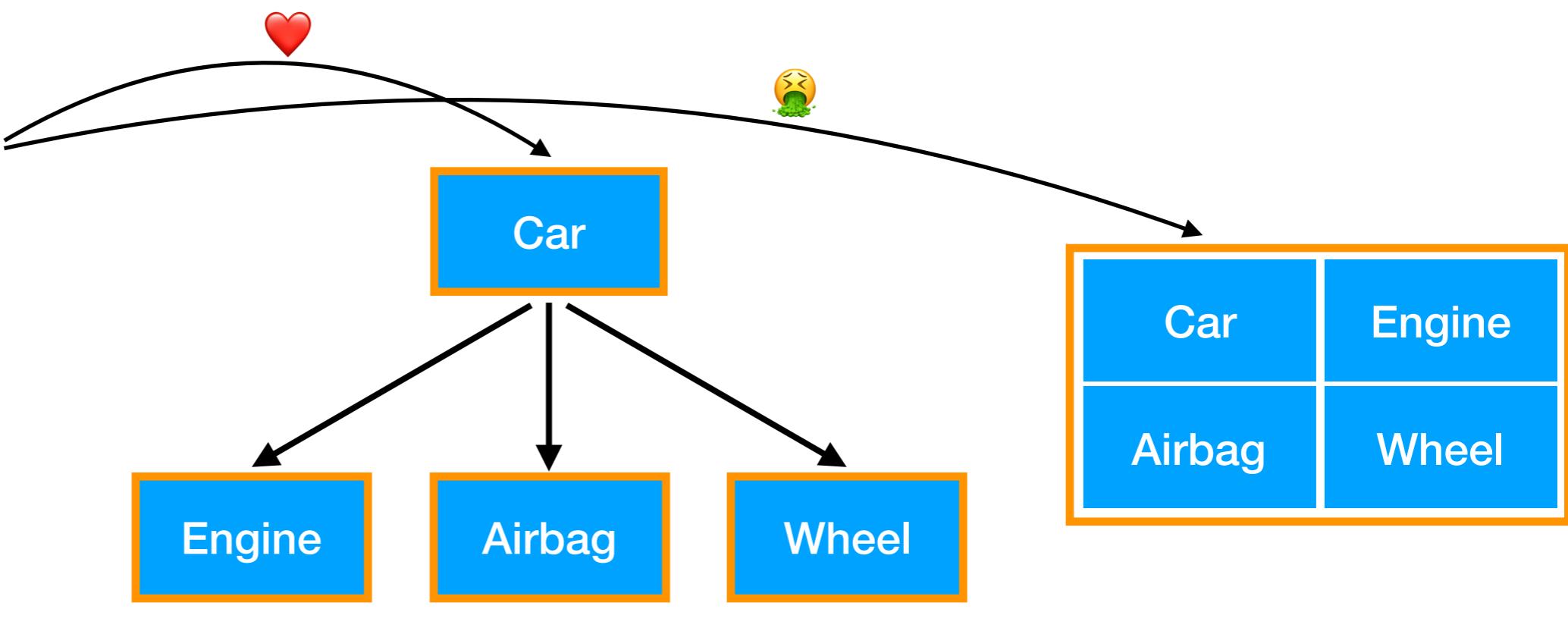


- Object composition is OOP
- Unencapsulated object aggregation is not
- Neither is shared encapsulation

# Abstraction Frontier



OOP is what OOP practitioners do



# Abstraction Frontier

```
void foo()
{
    while(...) {
        if(...)
        goto xxx
        return;
    }
}
```



Algorithm

# Abstraction Frontier

```
void foo()  
{  
    while(...) {  
        if(...)  
        goto xxx  
        return;  
    }  
}
```



```
void foo()  
{  
    range  
    .algorithm1  
    .algorithm2  
    .algorithm3  
    .consumer;  
}
```



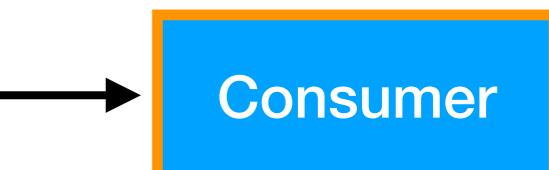
# Abstraction Frontier

```
void foo()  
{  
    while(...) {  
        if(...)  
        goto xxx  
        return;  
    }  
}
```



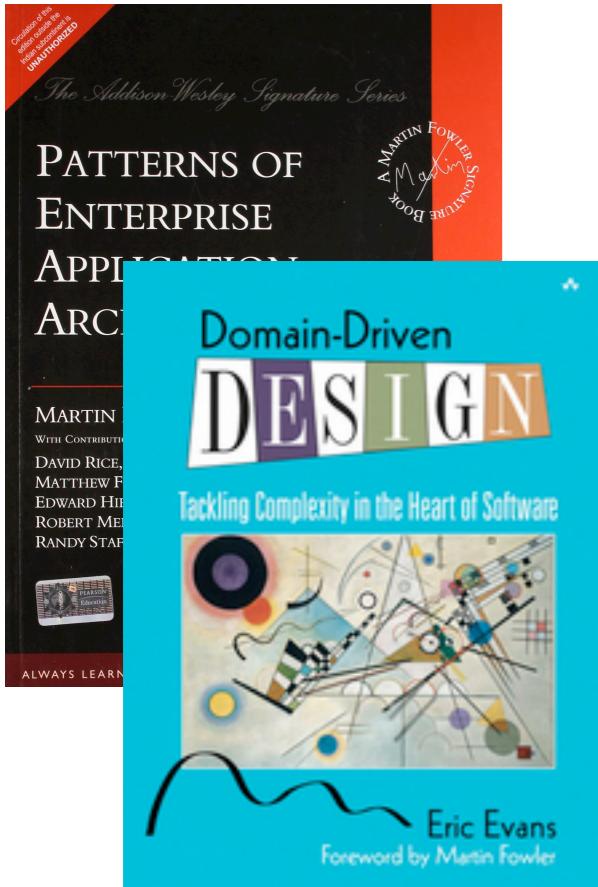
```
void foo()  
{  
    range  
    .algorithm1  
    .algorithm2  
    .algorithm3  
    .consumer;  
}
```

**PRINTF DEBUG ALL THE THINGS!**

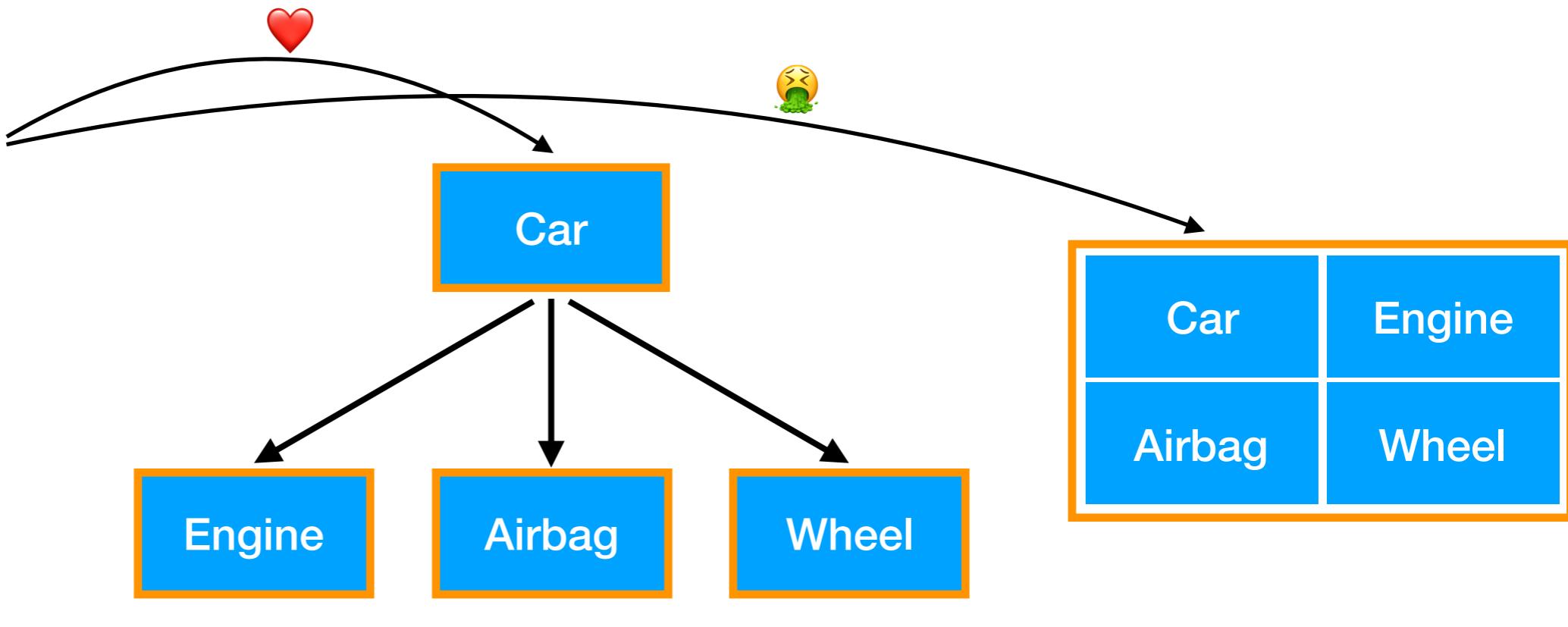


Consumer

# Abstraction Frontier



OOP is what OOP practitioners do



**1: one entity == one class**

**Dogma:** **2: always abstract along entity lines**  
**3: always act from inside the frontier**

# Example

**Athlete**



```
laps: int  
run()
```

# Example

**Athlete**

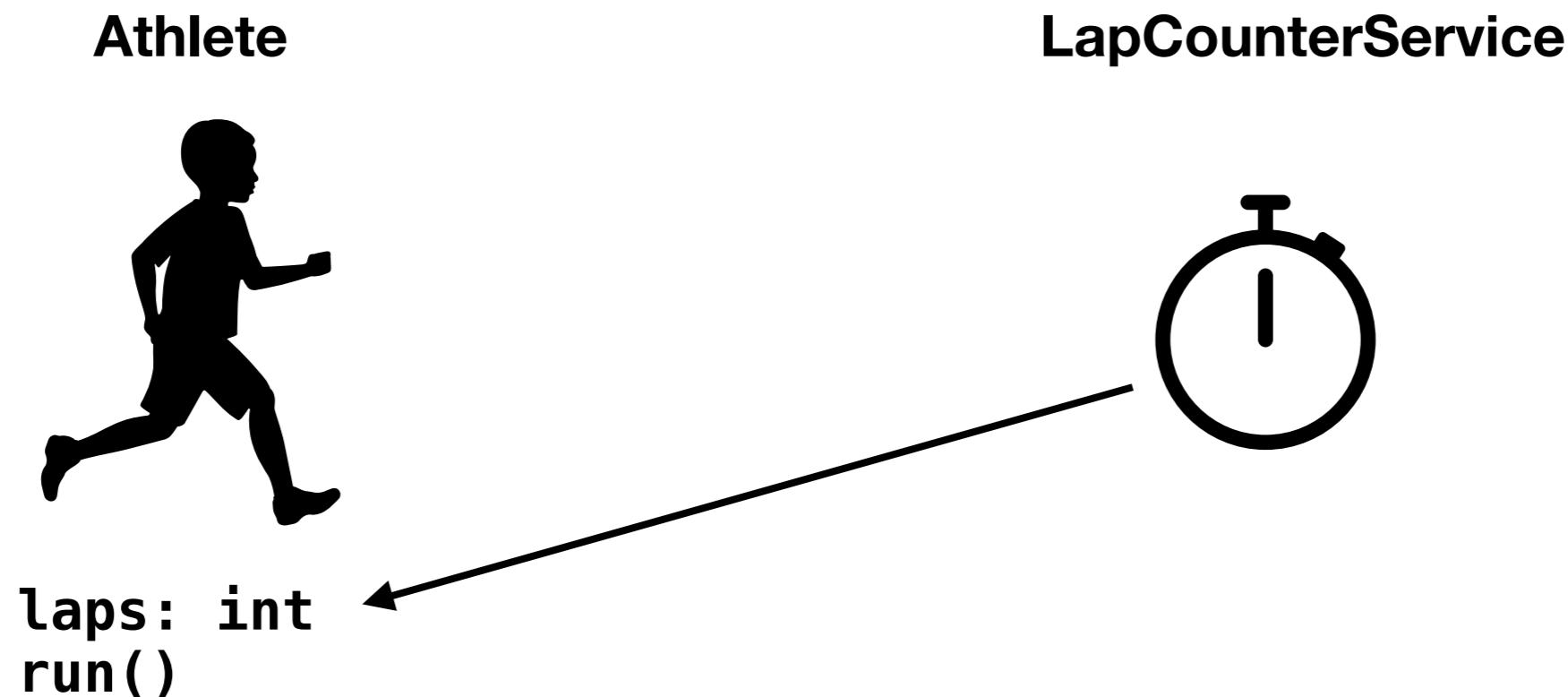


`laps: int  
run()`

**LapCounterService**



# Example



# Example

**Athlete**



**run()**

**LapCounterService**



**Athlete 1: 2 laps  
Athlete 2: 3 laps  
...**

# Example

**Athlete**



`run()`

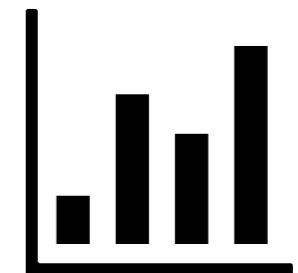
**LapCounterService**



**Athlete 1: 2 laps  
Athlete 2: 3 laps**

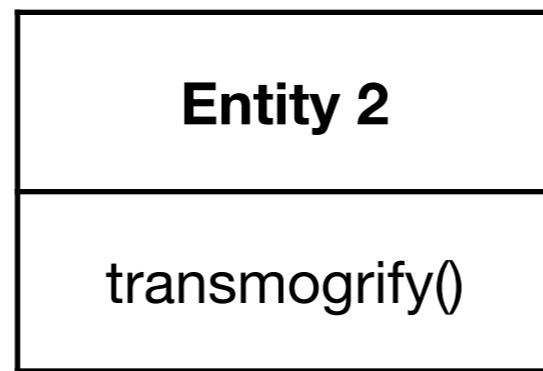
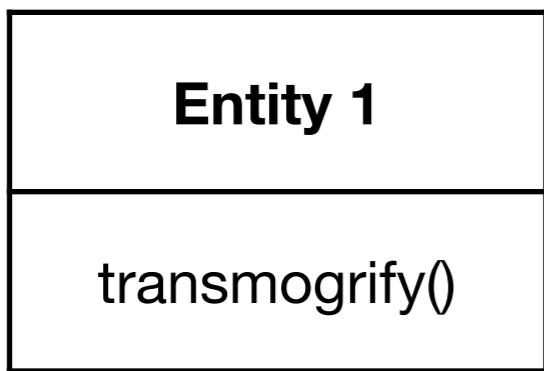
...

**LapStatisticsService**

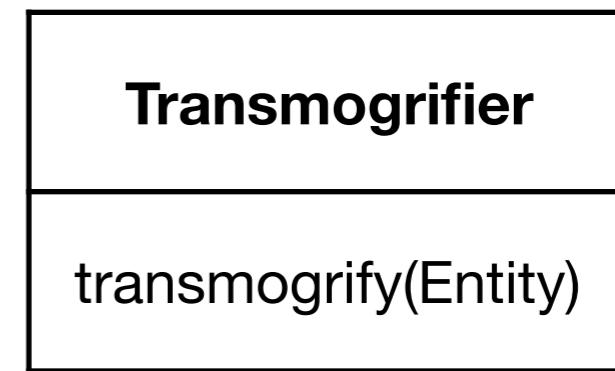
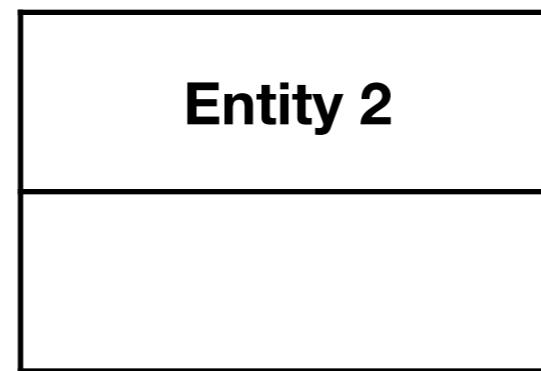
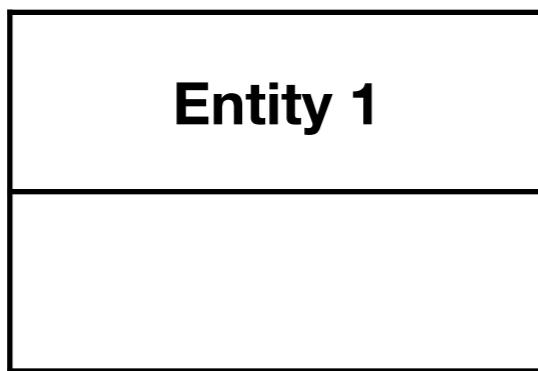


**Average laps: 2.5**

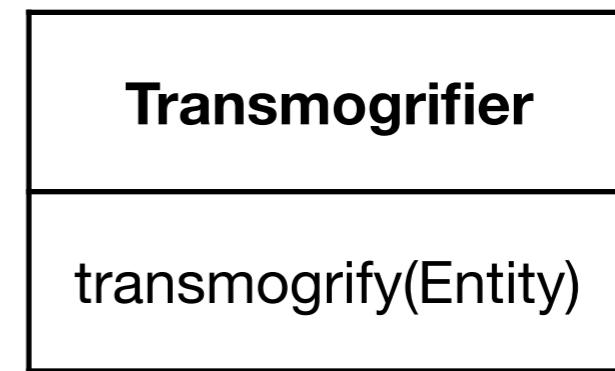
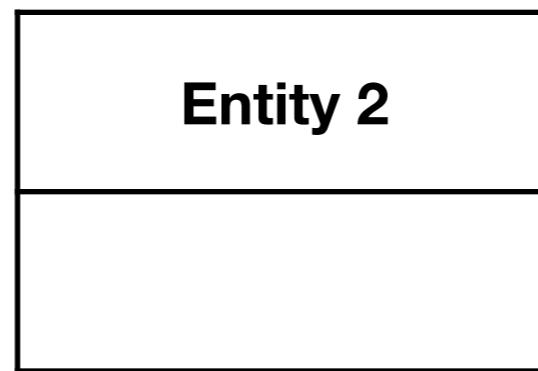
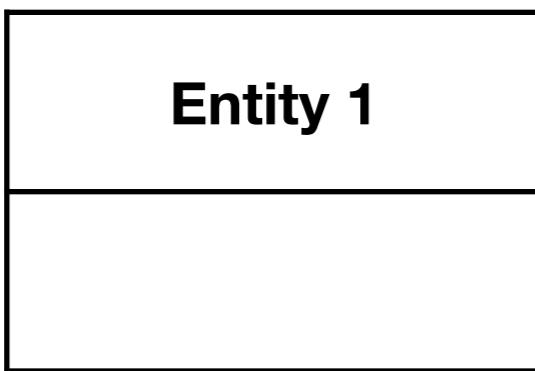
# Service Dispatch



# Service Dispatch



# Service Dispatch



**How do we polymorphically transmogrify?  
That is, how do we dispatch based on the entity argument?**

# Service Dispatch

**Entity 1**

```
visit(Transmogrifier)
```

**Entity 2**

```
visit(Transmogrifier)
```

**Transmogrifier**

```
transmogrify(Entity)
```

**How do we polymorphically transmogrify?  
That is, how do we dispatch based on the entity argument?**

# Service Dispatch

```
1  if (f needs to be virtual)
2      make f a member function of C;
3  else if (f is operator>> or
4          operator<<)
5      {
6          make f a non-member function;
7          if (f needs access to non-public
8              members of C)
9              make f a friend of C;
10         }
11     else if (f needs type conversions
12             on its left-most argument)
13     {
14         make f a non-member function;
15         if (f needs access to non-public
16             members of C)
17             make f a friend of C;
18         }
19     else if (f can be implemented via C's
20             public interface)
21         make f a non-member function;
22     else
23         make f a member function of C;
```

# Service Dispatch

```
1  if (f needs to be virtual)
2      make f a member function of C;           i.e. give up
3  else if (f is operator>> or
4          operator<<)
5      {
6          make f a non-member function;
7          if (f needs access to non-public
8              members of C)
9              make f a friend of C;
10         }
11     else if (f needs type conversions
12             on its left-most argument)
13         {
14             make f a non-member function;
15             if (f needs access to non-public
16                 members of C)
17                 make f a friend of C;
18         }
19     else if (f can be implemented via C's
20             public interface)
21         make f a non-member function;
22     else
23         make f a member function of C;
```

# Open Methods

- Use Jean-Louis Leroy's openmethods.d library [4]

```
import openmethods;
mixin(registerMethods);

void transmogrify(virtual!Entity);

@method
void _transmogrify(Entity1 entity) {
    (...)
}

@method
void _transmogrify(Entity2 entity) {
    (...)
}
```

# The Expressive Style

- Orthodox OOP

```
class A
{
    void foo() {
        ...
    }
    ...
}

A a = new A;
a.foo();
```

# The Expressive Style

- Maximal encapsulation  
OOP
- foo only depends on the  
public interface of A

```
// module 1
class A
{
    ...
}

// module 2
void foo(A a) {
    ...
}

A a = new A;
a.foo(); // UFCS
```

# The Expressive Style

- Generic / Dbl design
  - You can't go back to the orthodox style
  - To which class does foo belong to?
  - Compile-time design

```
// module 1
class A
{
    ...
}

// module 2
void foo(T)(T a) {
    ...
}

A a = new A;
a.foo(); // UFCS
```

# The Expressive Style

- Expressive design

- The runtime counterpart to the generic / Dbl design
- Open methods ⊃ member methods

```
// module 1
class A
{
    ...
}
```

```
// module 2
void foo(virtual!Object a) {
    ...
}
```

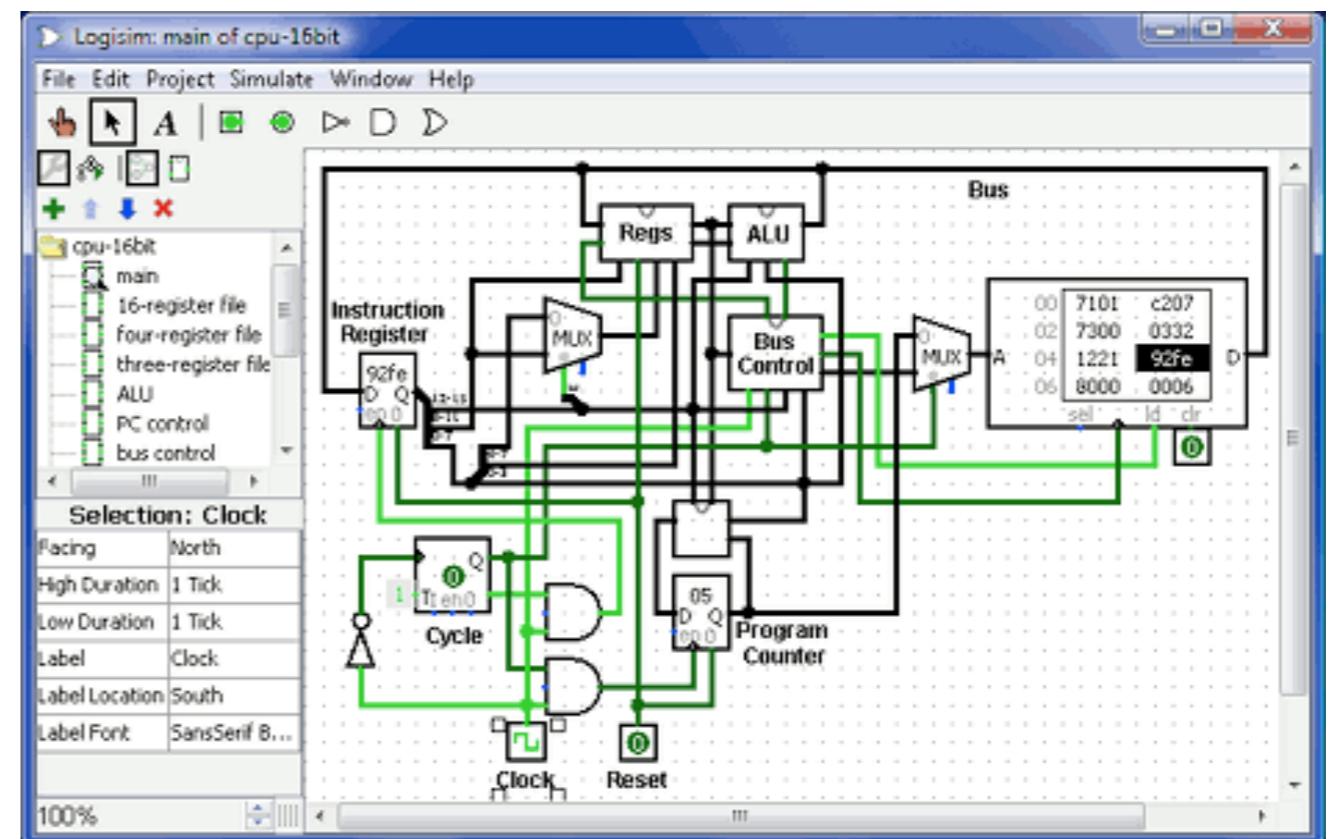
```
@method
void _foo(A entity) {
    ...
}
```

```
A a = new A;
a.foo(); // UFCS
```

# Case Study

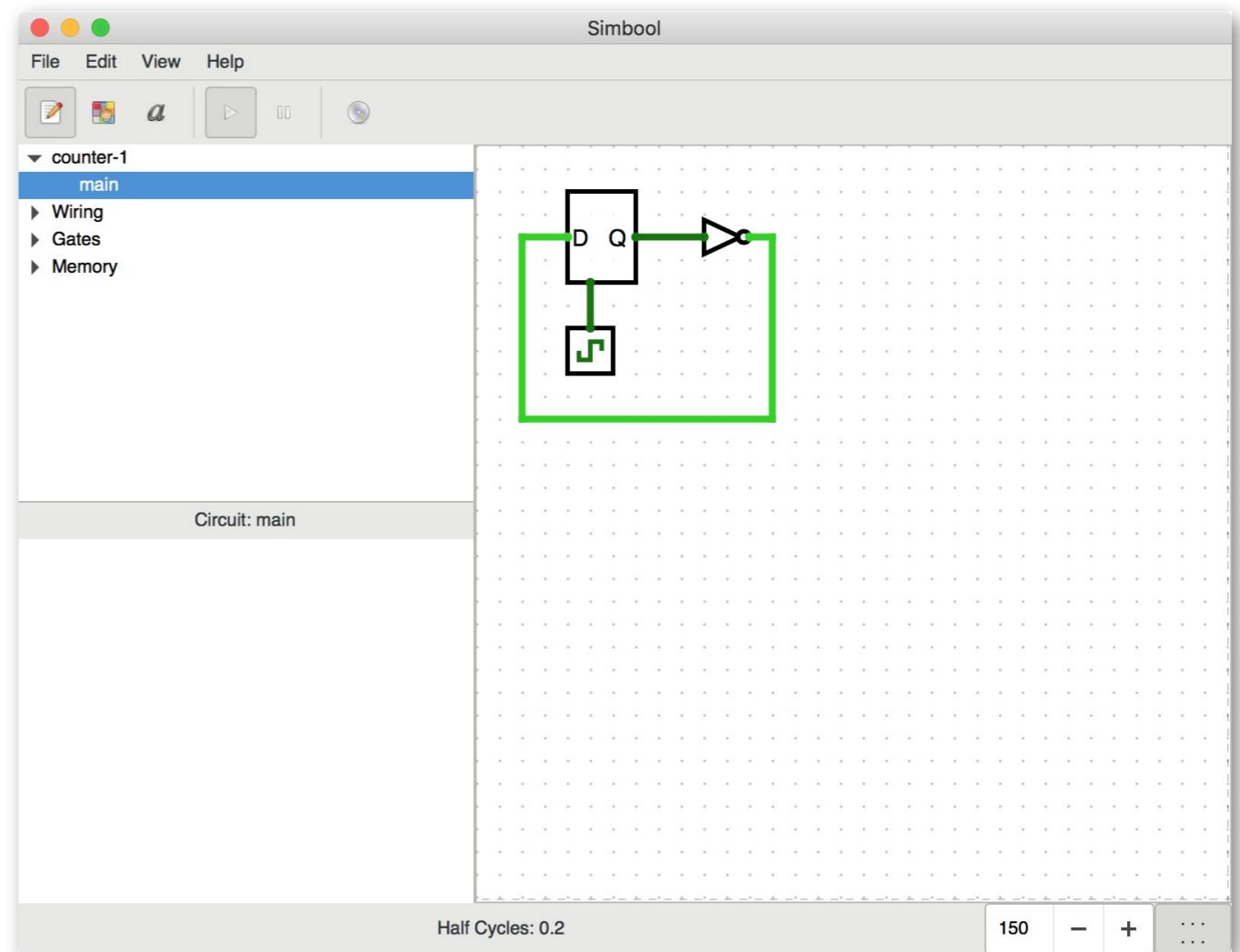
# Logisim

- Good educational app
- No longer maintained
- Confusing Java OOP architecture
- Poor simulation performance
- Non-native UI



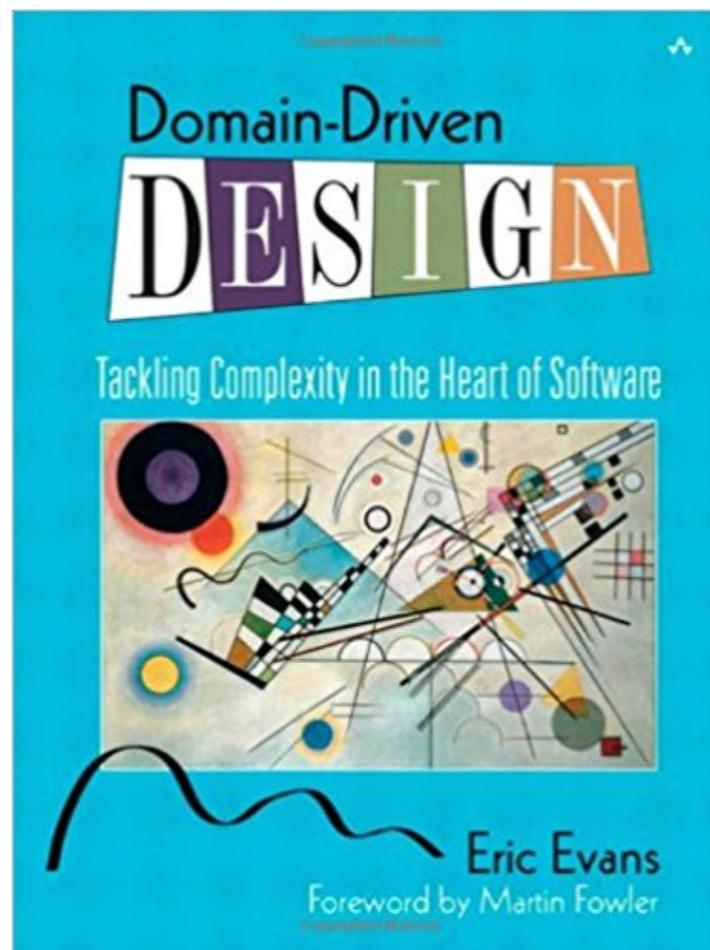
# Simbool

- Logisim compatibility
- Better multi-valued logic support, better timing model, bidirectional ports, etc.
- Export to Verilog / VHDL / DHDL (“run” on FPGAs)
- JIT accelerated simulator
- Native UI planned for Windows, macOS, Linux



# OO Design Example

- Domain-driven design book example



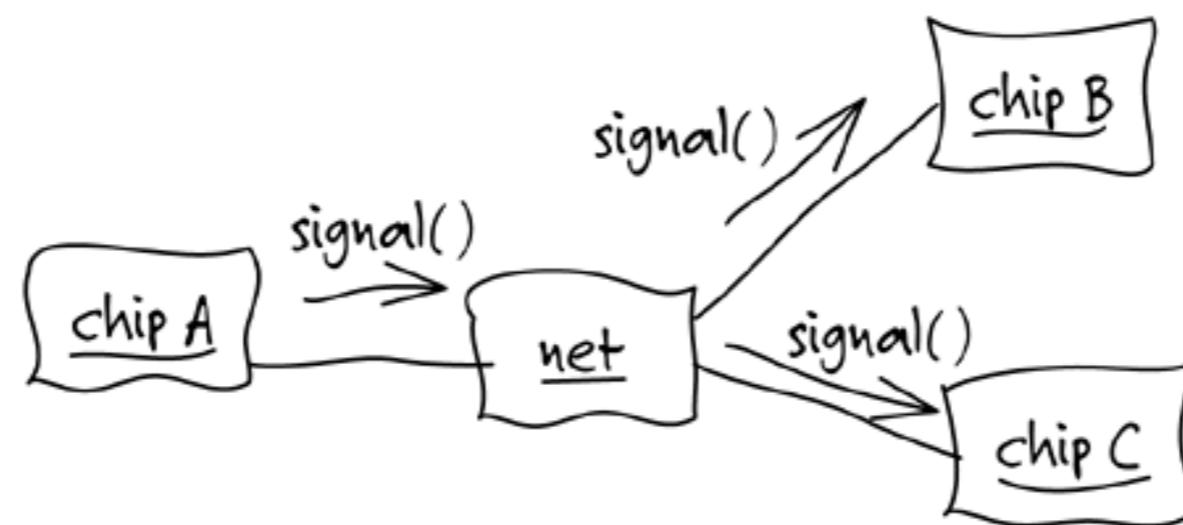
# OO Design Example

- Domain-driven design book example



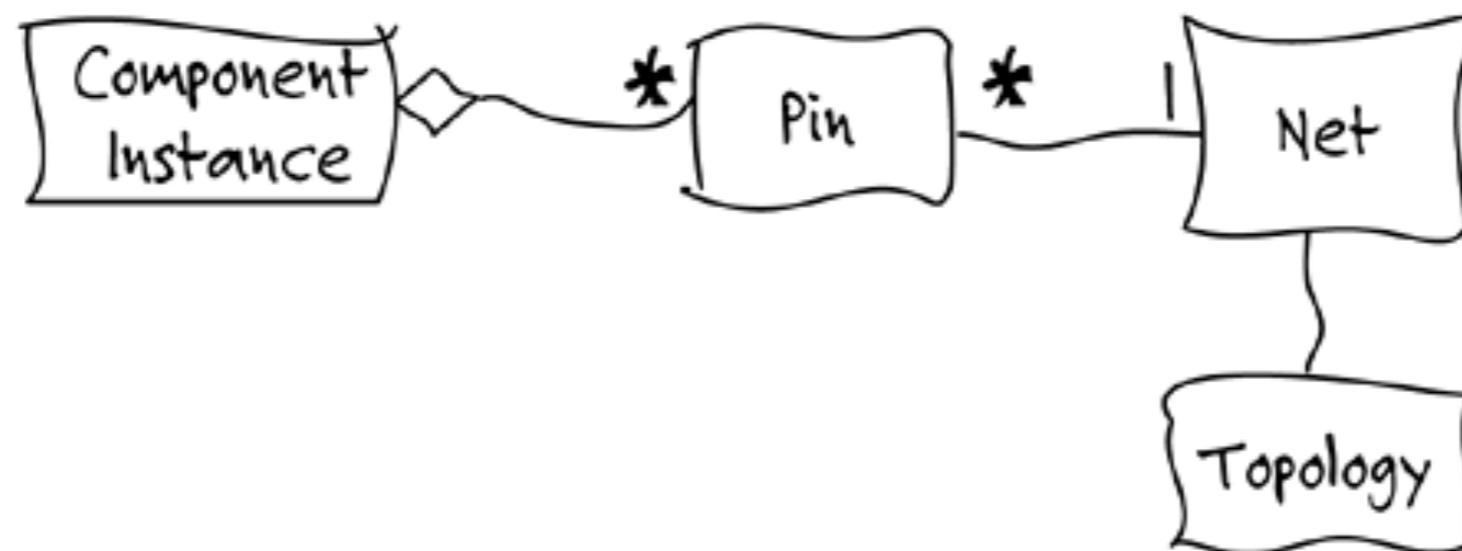
# OO Design Example

- Domain-driven design book example



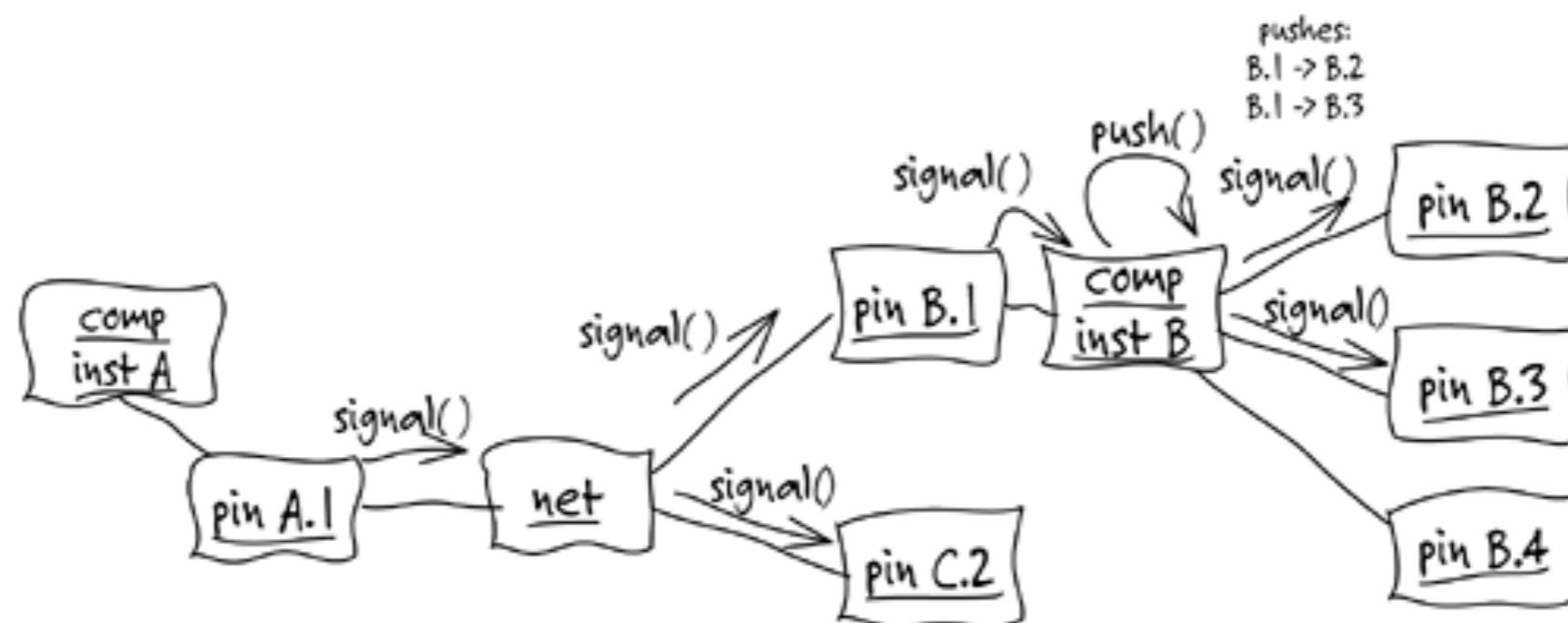
# OO Design Example

- Domain-driven design book example



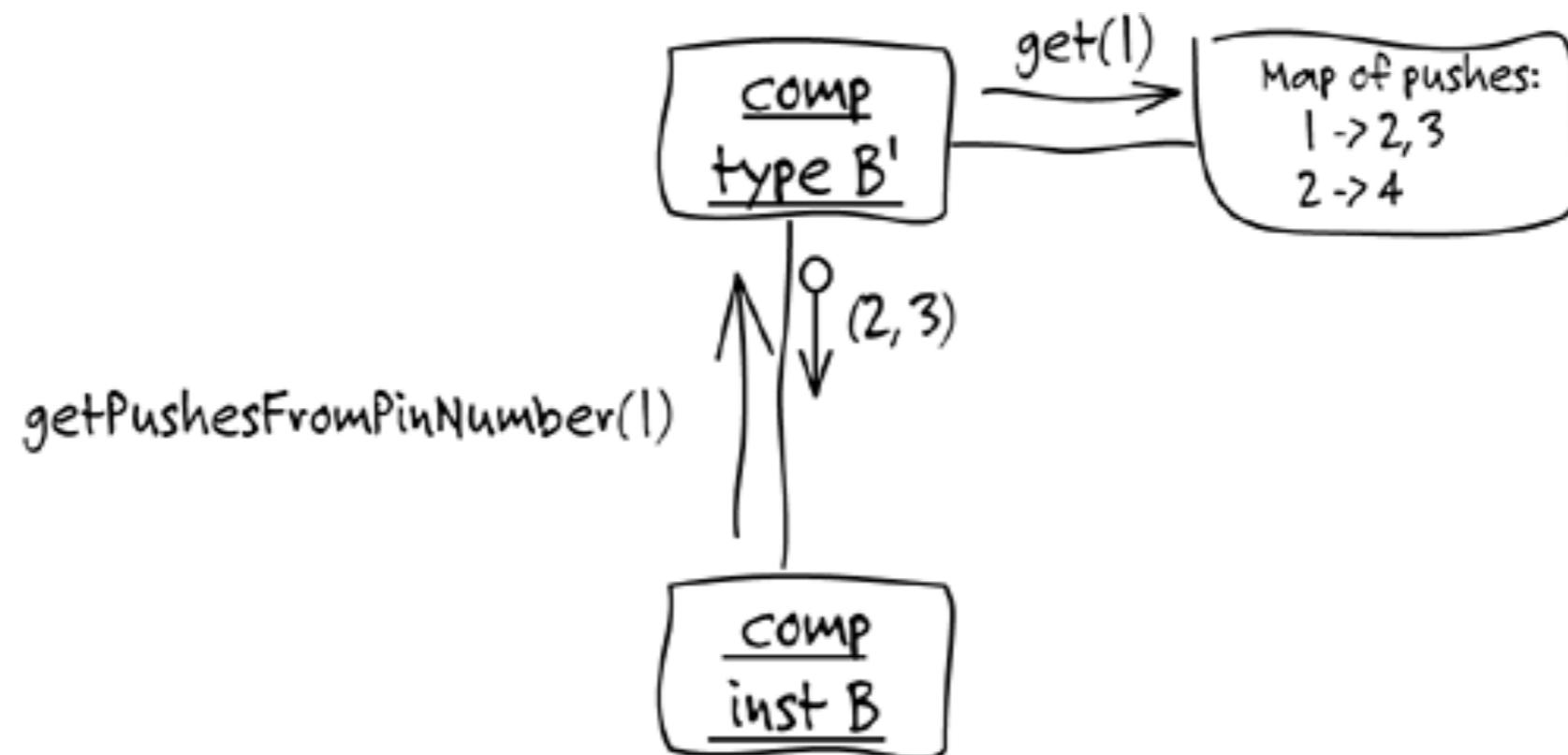
# OO Design Example

- Domain-driven design book example



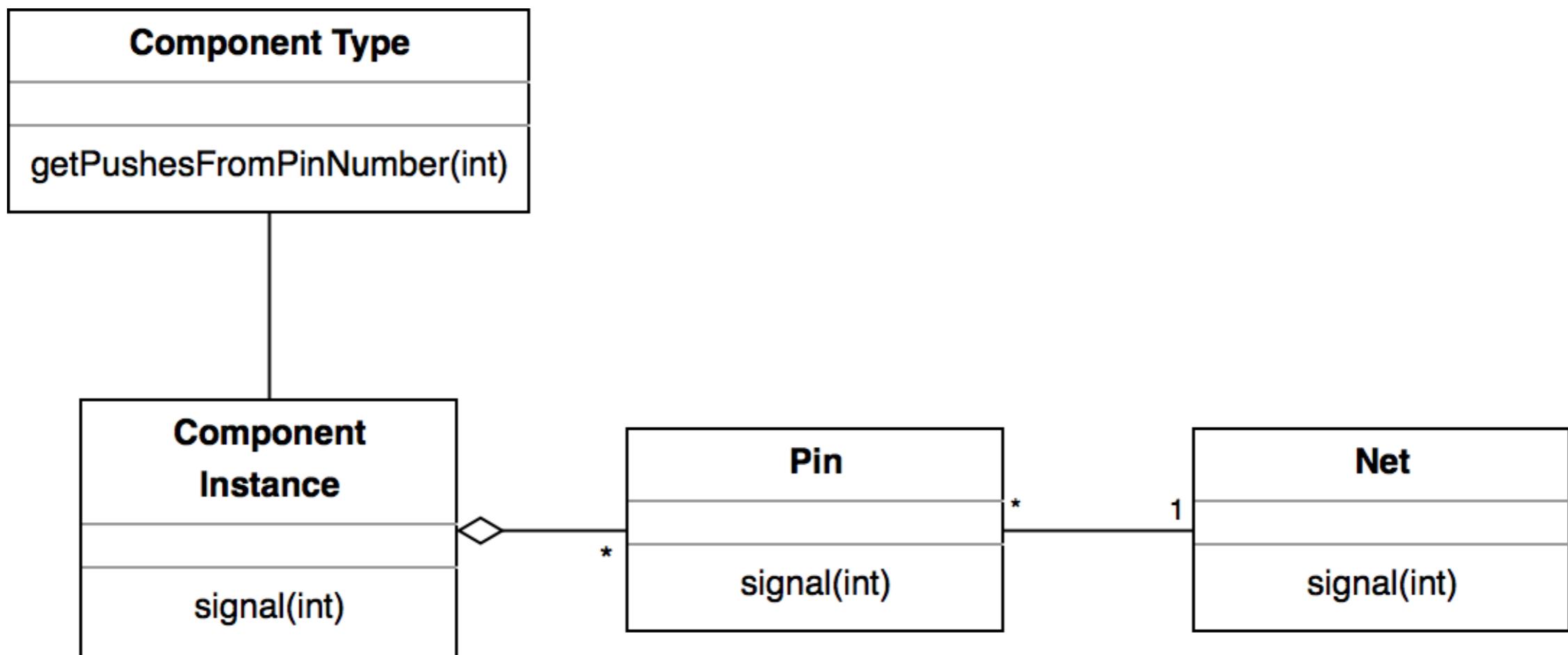
# OO Design Example

- Domain-driven design book example



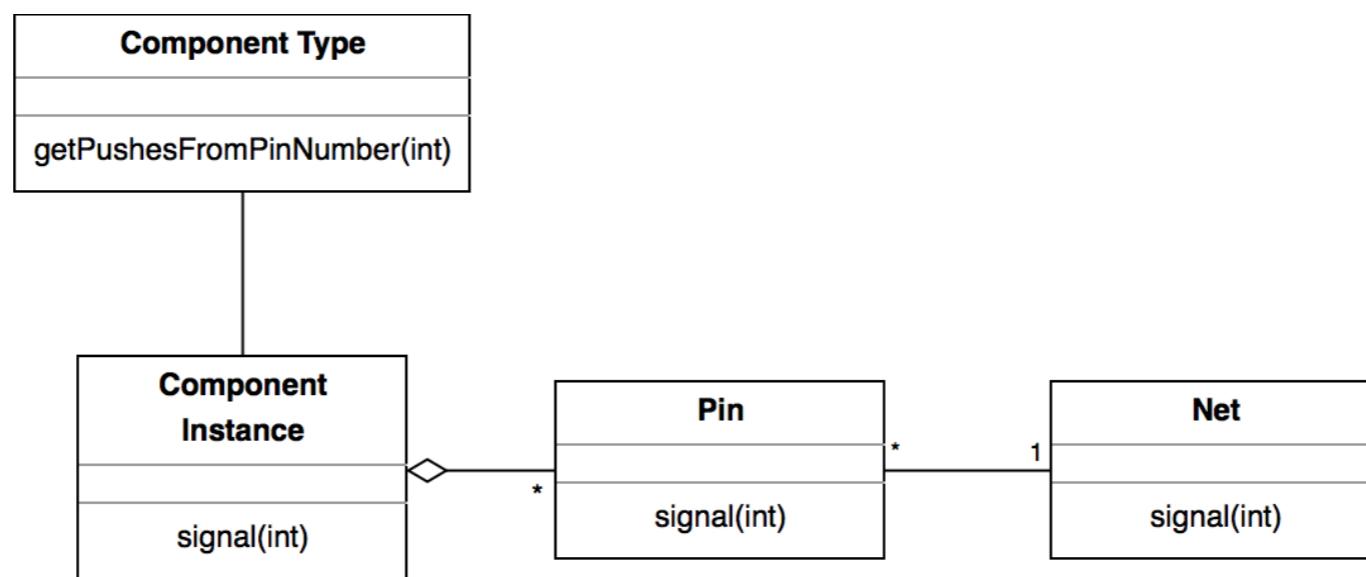
# OO Design Example

- Domain-driven design book example



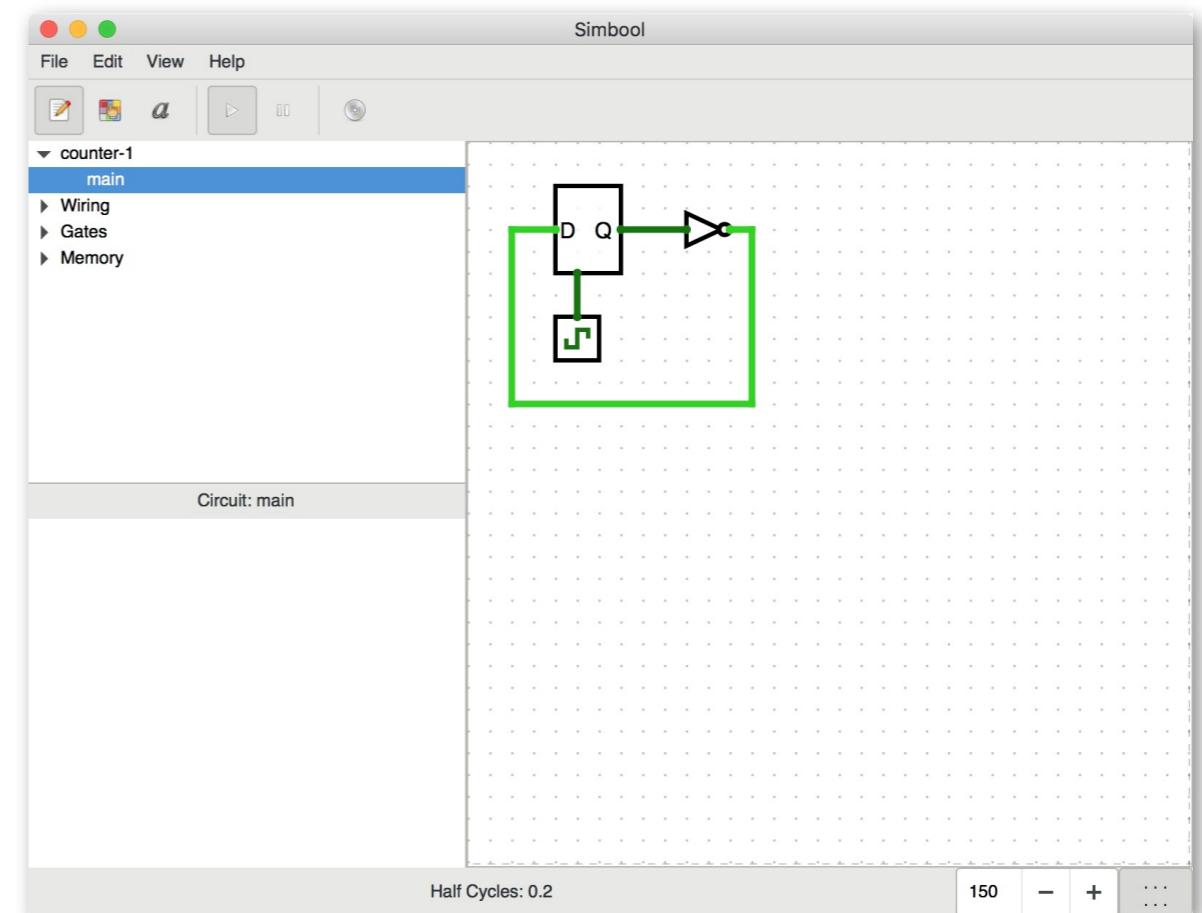
# OO Design Example

- This is a model of the *problem*
- Why are we assuming the most straightforward *solution* looks like the problem?
- Why are we assuming the *component* stores the pushes?



# Simbool Design

- Classes?
  - Circuit
  - Component
  - Pin
  - Port
  - Value
  - Wire
  - ...
- Responsibilities?



# Simbool Design

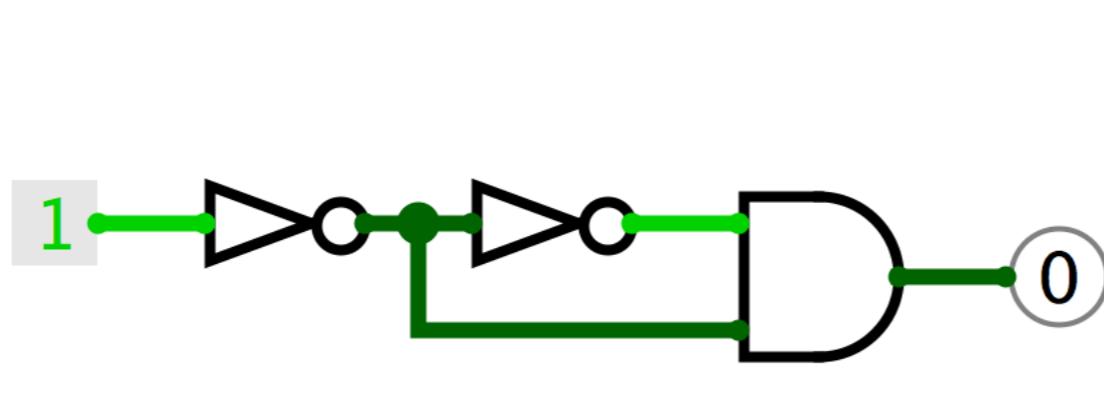
- A revisionist account of how the design came to be...



```
bool input;  
bool output;  
output = !input;
```

- No classes yet. I was just thinking about the computation

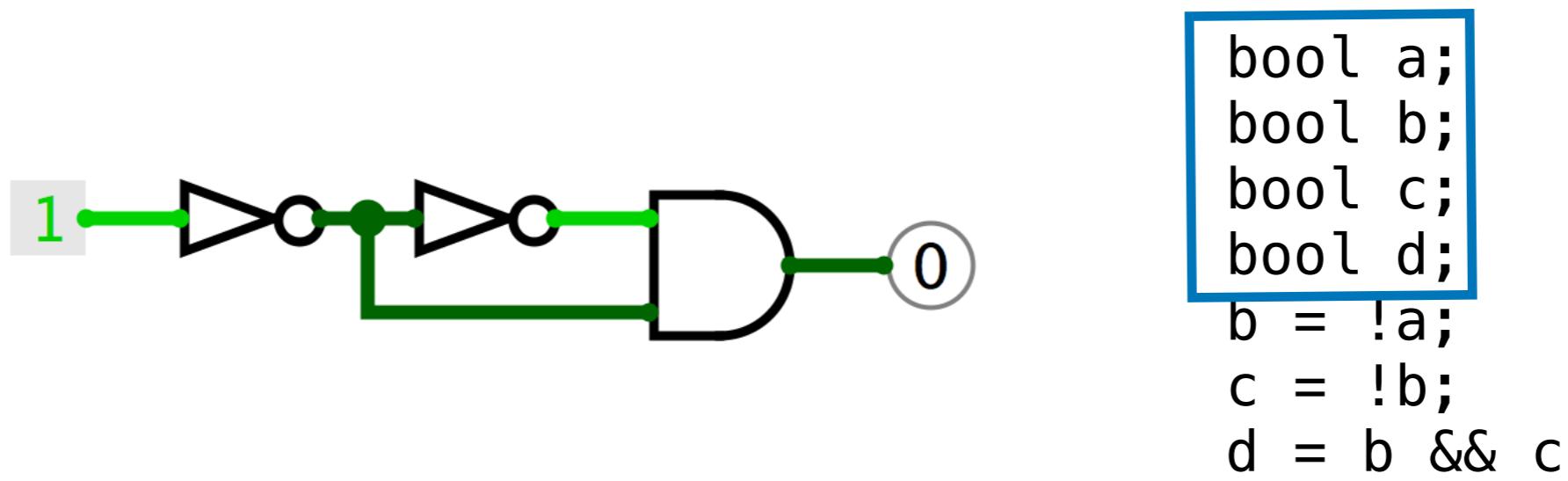
# Simbool Design



```
bool a;  
bool b;  
bool c;  
bool d;  
b = !a;  
c = !b;  
d = b && c
```

- How do we generalize this?

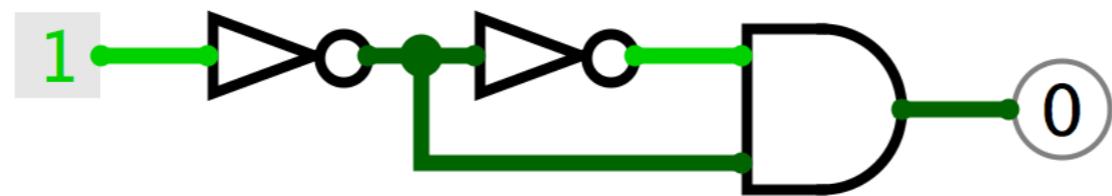
# Simbool Design



- How do we generalize this?

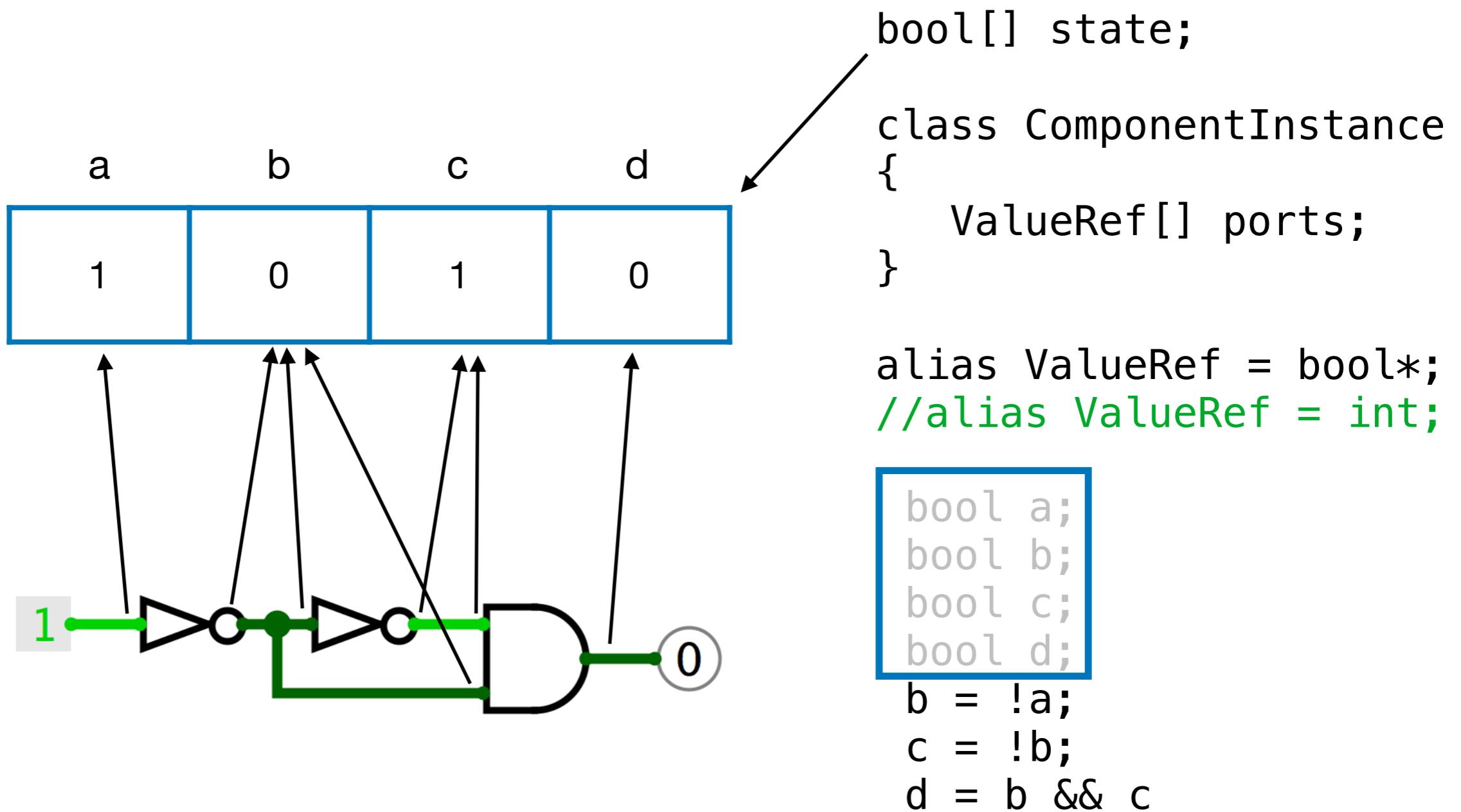
# Simbool Design

```
bool[] state;  
  
class ComponentInstance  
{  
    ValueRef[] ports;  
}  
  
alias ValueRef = bool*;  
//alias ValueRef = int;
```



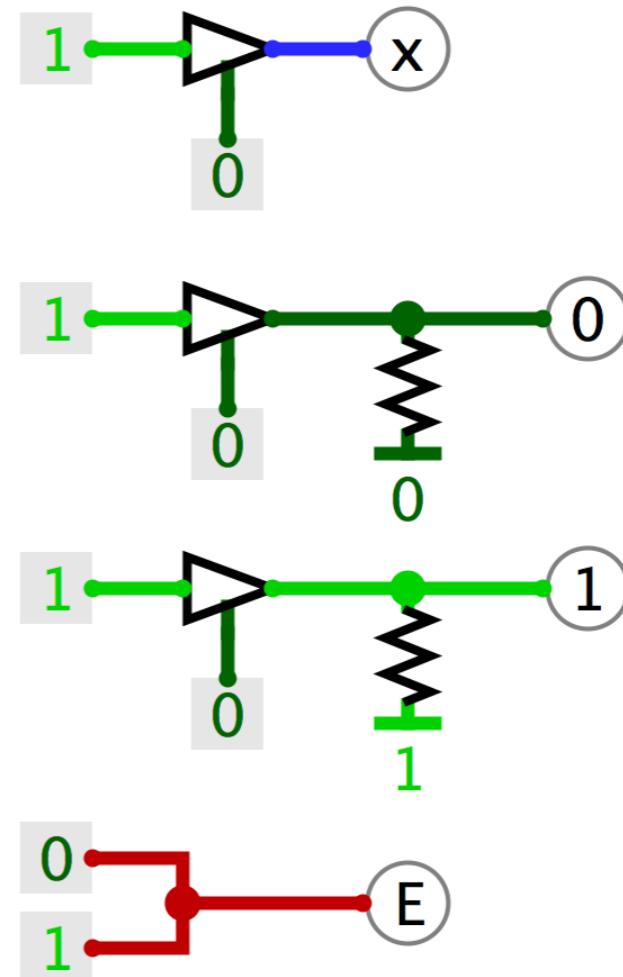
```
bool a;  
bool b;  
bool c;  
bool d;  
b = !a;  
c = !b;  
d = b && c
```

# Simbool Design

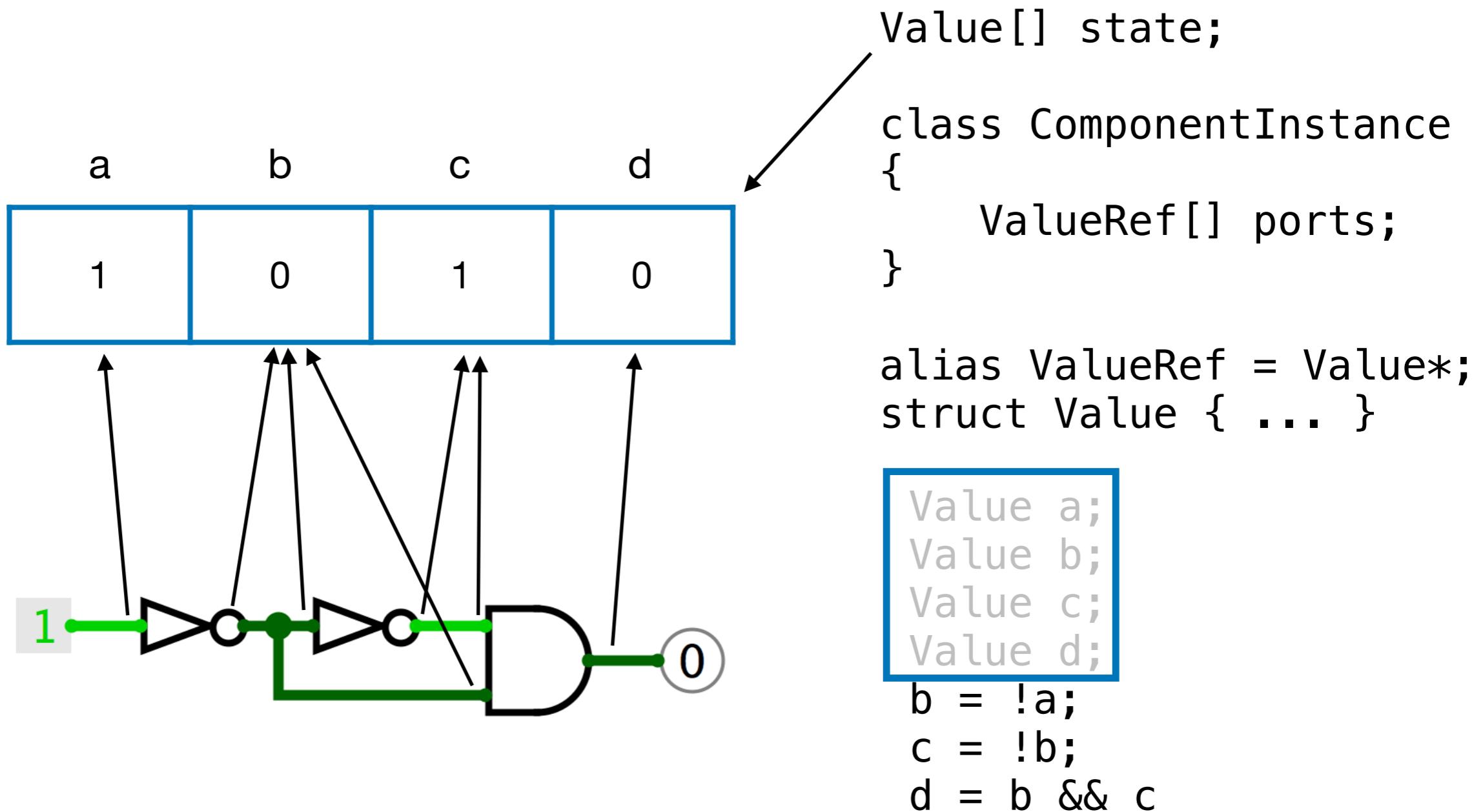


# Simbool Design

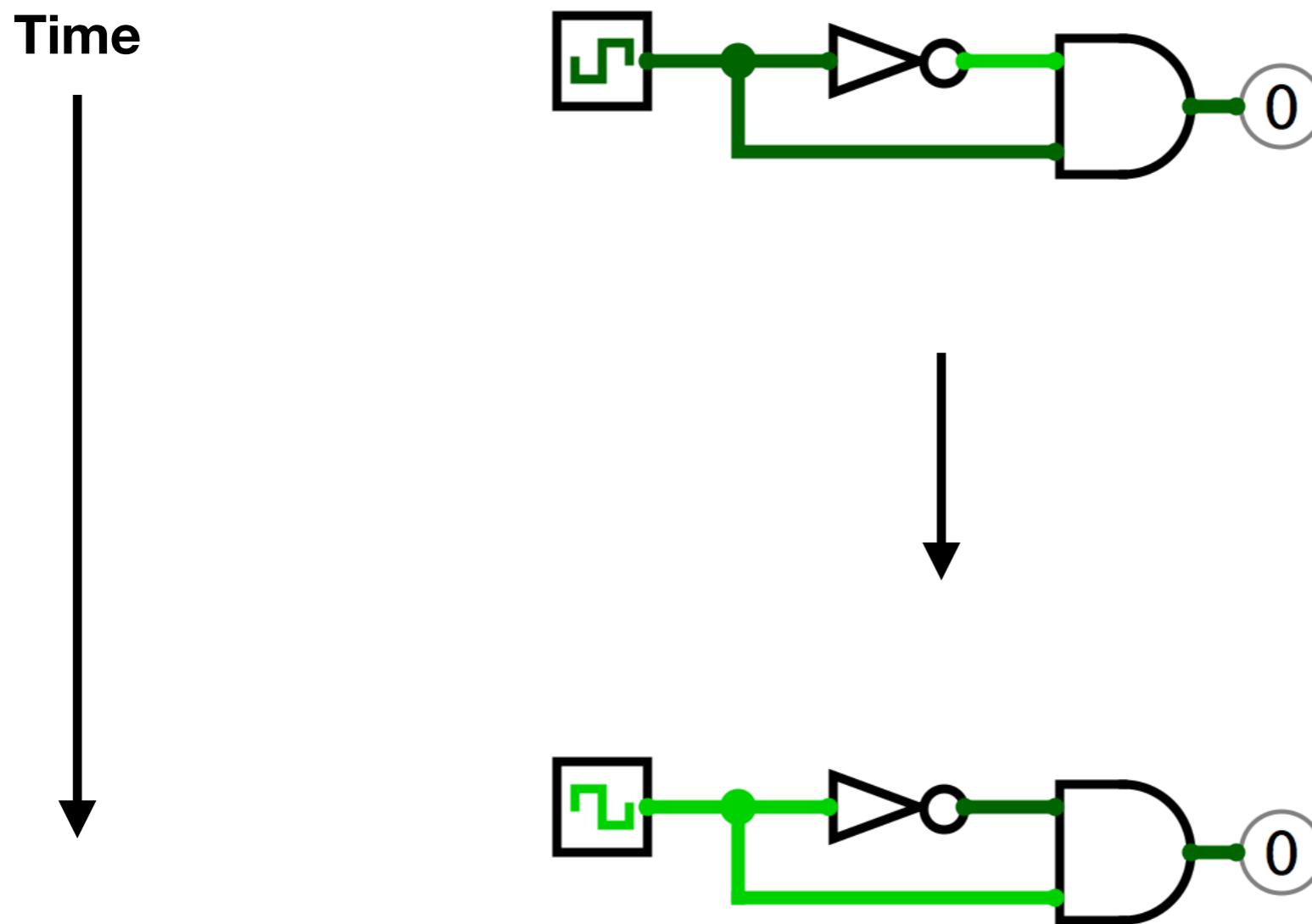
- Many-valued logic
  - False (0)
  - True (1)
  - Floating (high impedance)
  - Weak low (pull-down resistor)
  - Weak high (pull-up resistor)
  - Forcing unknown / error
  - Weak unknown



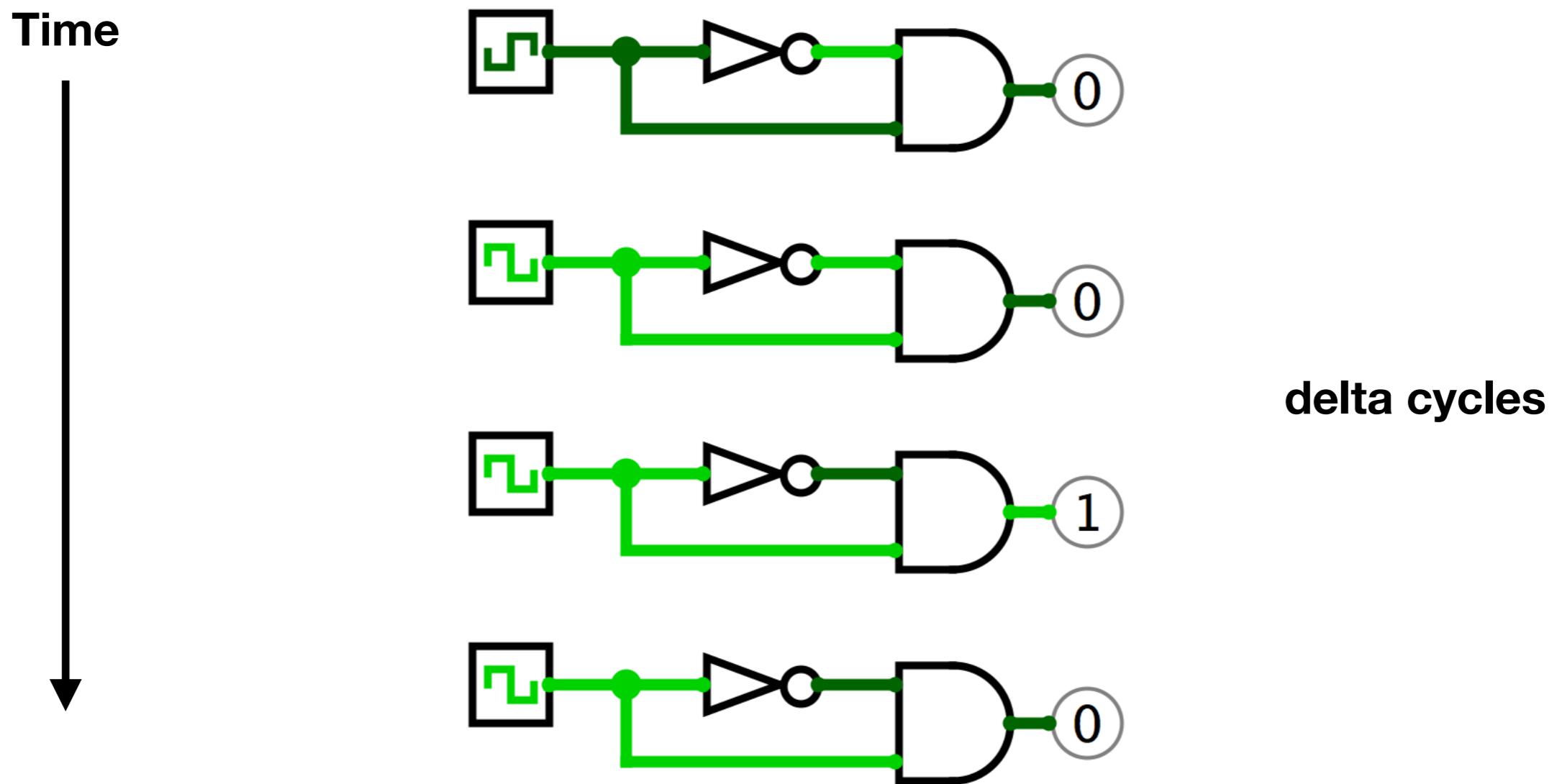
# Simbool Design



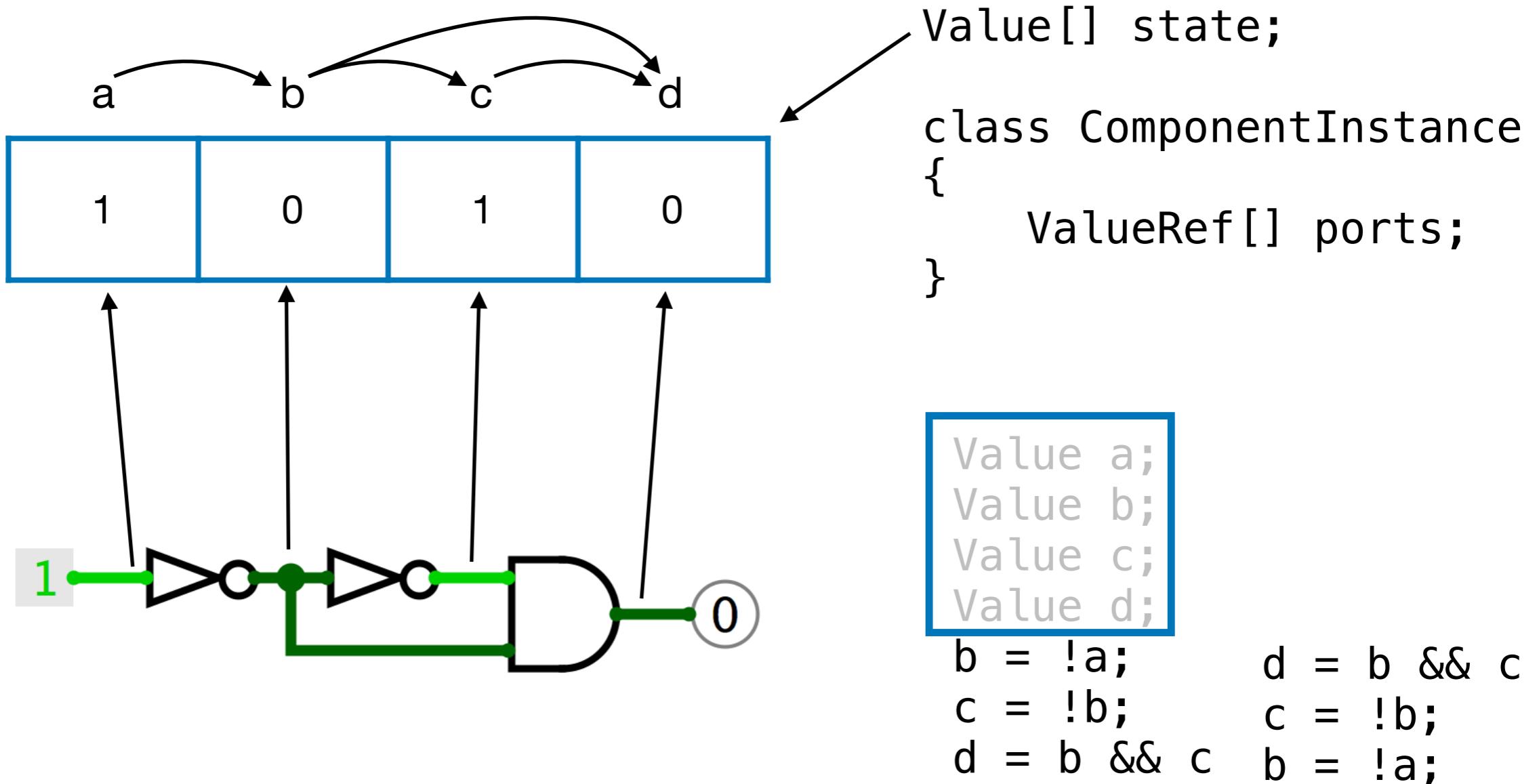
# Simbool Design



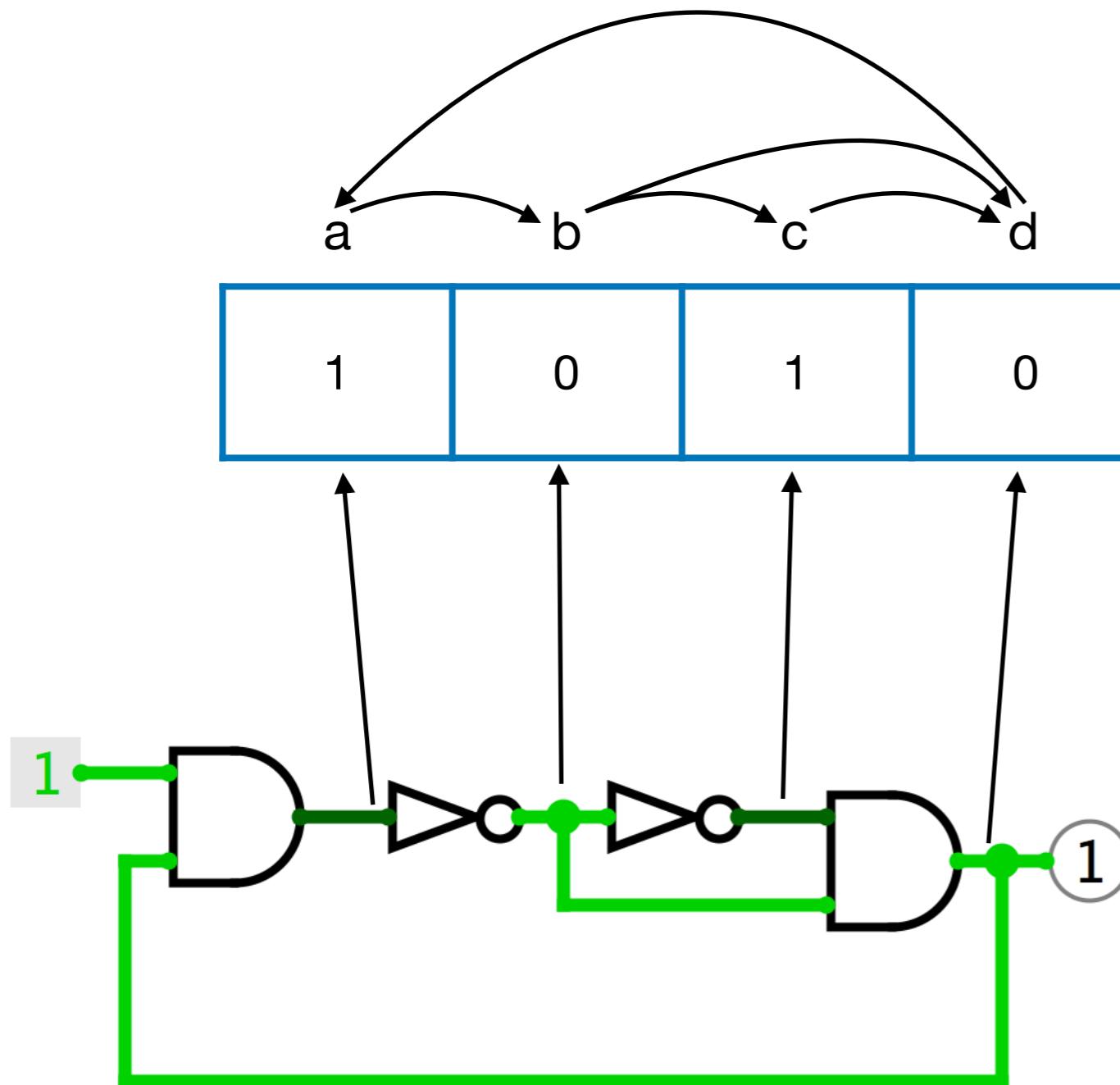
# Simbool Design



# Simbool Design



# Simbool Design

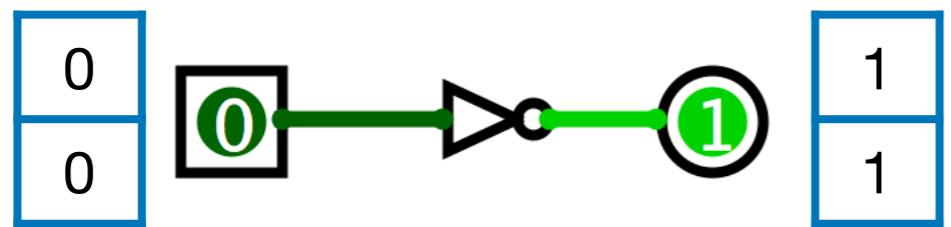


```
Value[] state;  
class ComponentInstance  
{  
    ValueRef[] ports;  
}
```

```
Value a;  
Value b;  
Value c;  
Value d;
```

```
b = !a;          d = b && c  
c = !b;          c = !b;  
d = b && c      b = !a;
```

# Simbool Design

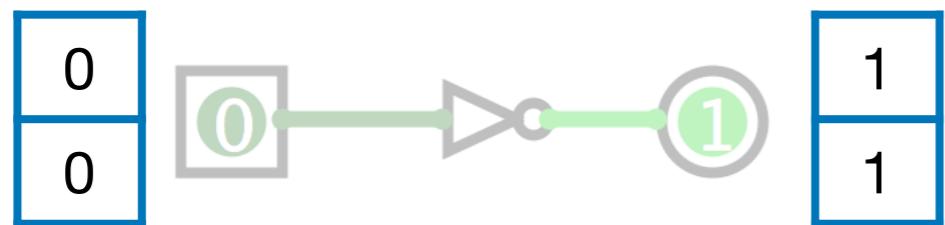


```
Value[2][] values;
```

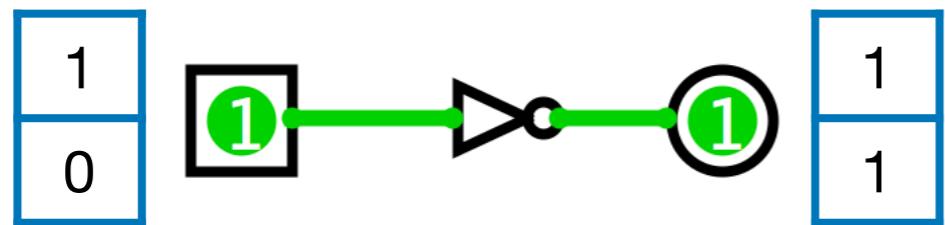
```
class ComponentInstance  
{  
    ValueRef[] ports;  
}
```

```
alias ValueRef = Value[2]*;  
struct Value { ... }
```

# Simbool Design



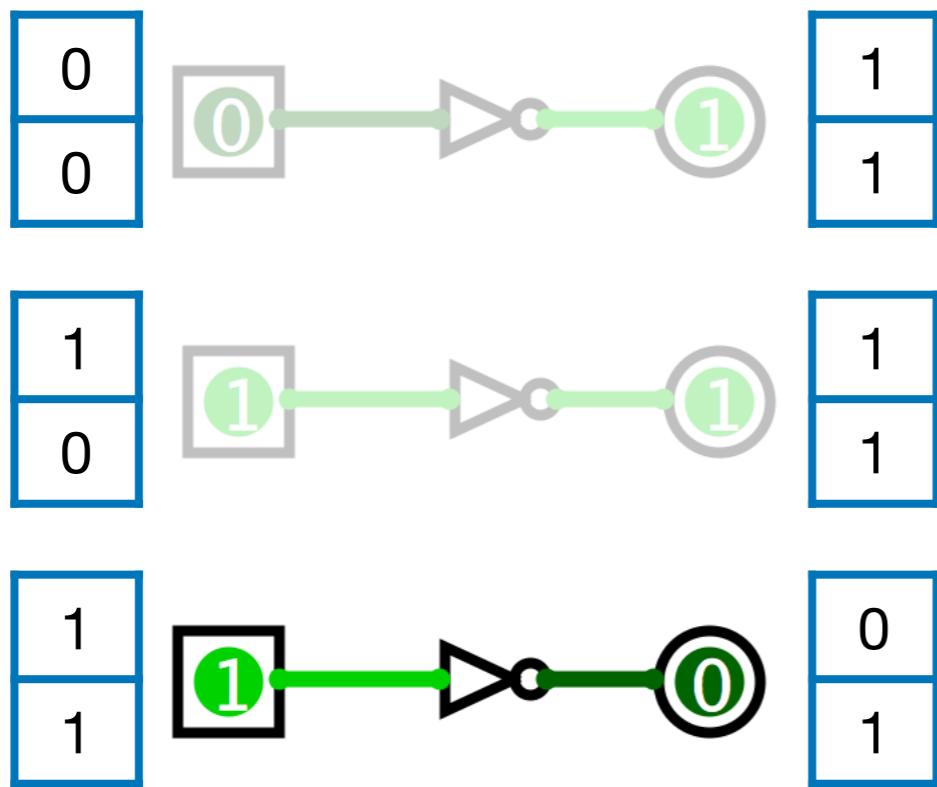
```
Value[2][] values;
```



```
class ComponentInstance  
{  
    ValueRef[] ports;  
}
```

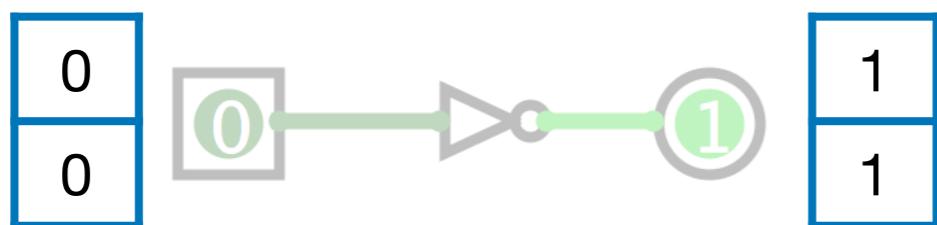
```
alias ValueRef = Value[2]*;  
struct Value { ... }
```

# Simbool Design

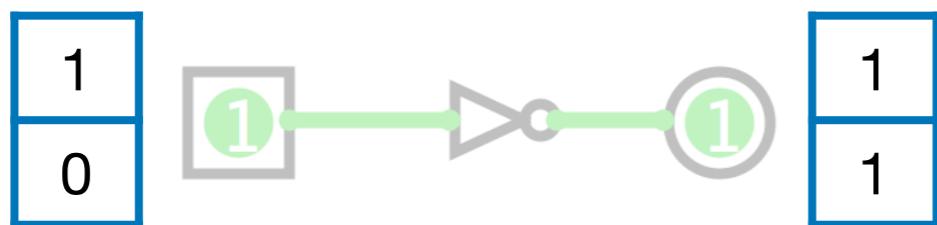


```
Value[2][] values;  
  
class ComponentInstance  
{  
    ValueRef[] ports;  
}  
  
alias ValueRef = Value[2]*;  
struct Value { ... }
```

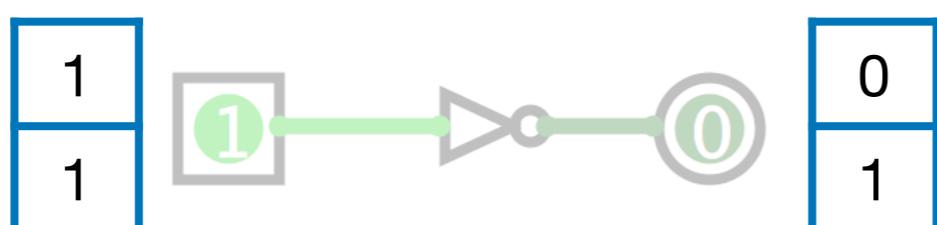
# Simbool Design



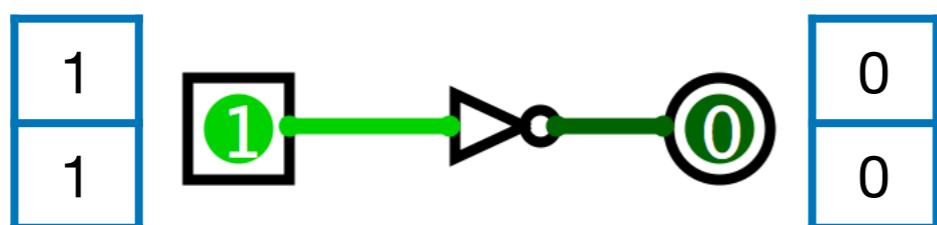
```
Value[2][] values;
```



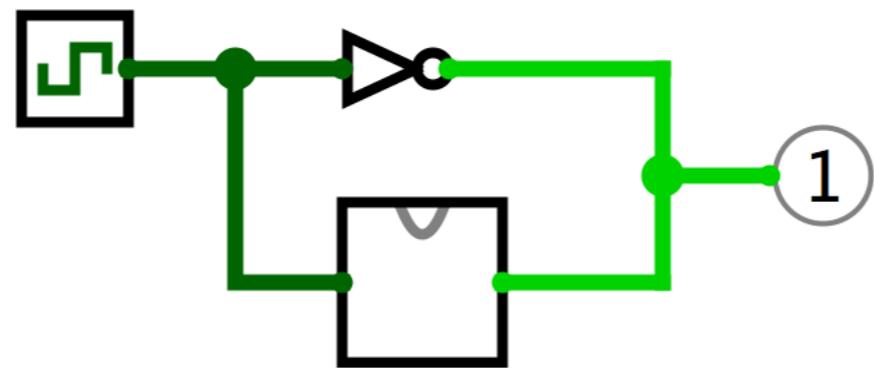
```
class ComponentInstance  
{  
    ValueRef[] ports;  
}
```



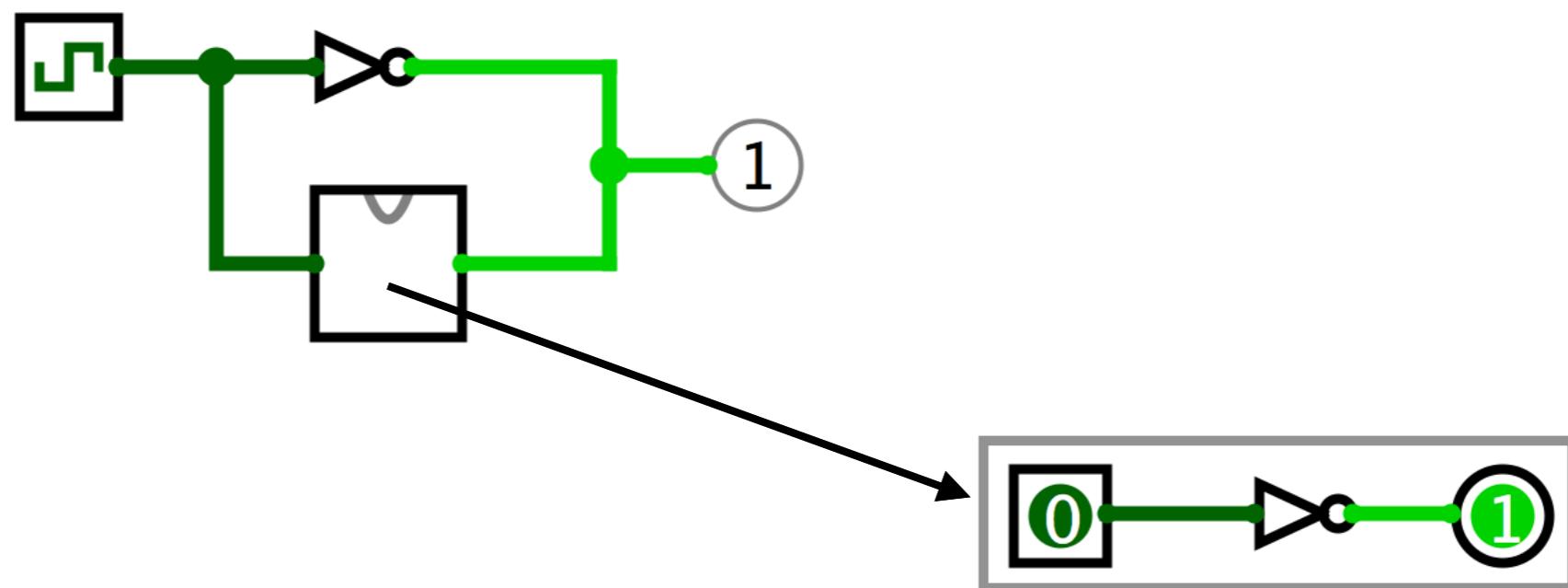
```
alias ValueRef = Value[2]*;  
struct Value { ... }
```



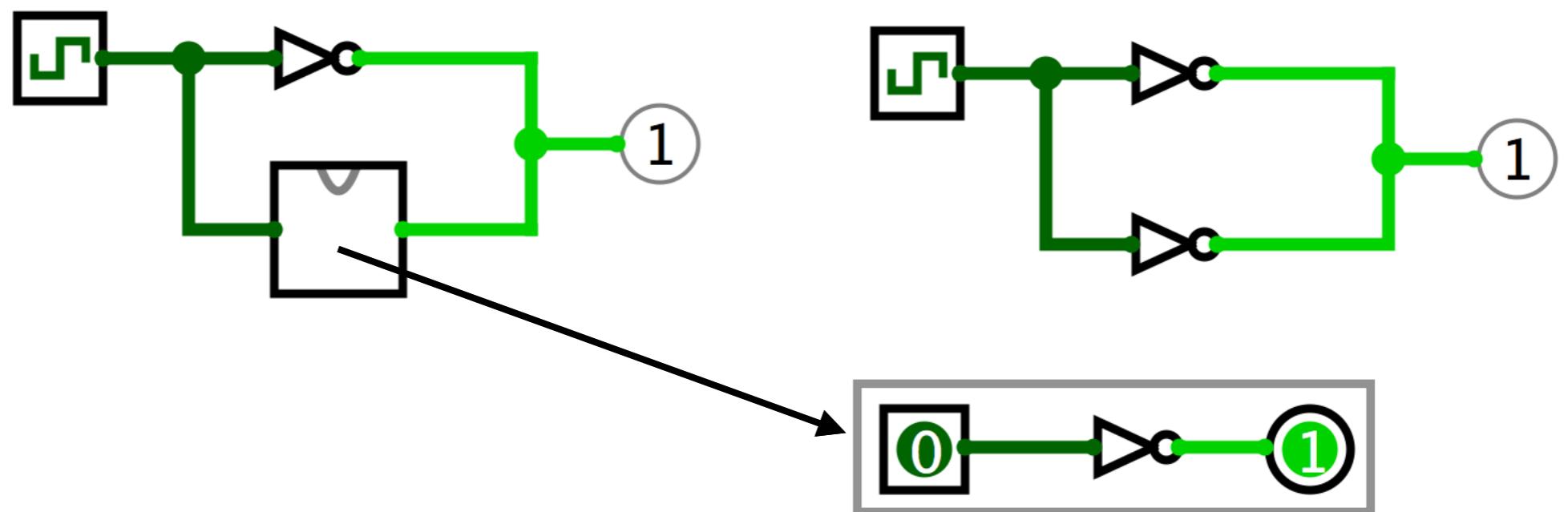
# Simbool Design



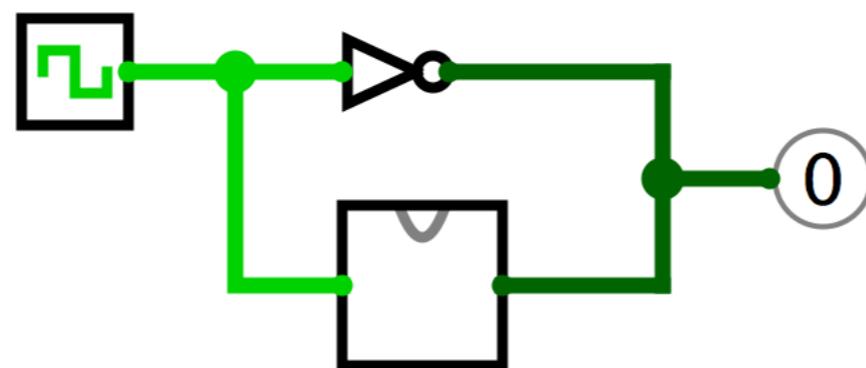
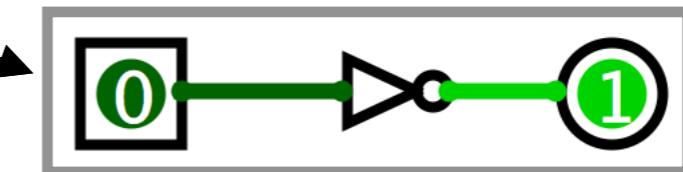
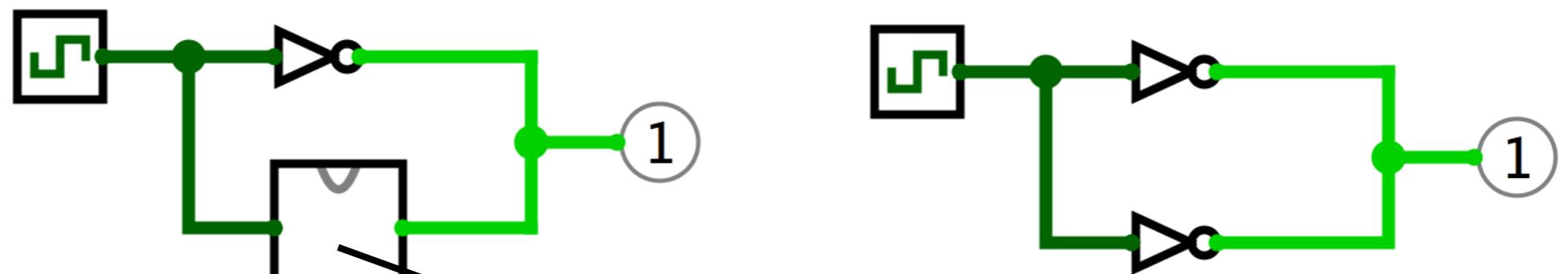
# Simbool Design



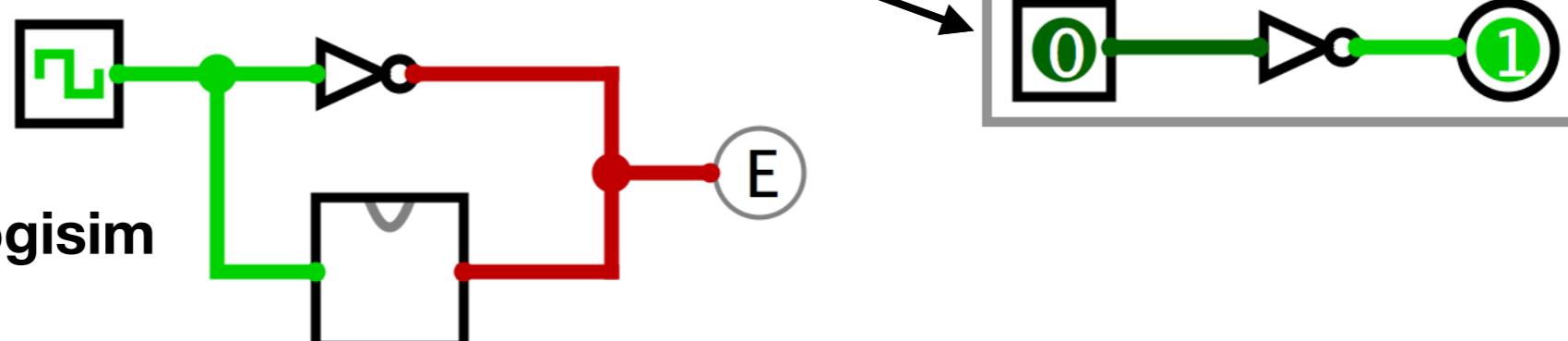
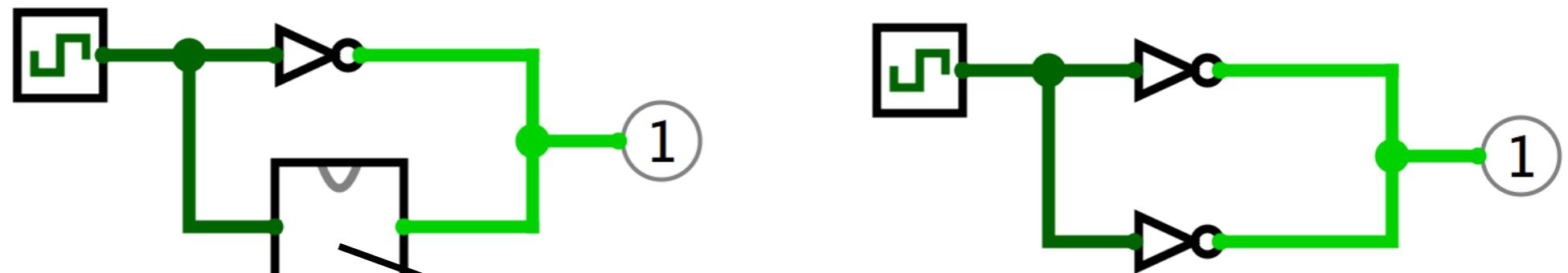
# Simbool Design



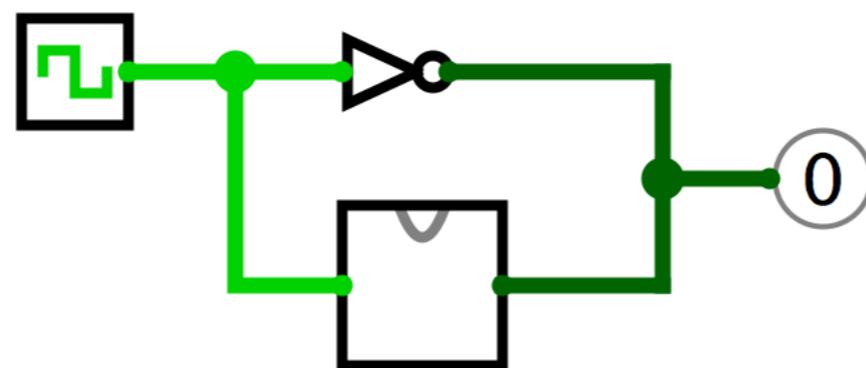
# Simbool Design



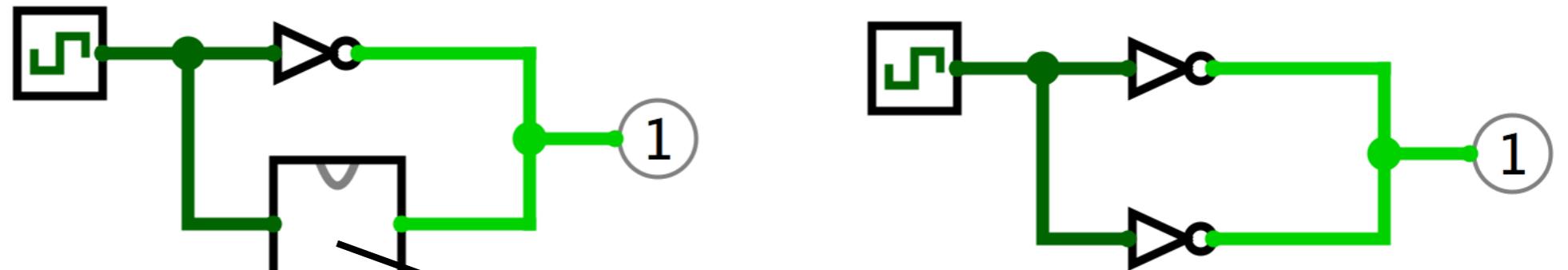
# Simbool Design



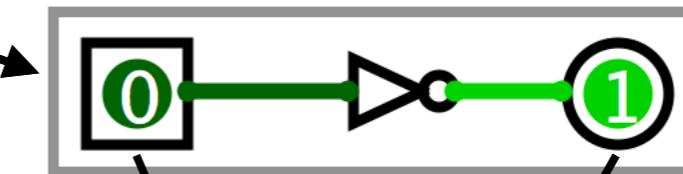
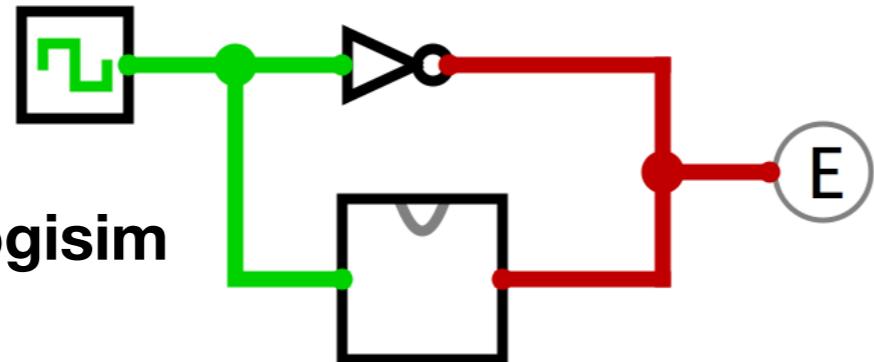
# This happens in Logisim



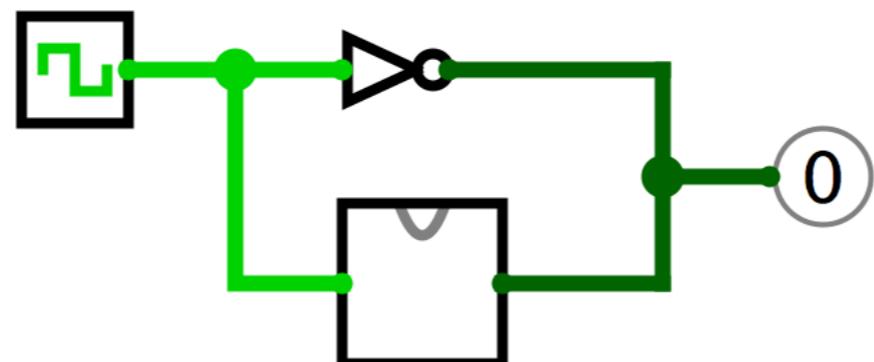
# Simbool Design



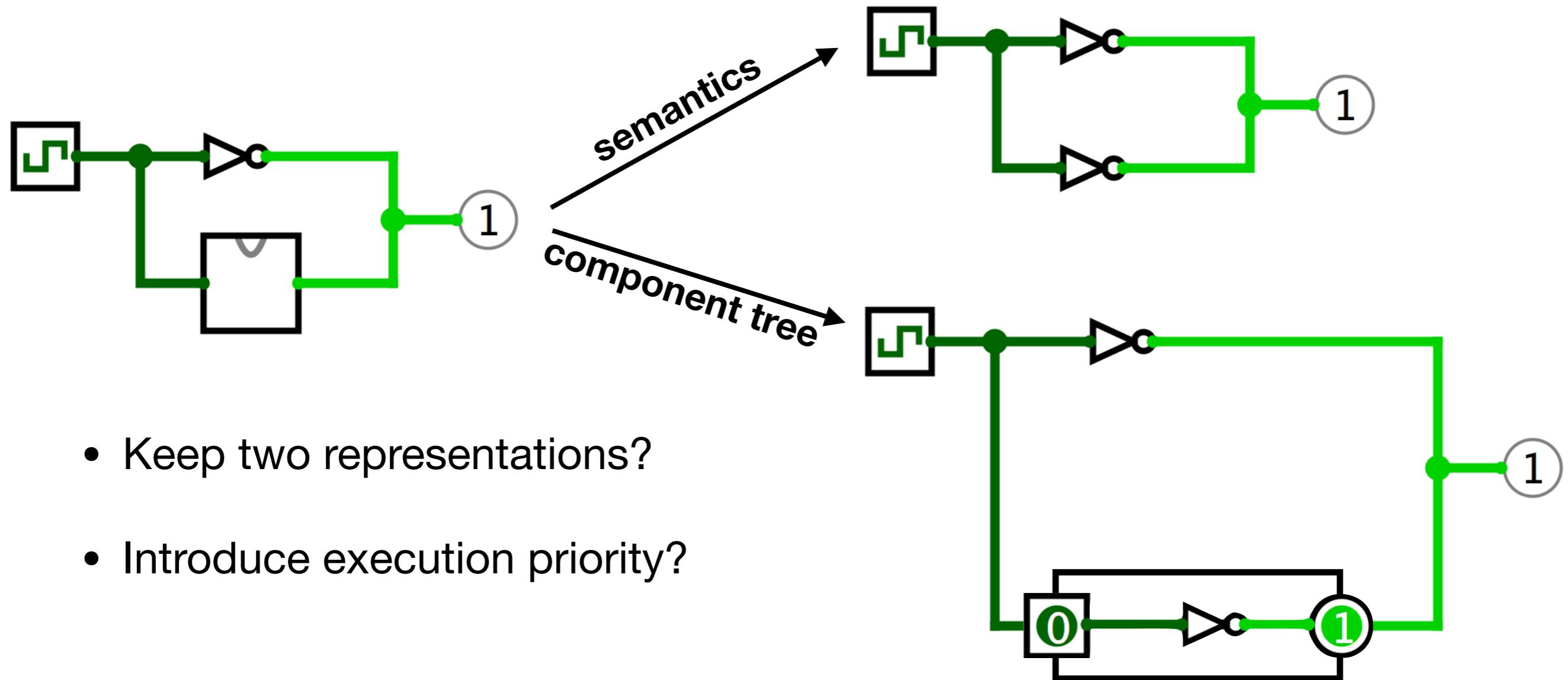
This happens in Logisim



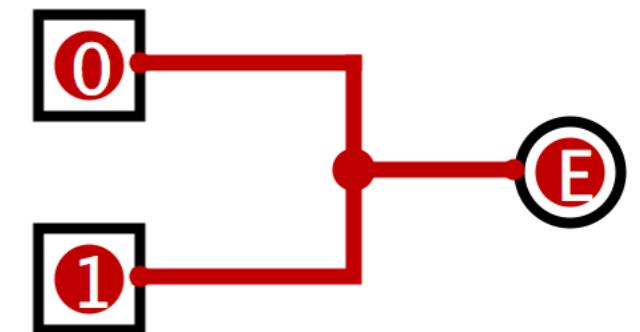
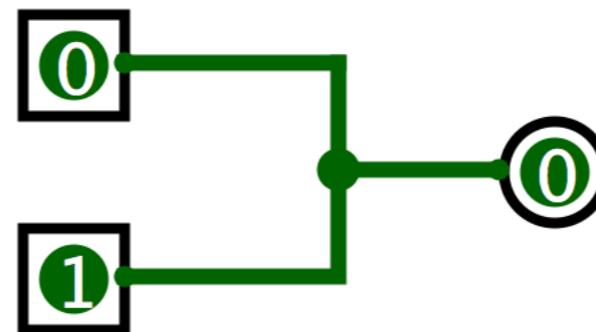
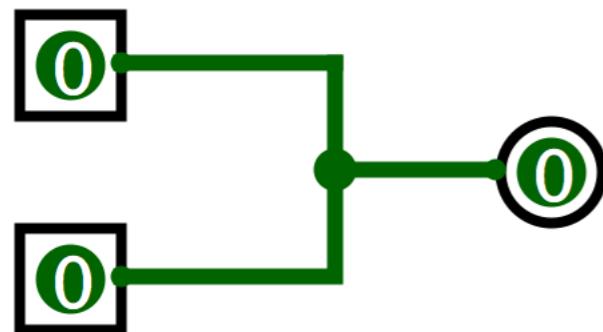
These introduce 1 delta cycle of delay each  
Not a good timing model



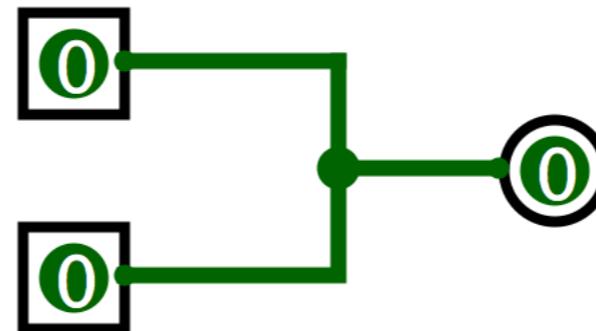
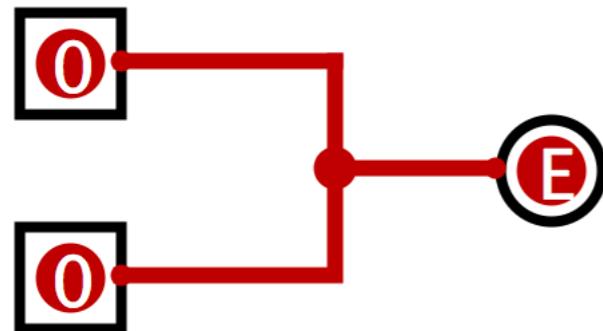
# Simbool Design



# Simbool Design

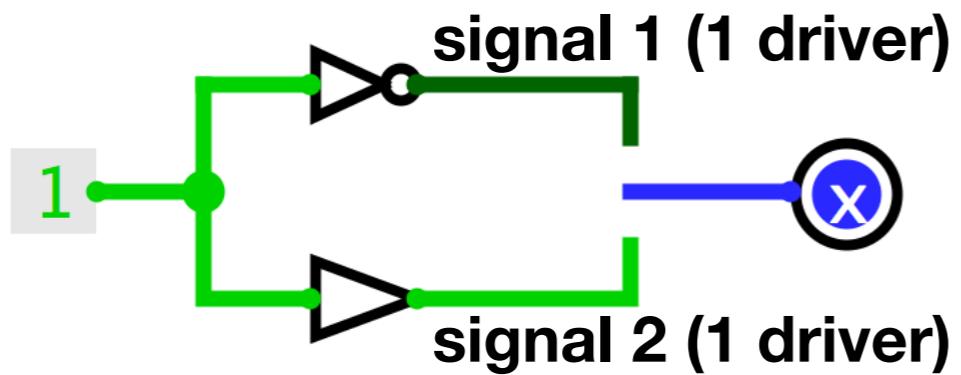


must combine (true + false)



you can't just do (**E** - true)

# Simbool Design

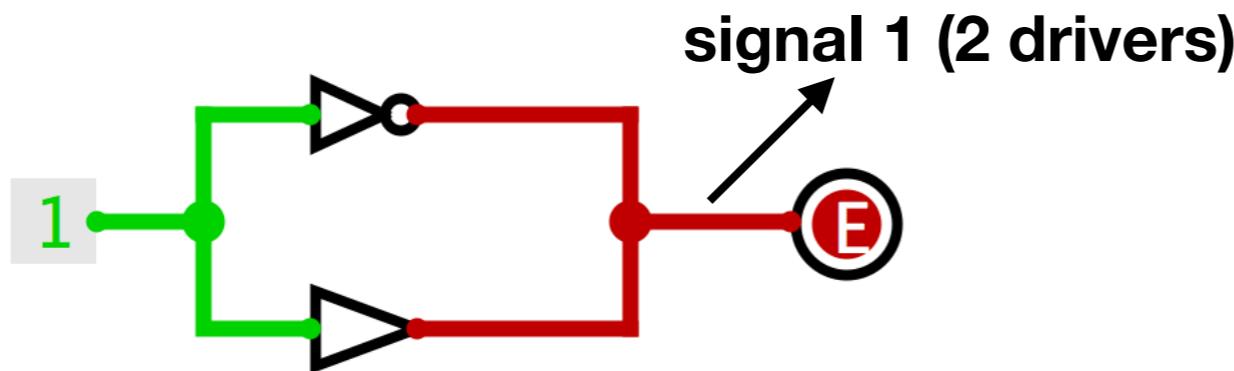
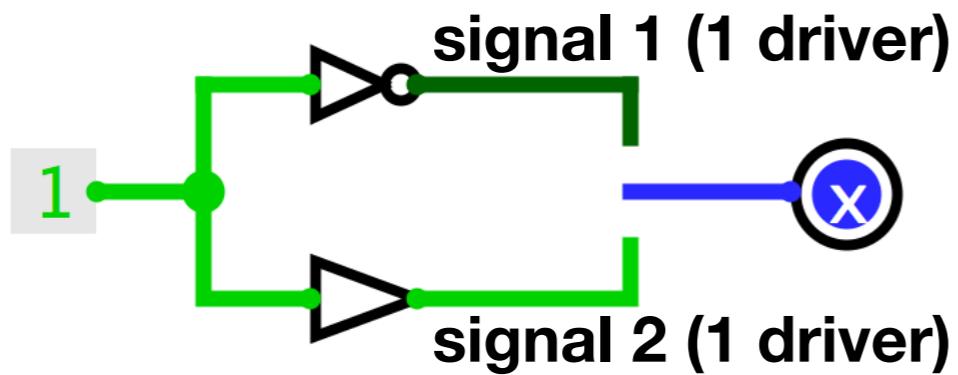


```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

# Simbool Design

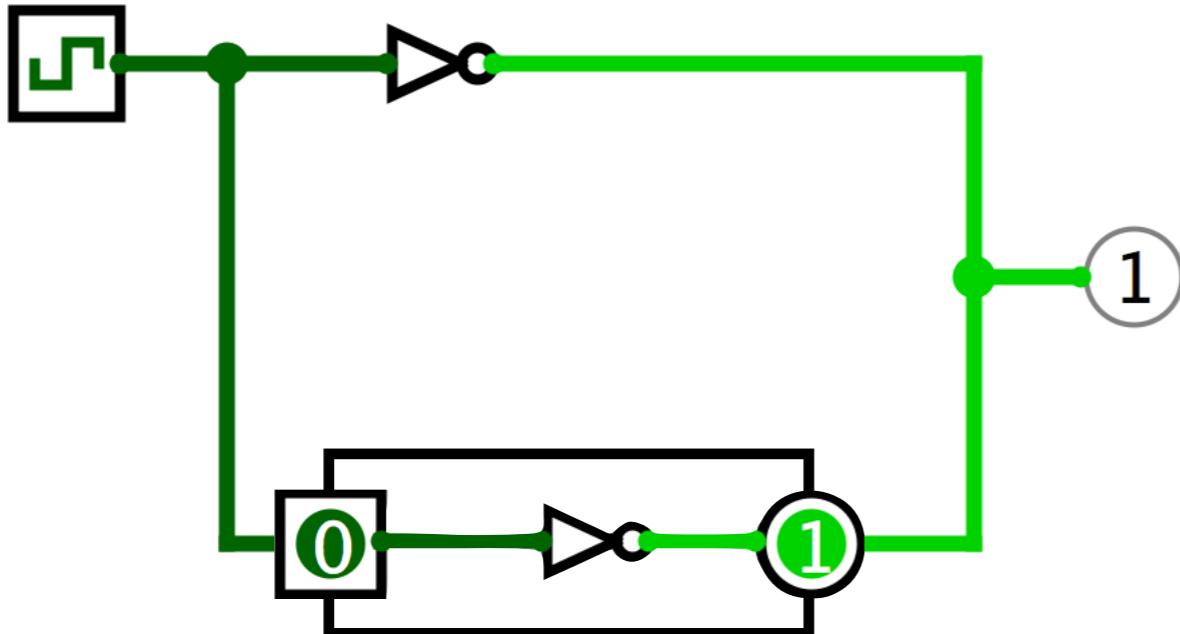


```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

# Simbool Design

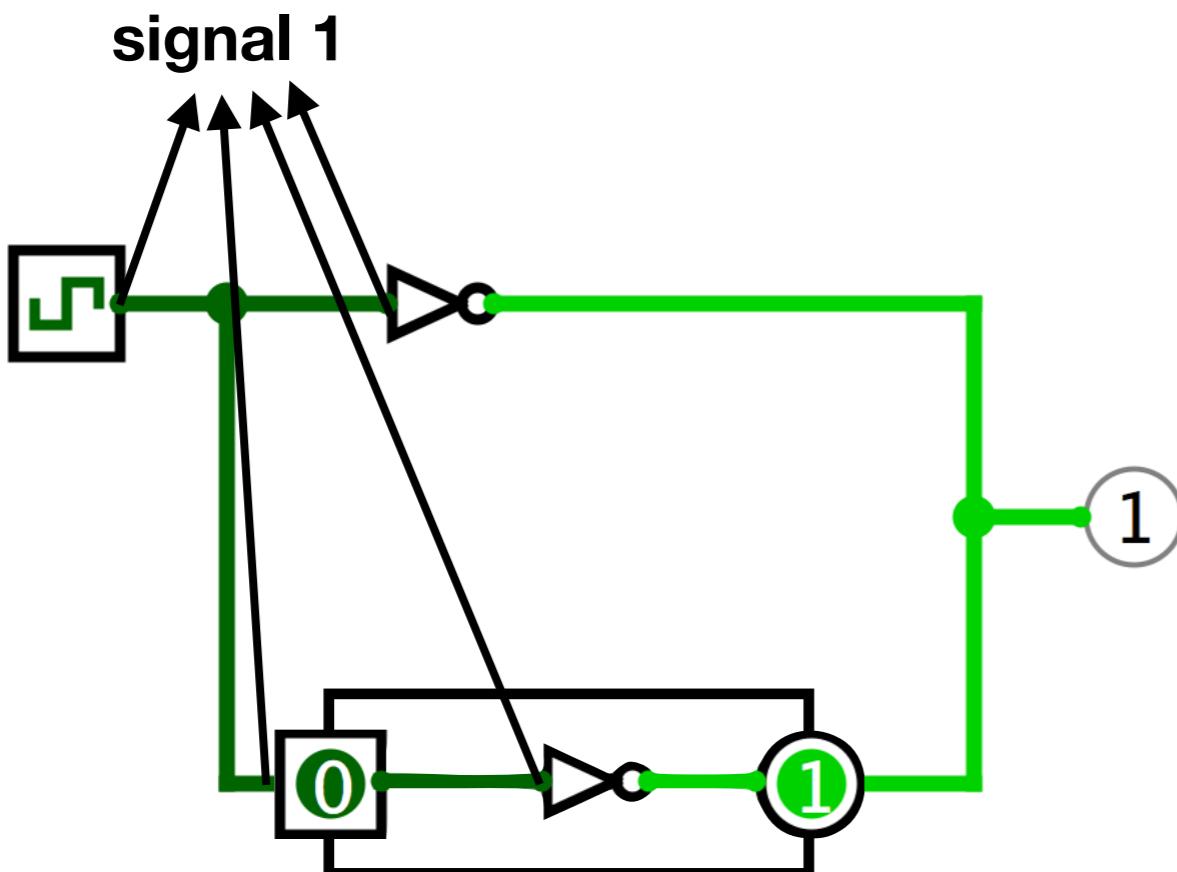


```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

# Simbool Design

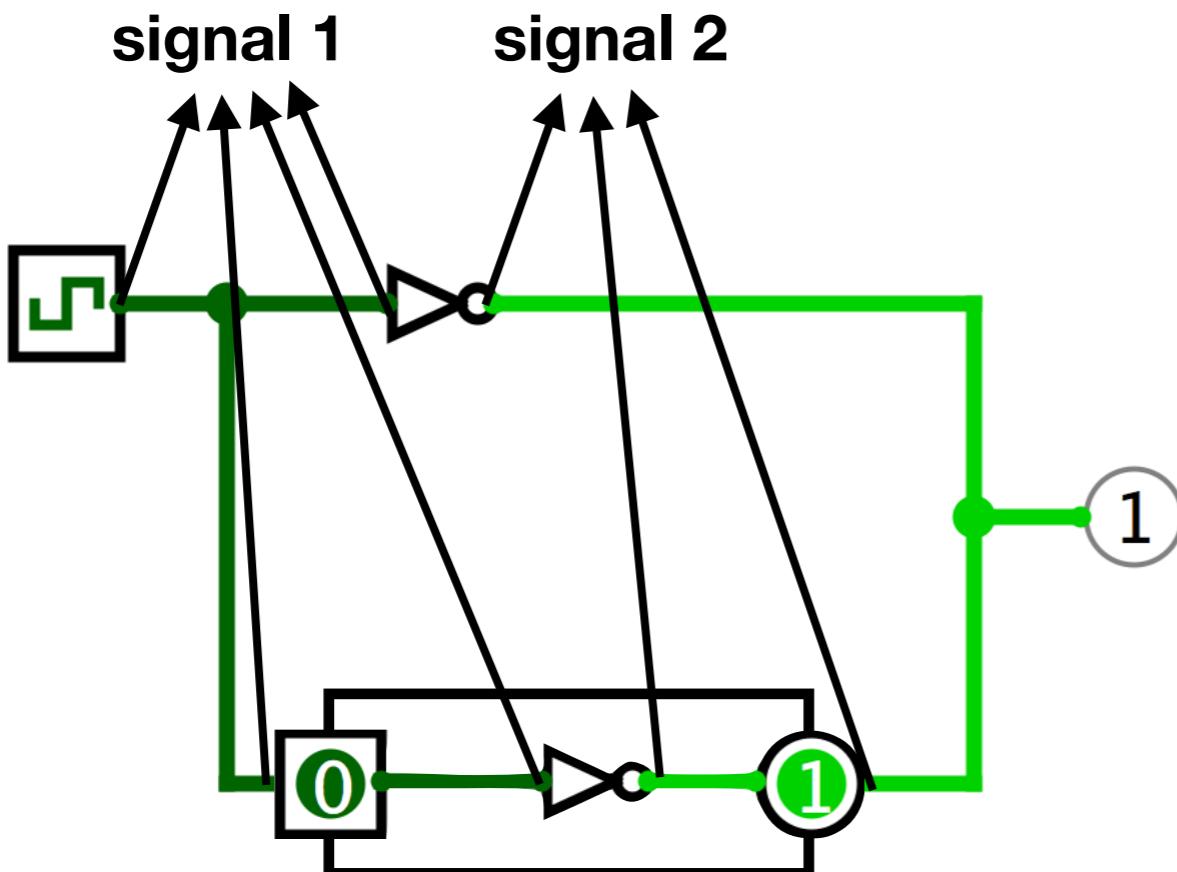


```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

# Simbool Design



```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

# Simbool Design

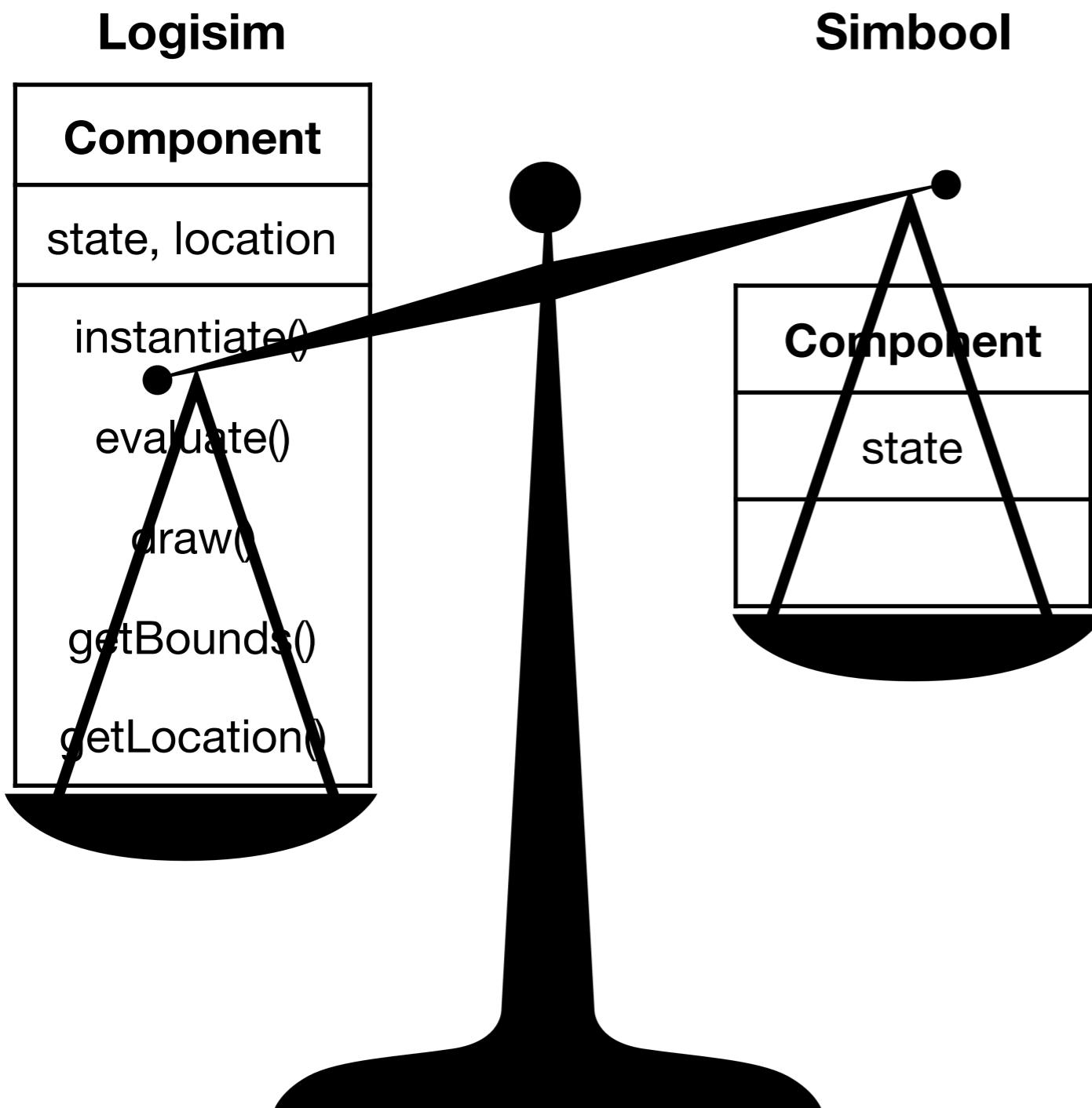
- We've arrived at our design by thinking about the computation we wanted to perform
  - Avoided architecture astronaut type decisions
  - We only abstracted what we actually needed to abstract
  - We improved the encapsulation

```
class Signal
{
    Value value;
    Value[] drivers;
    ...
}

class ComponentInstance
{
    PortInstance[] ports;
    ...
}

struct PortInstance
{
    Signal signal;
    int driver = -1;
}
```

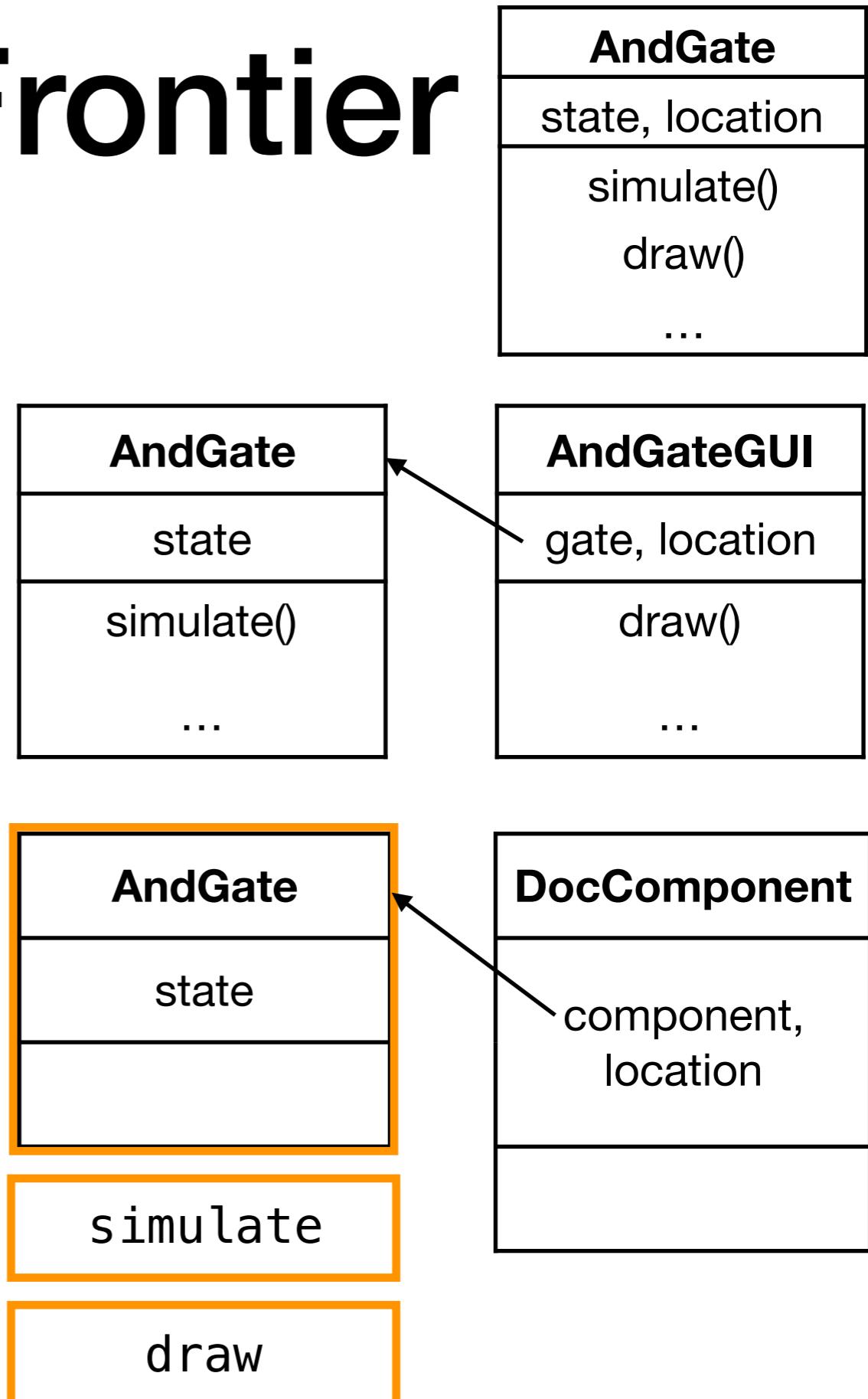
# Abstraction Frontier



- Simbool's Component classe's single responsibility: maintain the simulation state
- How do we simulate them?
- How do we draw the components?
- How do we know *where* to draw them?

# Abstraction Frontier

- Orthodox OOP Design:
  - AndGate
- Maximal encapsulation OOP:
  - AndGate
  - AndGateGUI
- Expressive style:
  - AndGate
  - simulate open method
  - draw open method



# Abstraction Frontier

- Maximal encapsulation OOP:

```
void render() {  
    ComponentGUI cg; // AndGateGUI  
    cg.draw();  
}
```

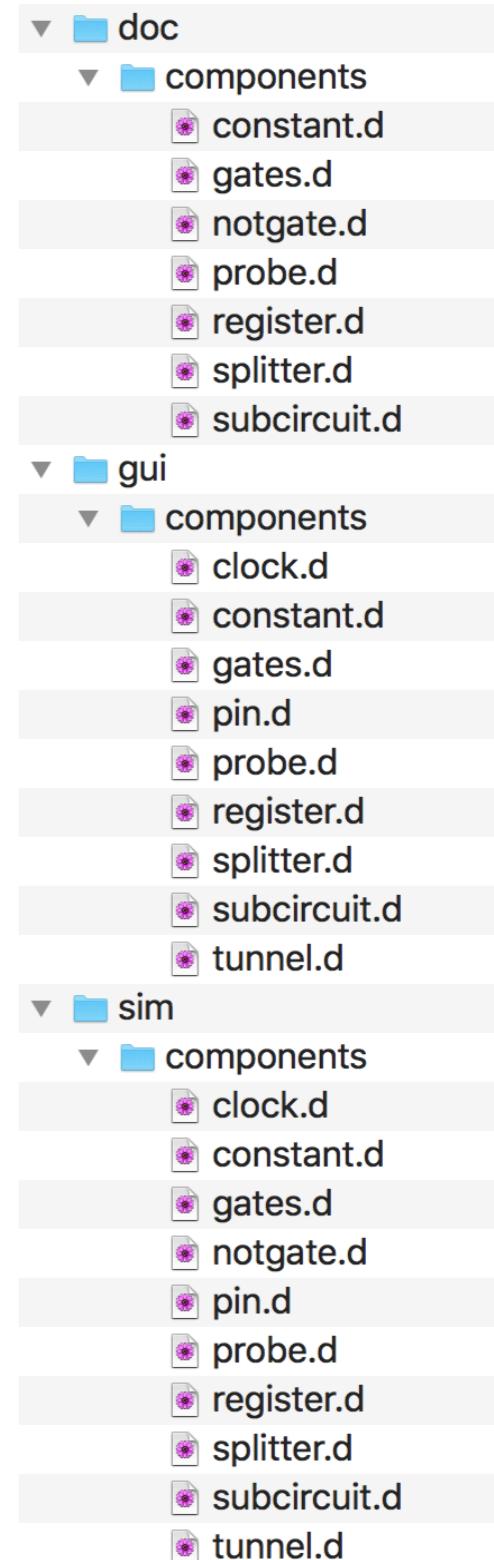
- Expressive style:

```
void draw(DocComponent, virtual!Component);
```

```
void render() {  
    DocComponent dc;  
    auto comp = dc.component  
    draw(dc, comp);  
}
```

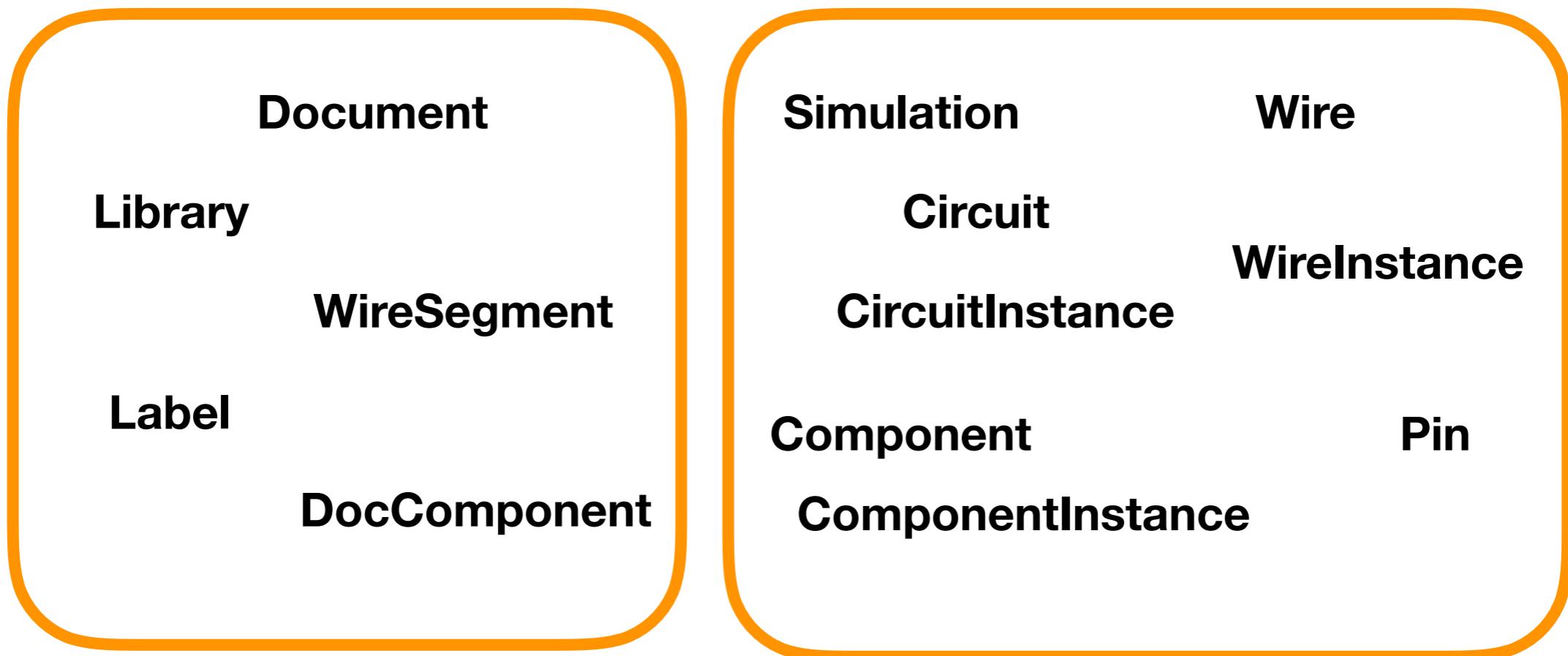
# Abstraction Frontier

- sim package only knows about the simulation
  - All you need for the actual simulation
- doc package knows about document geometry semantics (depends on sim)
  - “Do these wires join?”
- gui knows about both the simulation state (i.e. in which state to draw the components) and the document geometry (where to draw them; depends on both)



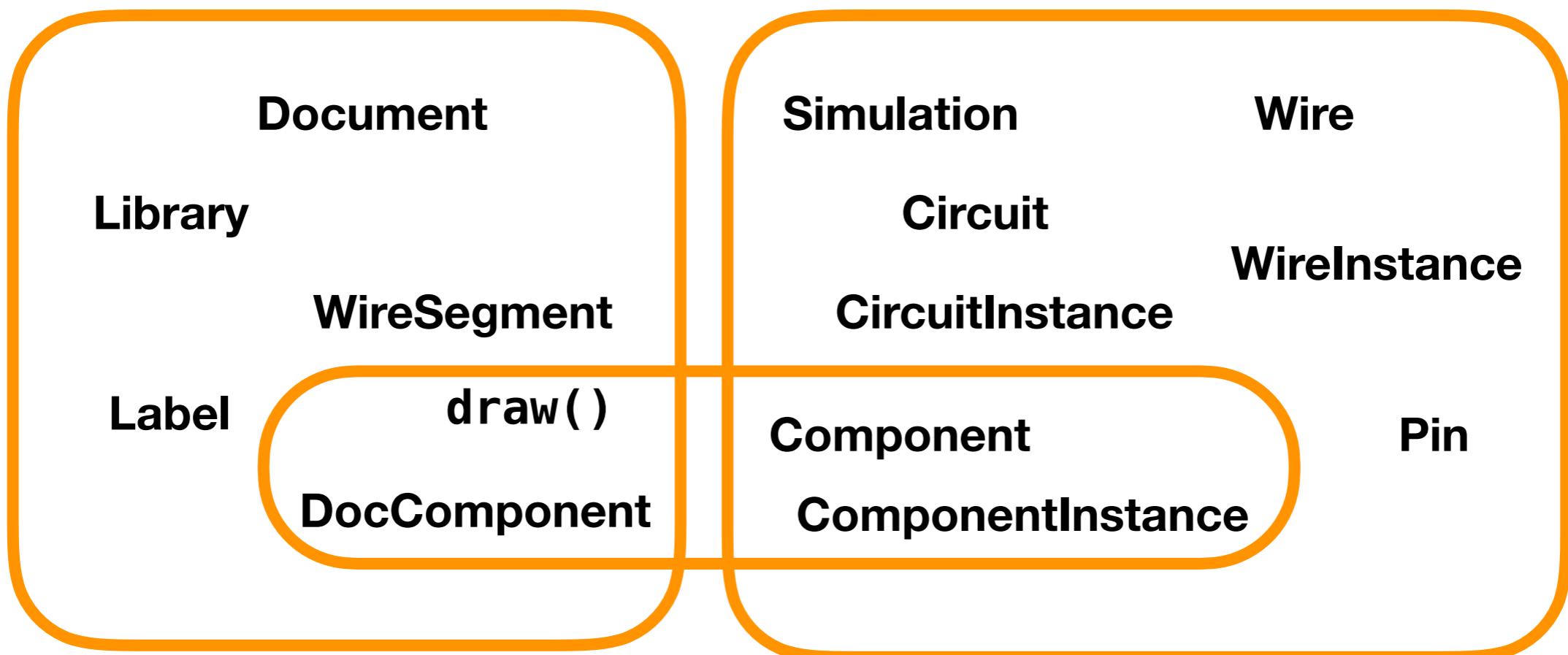
# Abstraction Frontier

- Example abstraction frontiers (not entity-oriented)

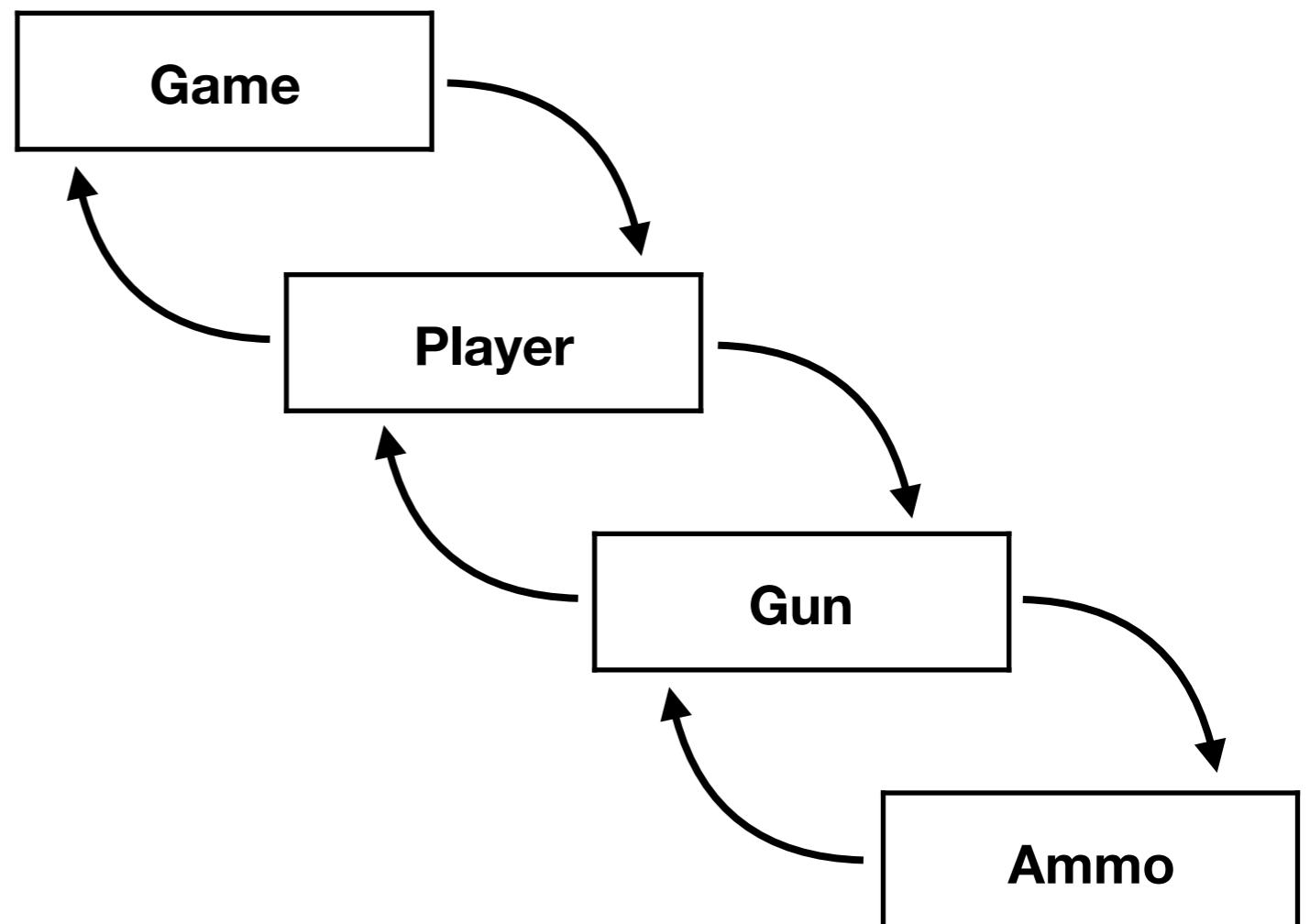


# Abstraction Frontier

- Example abstraction frontiers (not entity-oriented)

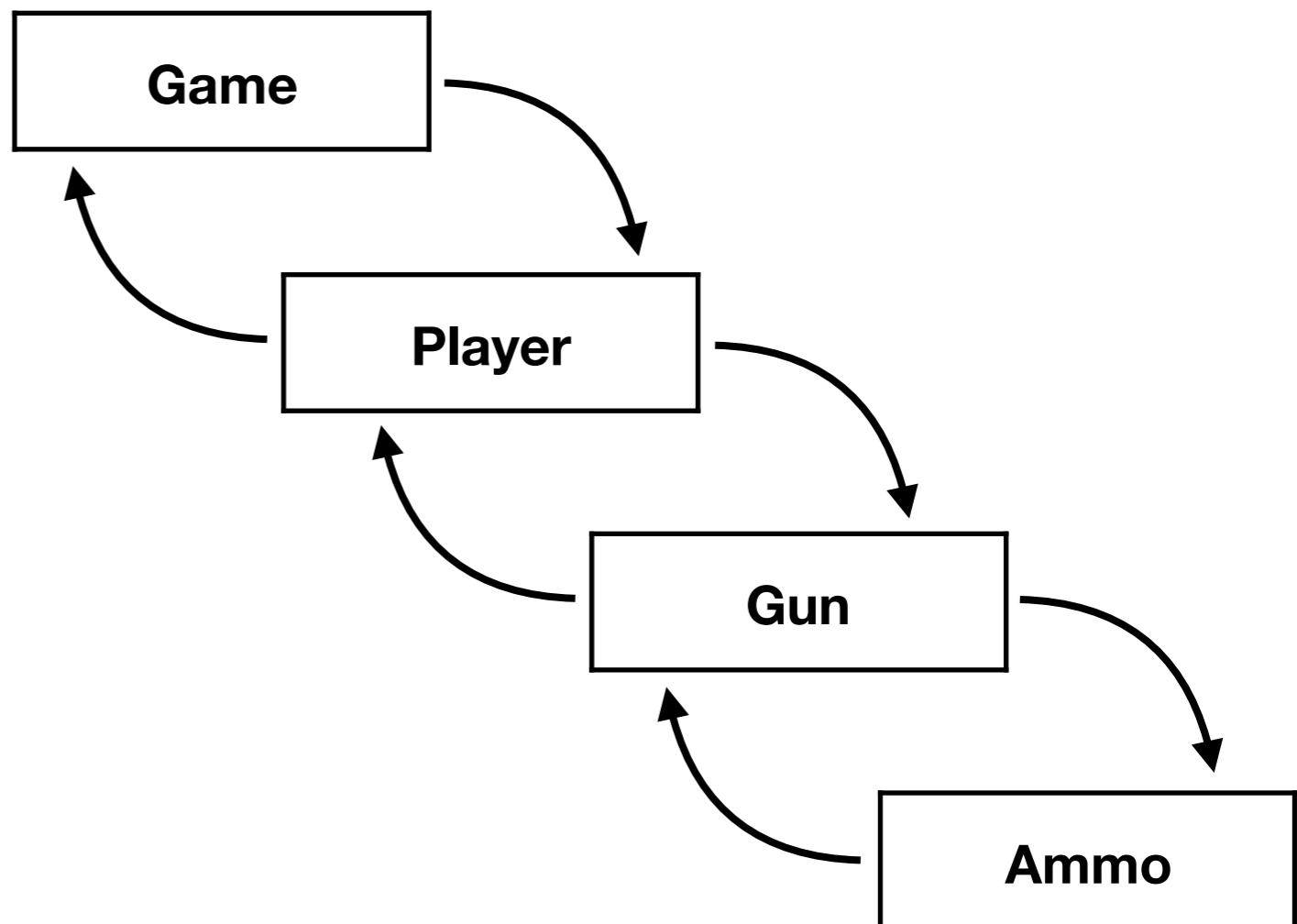


# Object Graphs



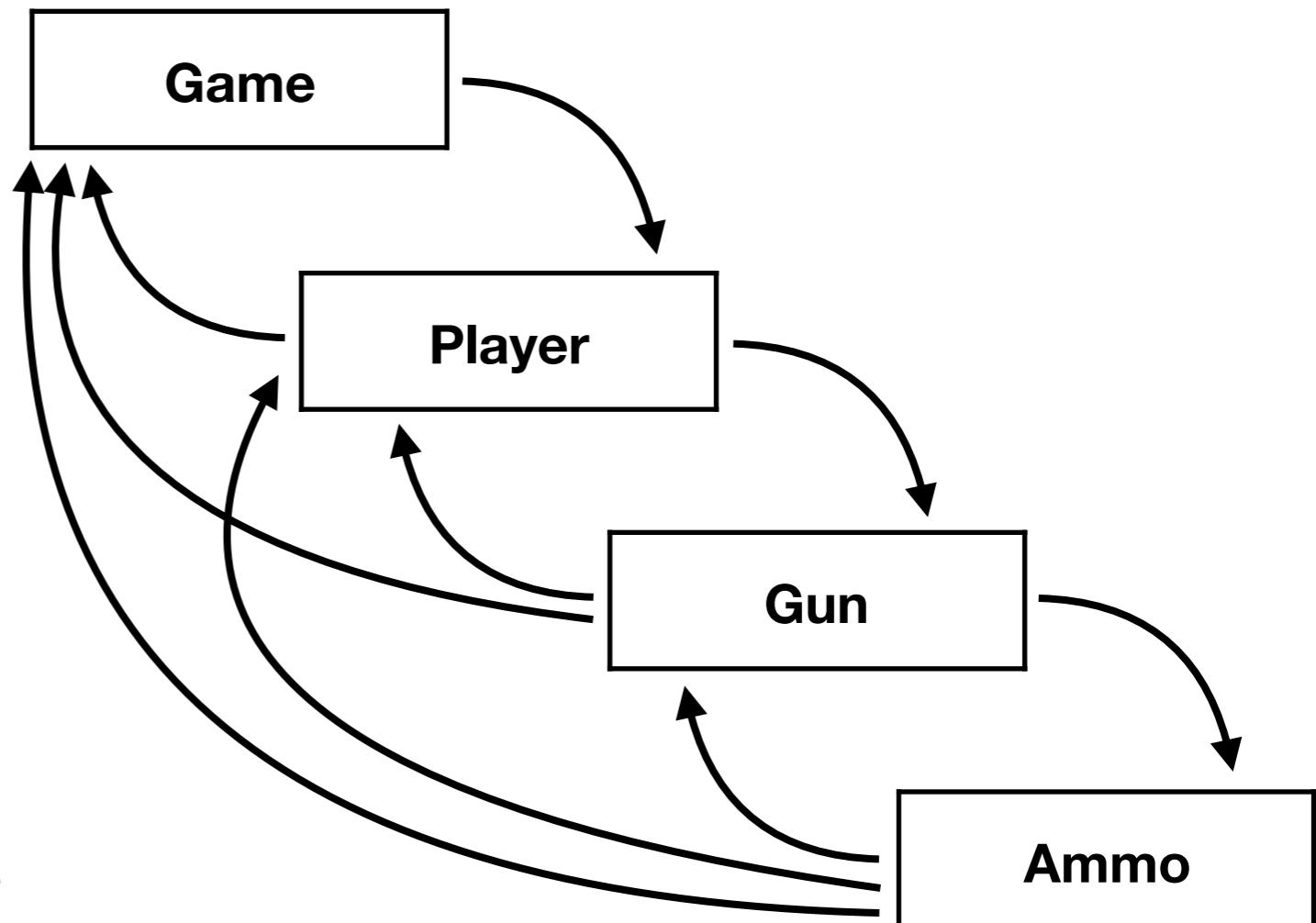
# Object Graphs

- Law of Demeter?
  - no obj
    - .getX()
    - .getY()
    - .getZ()
    - .doSomething()



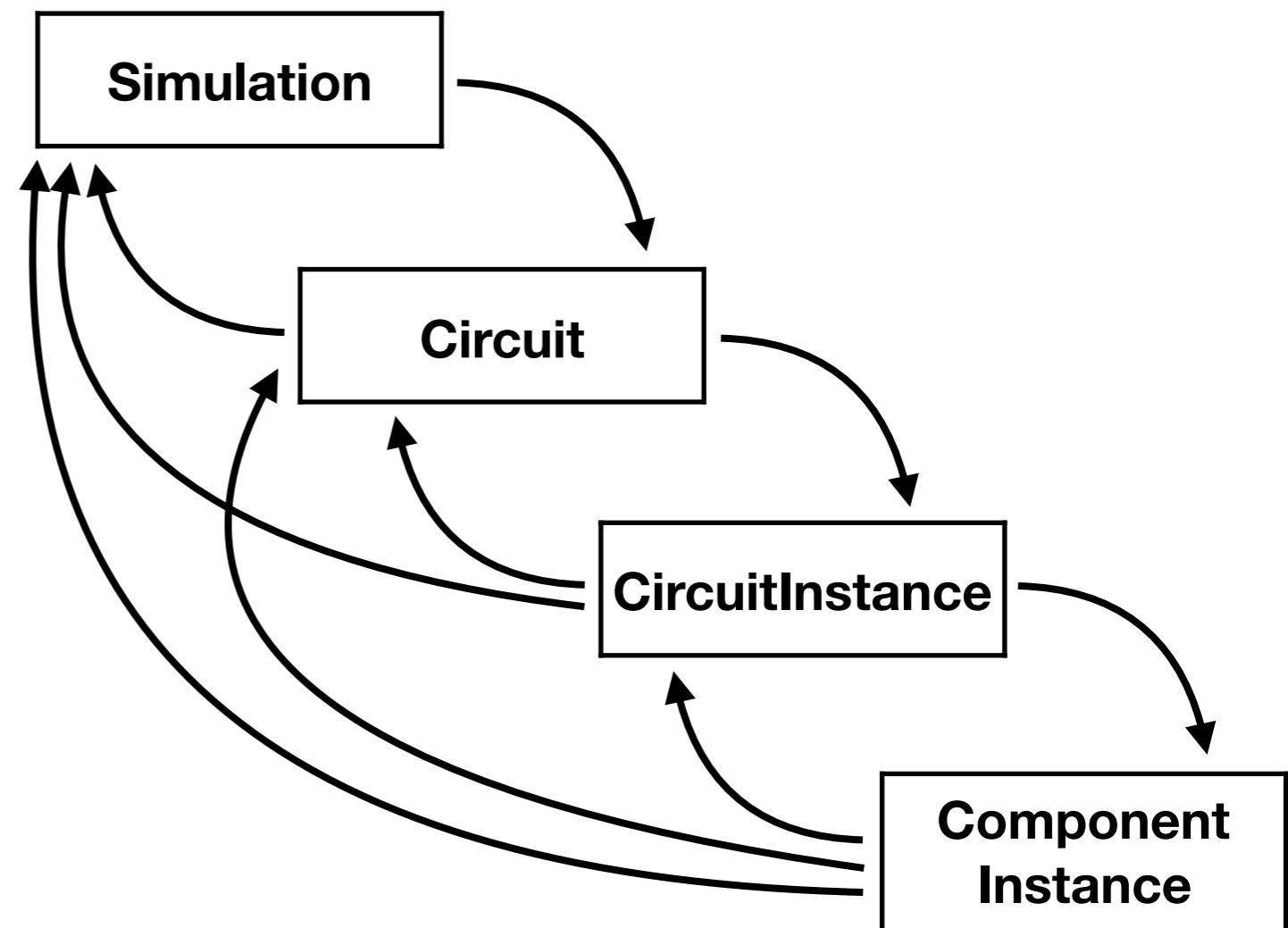
# Object Graphs

- Law of Demeter?
  - no obj
    - .getX()
    - .getY()
    - .getZ()
    - .doSomething()
- Not great for performance



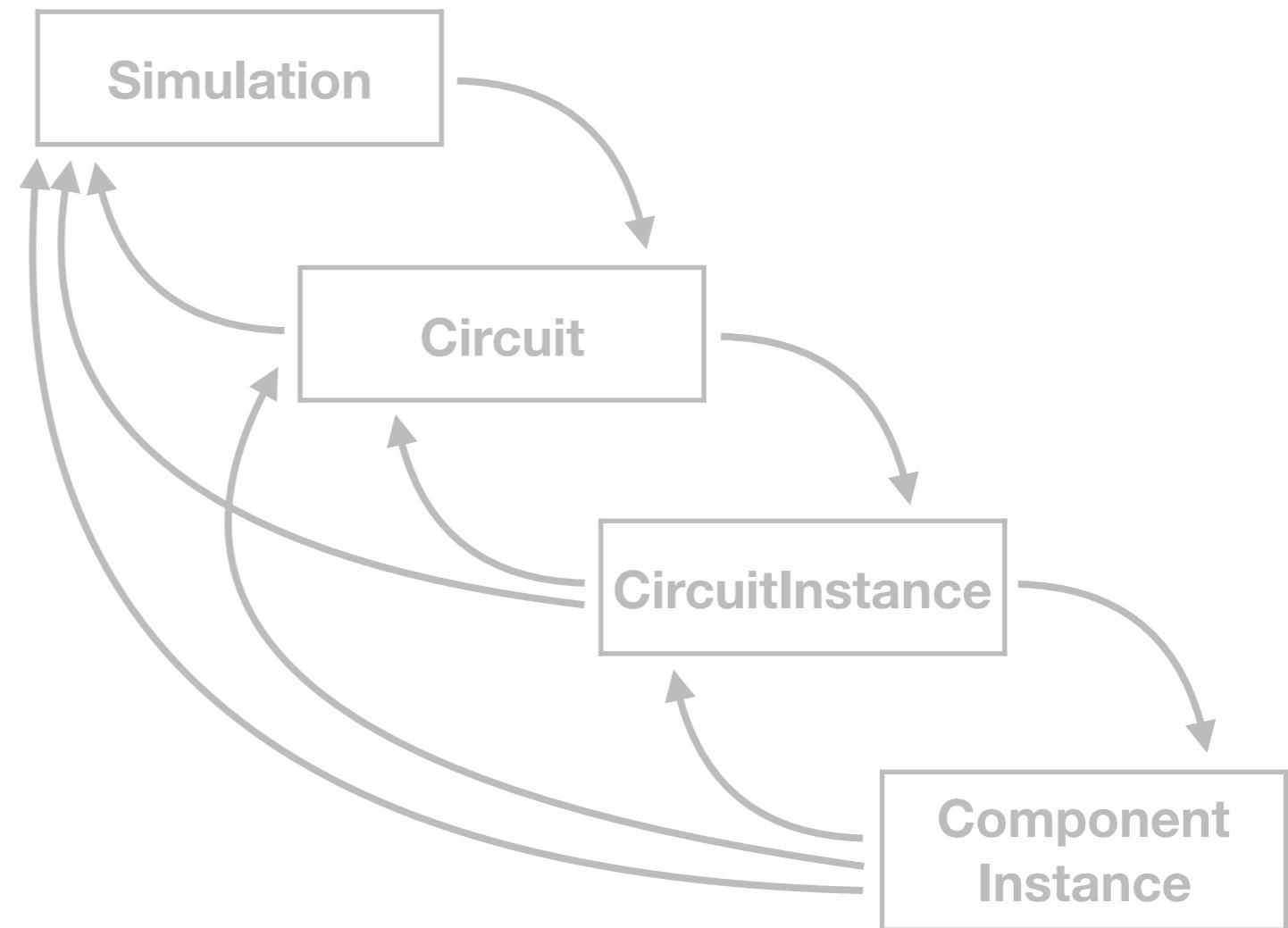
# Object Graphs

- We know which simulation we are simulating at the point of execution
- Why store redundant information in the component instance?



# Object Graphs

- We know which simulation we are simulating at the point of execution
- Why store redundant information in the component instance?



```
evaluate(Simulation, virtual!ComponentInstance);
```

# Memory Allocation

- In OOP, objects are an island of abstraction
  - They live by themselves, have references to other objects
  - Each object knows how to create itself (ctor)
    - When that's not the case, we use a special name
  - We have context that we can use at the point of allocation

```
ComponentInstance instantiate(Simulation,  
virtual!Component, CircuitInstance parentInstance);
```

# OOP Assessment

- AndGate and Register inherit from Component
- OOP mindset:
  - Component provides an abstract simulation method
  - AndGate knows how an AND gate actually works
  - Nobody else has to know it
    - That's the point of messaging! The object interprets the message!

# OOP Assessment

- Truth:
  - The AndGate *type* indicates what should be simulated
  - Two simulation services:
    - The interpreter (evaluate open method)
    - The JIT
  - Conclusion: OOP perspective wasn't very illuminating

# OOP Assessment

- Keeping nimble despite uncertain requirements
  - Use minimal abstractions
  - `size_t`
  - Accessor methods vs const reference
  - alias `ValueRef` = ...
- Validate the design first. Only abstract what would be a pain to change later

# OOP Assessment

- Why is OOP appealing?
  - Main header syndrome
  - GUI classes

# Testing

- Empire refactoring experience
  - Not OOP (yet easier to work with than many other code bases!)
  - No tests
    - Michael Feathers: legacy code is code without tests
  - Create unit tests for everything?
    - Too much work!
    - Not even clear what the *exact* game rules were
  - Smarter alternative

# Testing

- Empire refactoring experience
  - Set up 1+ games (different seeds), with no UI
  - Record all game messages and the map state to a buffer
  - Hash the buffer (SHA-1)
  - Check that the code about to be refactored is covered by the test, using the `-cov` option
  - Refactor
  - Run the test again; compare the hash
  - Worked amazingly well (fast and effective)

# Testing

- Simulation, document, GUI are separate packages
- Orthodox advice would be to have separate unit tests
- Started out with simulation unittest blocks
  - The tests were very verbose
- Replaced those with integrated tests
  - Design circuit in GUI; save to file; read file in test; simulate; assert the desired property
  - Easier to visually debug wrong results
  - Can compare results with Logisim

# Conclusion

- Don't assume two classes always have to hide all of their internals from each other (orthodox OOP). There may be better lines along which to define abstraction frontiers, possibly cutting across entities.
- Consider thinking of the computation first, and only then what abstractions better support it
- Program to the public interface; don't make it a member
- A foo method on class C only works for C objects and subclasses
  - A template function works for any compatible type
  - An open method is the runtime counterpart
  - Dbl ❤️ open methods. Sitting in a tree. KISSing

# References

1. Meyers, Scott. How Non-Member Functions Improve Encapsulation. <<http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197>>
2. Marques, Luís. A defense of so-called anemic domain models. <<http://www.coredump.xyz/meetup%20-%20anemic%20domain%20models.pdf>>
3. Core Dump Podcast, episode 1.  
<<http://www.coredump.xyz/1>>
4. Leroy, Jean-Louis. openmethods.d.  
<<https://code.dlang.org/packages/openmethods>>