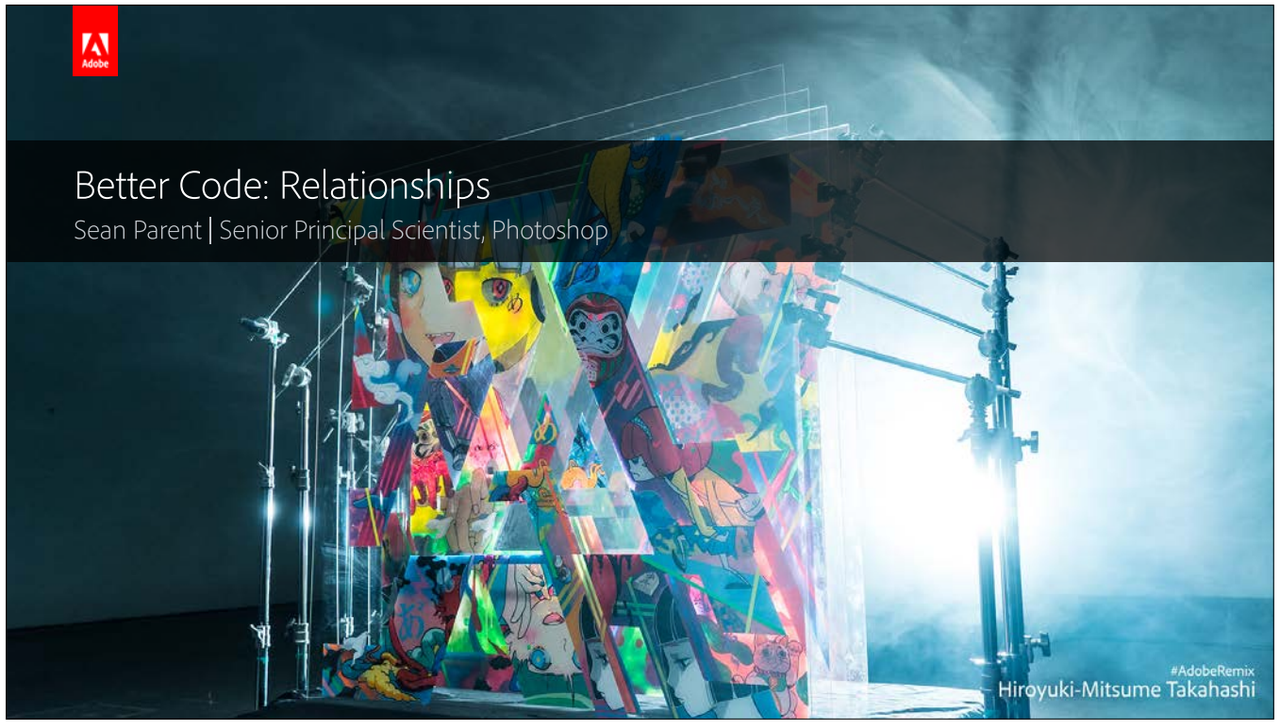
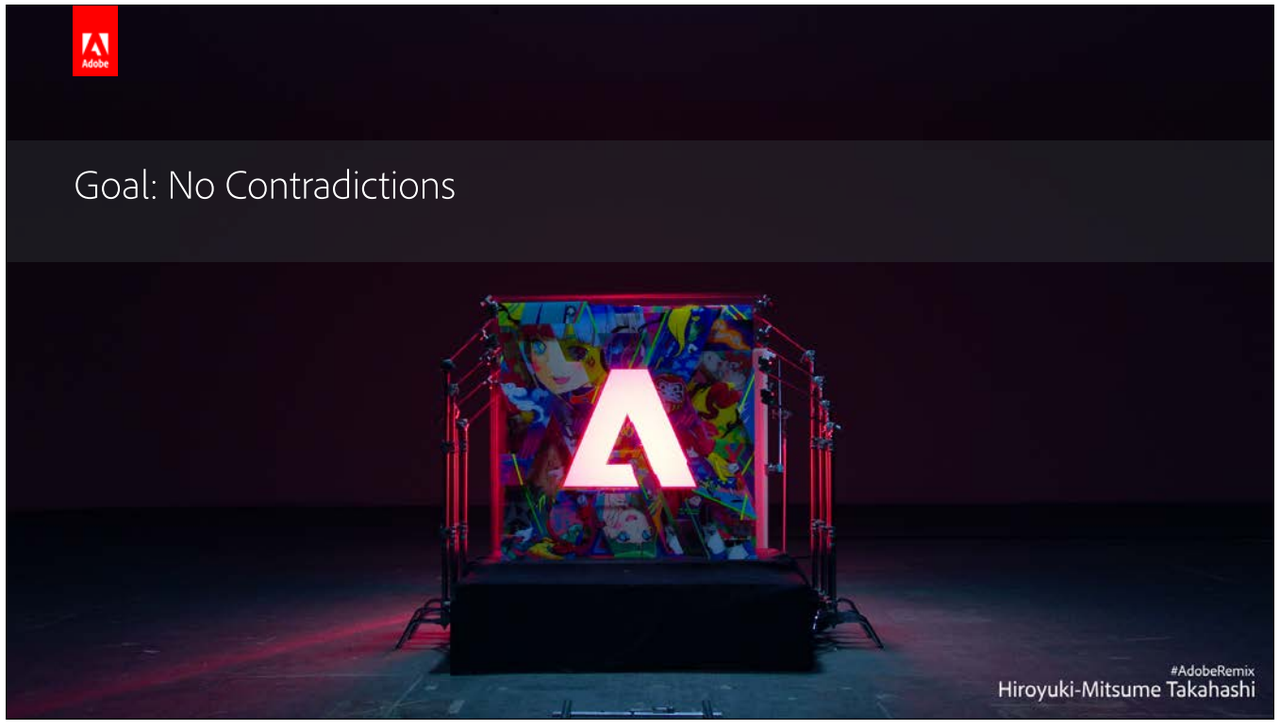
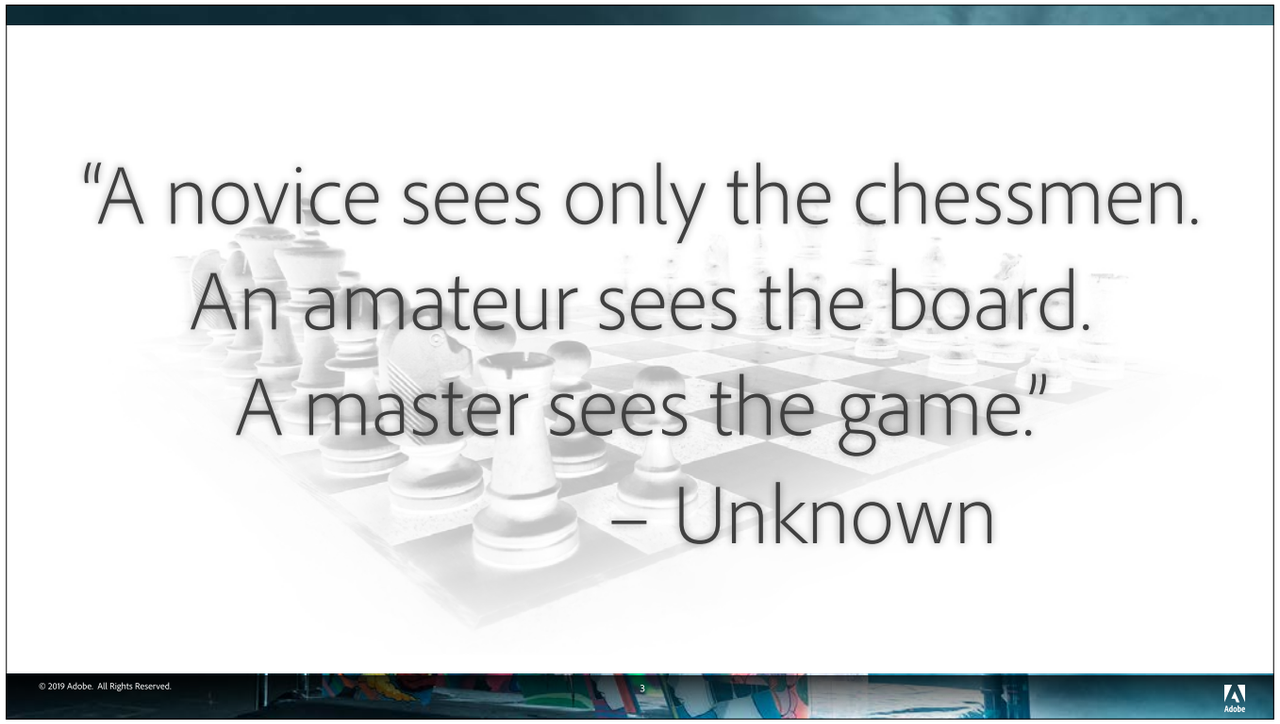

Better Code: Relationships
Sean Parent | Senior Principal Scientist, Photoshop
#AdobeRemix
Hiroyuki-Mitsume Takahashi

Goal: No Contradictions





If anyone can find this - wonderful, otherwise I may lay claim
What is meant by “sees” - clearly it isn’t optics
I contend it is relationships.

"Computer scientists are bad at relationships."

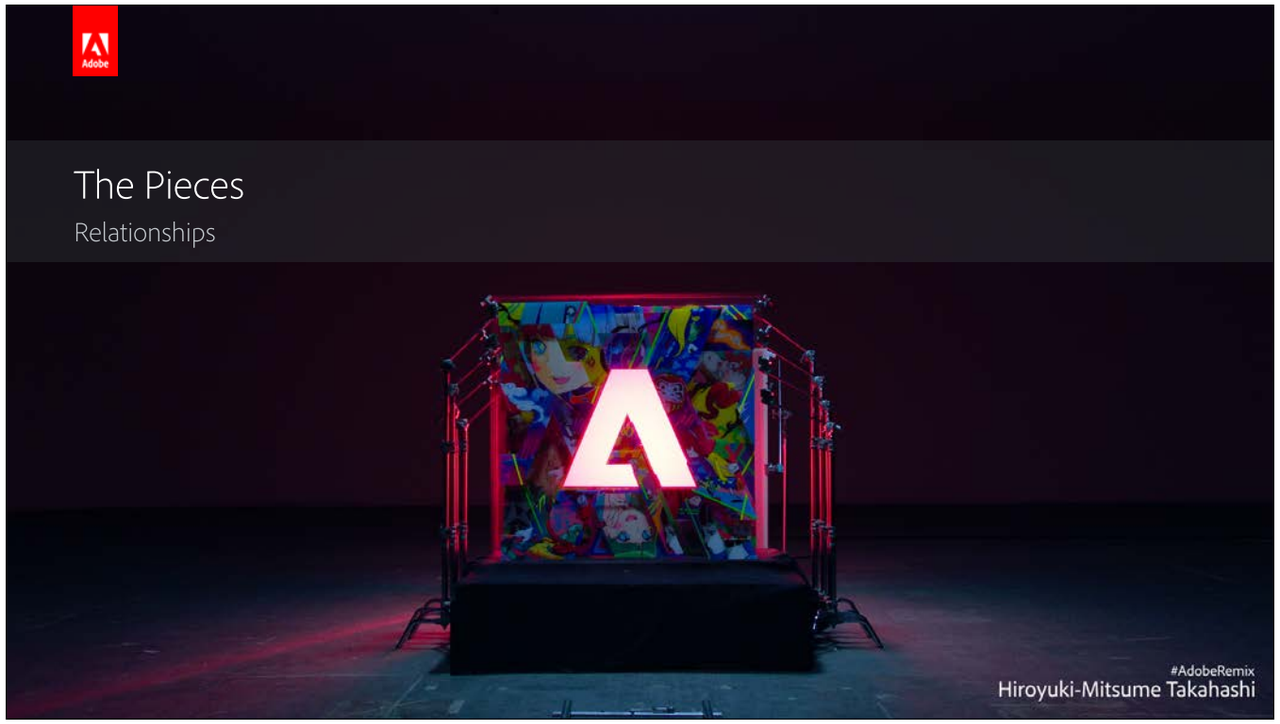
– Me

Adobe

Which brings me to my quote -
We spend our time on types and functions, but the really important part is what happens between them.

The Pieces

Relationships



Relations in Math

- A *relation* is a set of ordered pairs mapping entities from a *domain* to a *range*

- Distinct from a *function* in that the first *entity* does not uniquely determine the second
- A *relationship* is the way two entities are connected

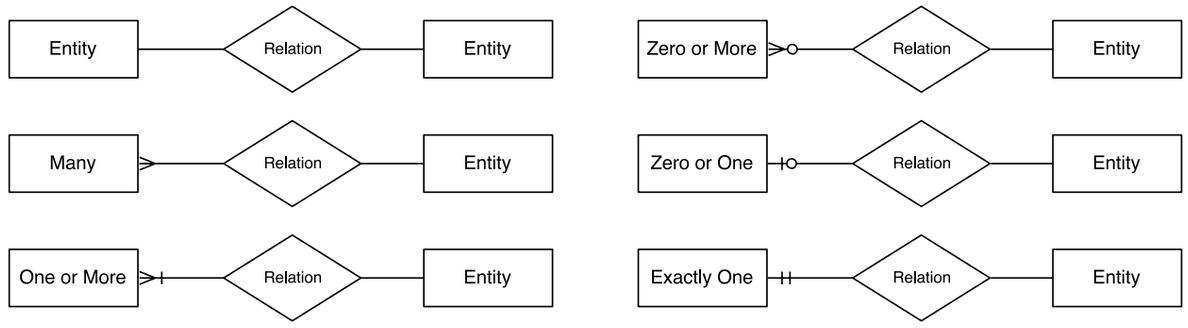
$$\{(x_0, y_0), (x_1, y_1), (x_2, y_2), ...\}$$

Relationship need not be an witnessed by an object - it just is. No code required.
Give examples:

element 2 is after element 1 in an array
you are married to your wife

Cardinality and Modality of Relationships

- *Cardinality* is the maximum number of times an entity can be associated with objects in the related entity
- *Modality* is the minimum number



Crows foot notation -
Commonly used in database system
Minor tip: Borrow terminology!

Relationships

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation

Relationships

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

Relationships

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

- John is married to Jane

Relationships

- A relationship implies a *corresponding predicate* that tests if a pair exists in the relation
  - If it is true, the relationship is *satisfied* or *holds*

- John is married to Jane
- Is John married to Jane?

Constraints

- A *constraint* is are relationship which *must* be satisfied

Constraints

- A *constraint* is are relationship which *must* be satisfied
  - For another relationship to be satisfied

Constraints

- A *constraint* is are relationship which *must* be satisfied
  - For another relationship to be satisfied

- The denominator must not be 0 for the result of division to be defined

Implication

$$a \Rightarrow b$$

(a implies *b)*

A constraint is a form of an implication relation.

Implication

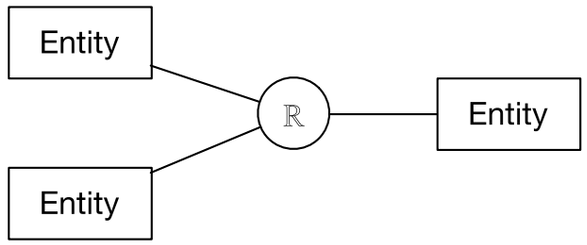
$$a \Rightarrow b$$

(a implies *b)*

a	b	a ⇒ b
0	0	1
0	1	1
1	0	0
1	1	1

A simple, but incomplete, notation

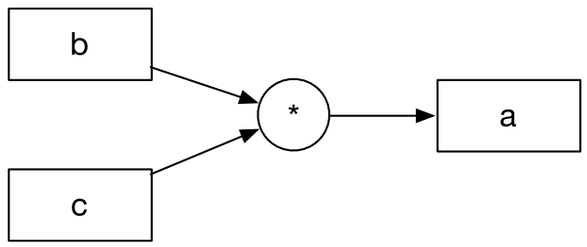
- Entities are represented with a rectangle, and relationships with a circle
- This forms a *bipartite* graph



This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

A simple, but incomplete, notation

- Implication is represented with directional edges



- This is shorthand for

$$(a = bc) \wedge (b = x) \wedge (c = y) \implies a = xy$$

i.e. This is how a function can be presented arguments -> f -> result.

Relationships

The address of an object is a relationship between the object and it’s memory space

[This slide is a little out of place, bending into structures? Diagram/animation for sever/maintained”

Relationships

- As soon as we have two entities we have implicit relationships

Relationships

- As soon as we have two entities we have implicit relationships
  - A memory space is an object

Relationships

- As soon as we have two entities we have implicit relationships
  - A memory space is an object
- When an object is copied or move, any relationship that object was involved in is either *severed* or *maintained*

Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
  - When a witnessed relationship is copied, the relationship is either maintained, severed, or *invalidated*
  - Destructing, or moving, an object in the witnessed relationship may sever or invalidate the relationship
  - We may choose not to implement copy for relationships

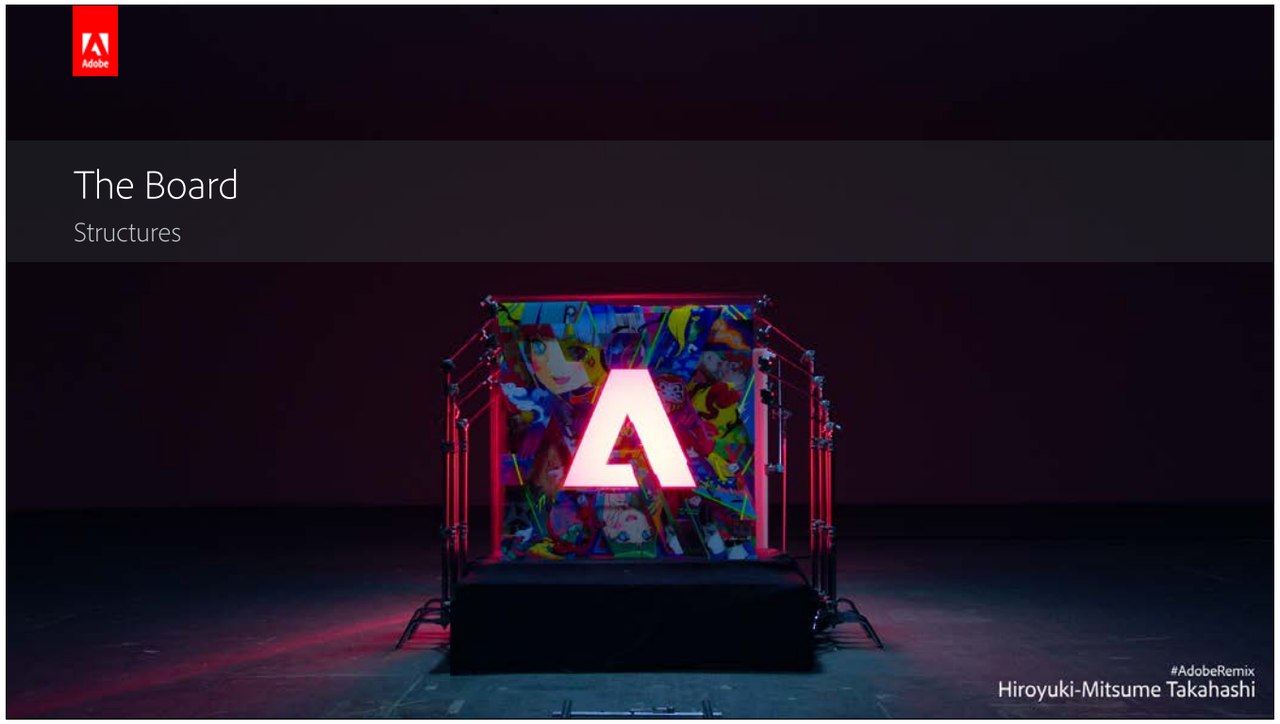
Use wedding band as example

Note that we are not talking about inbound relationships

Use weak_ptr and vector iterator for sever and invalidate examples

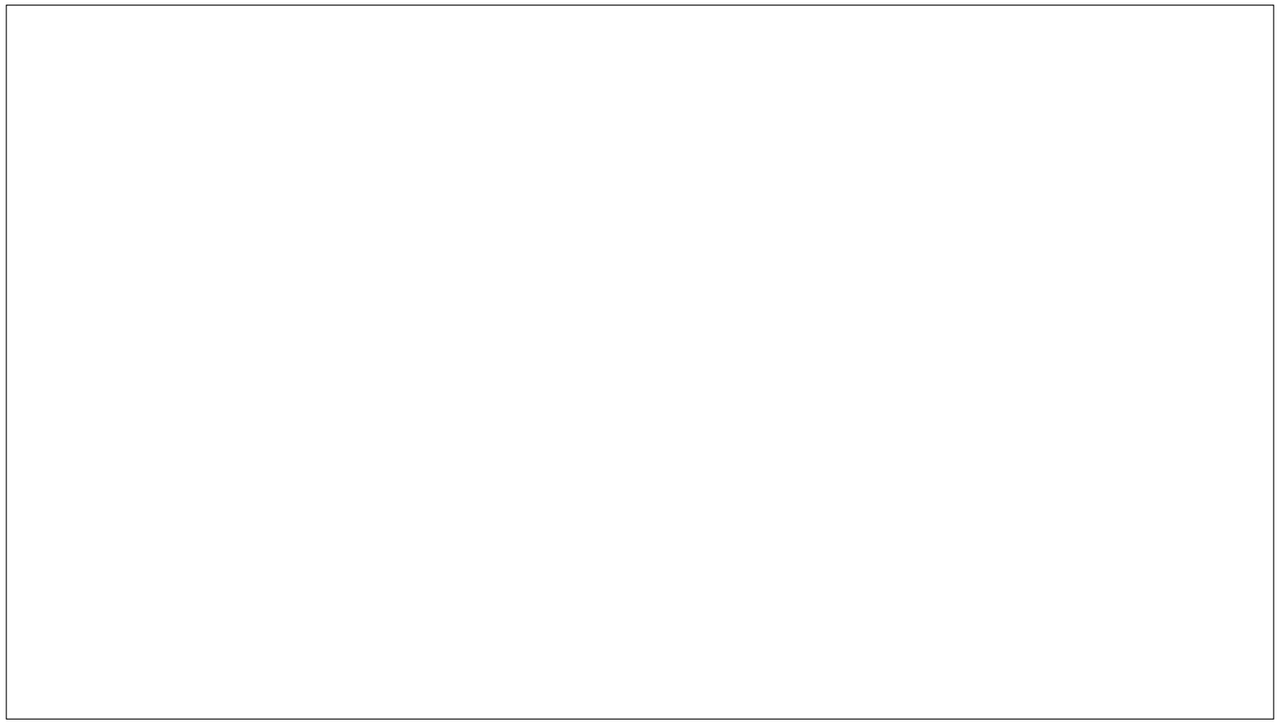
The Board

Structures

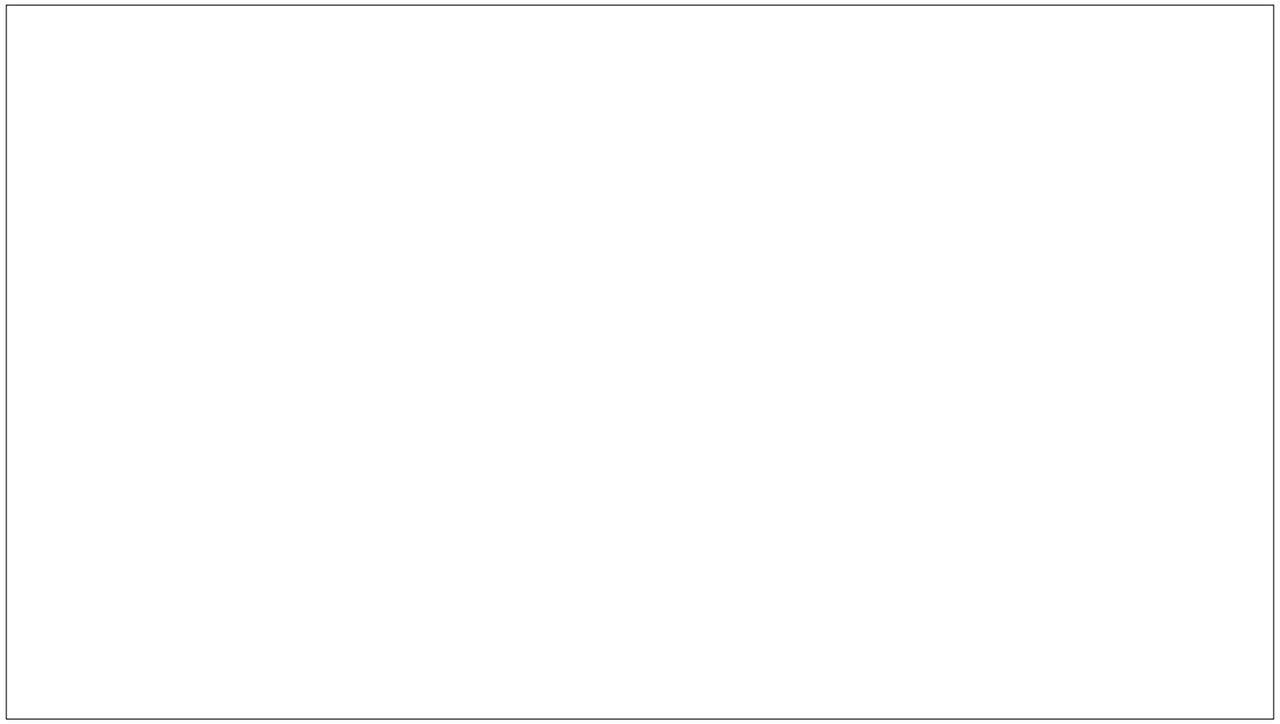


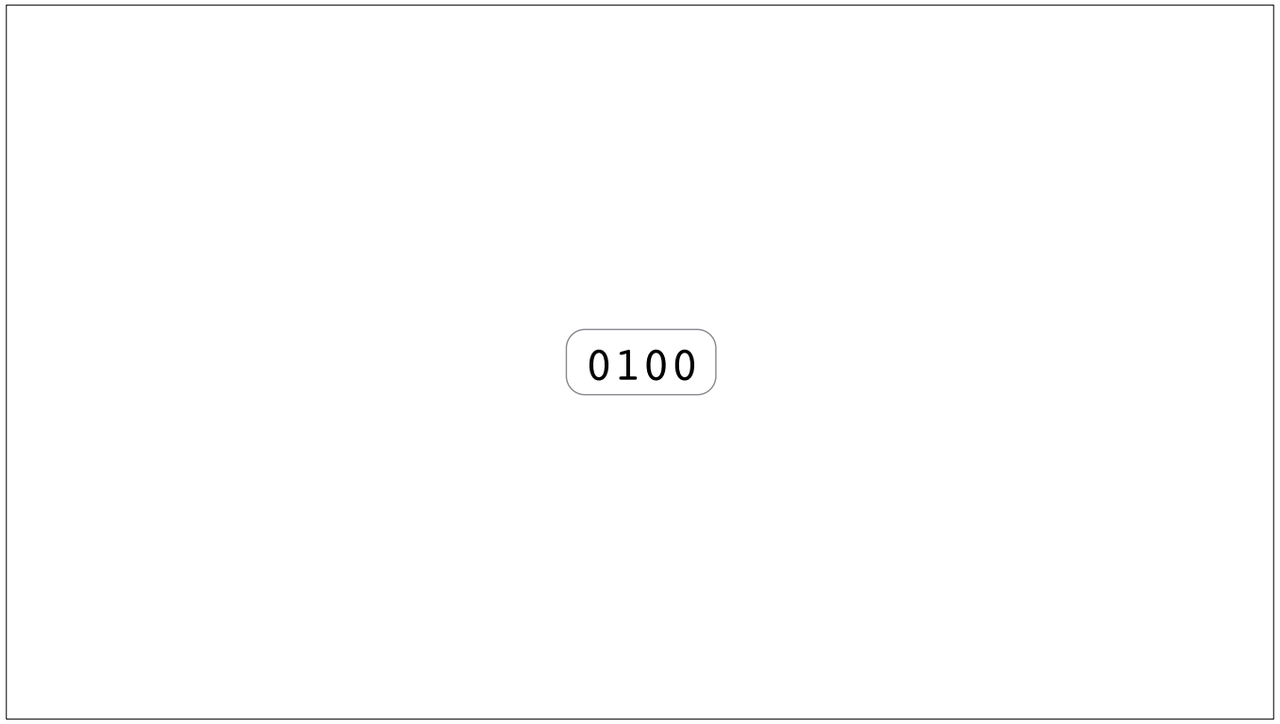
A *structure* on a set consists of additional entities that, in some manner, relate to the set, endowing the collection with meaning or significance.

Let us look at the our most basic structure -

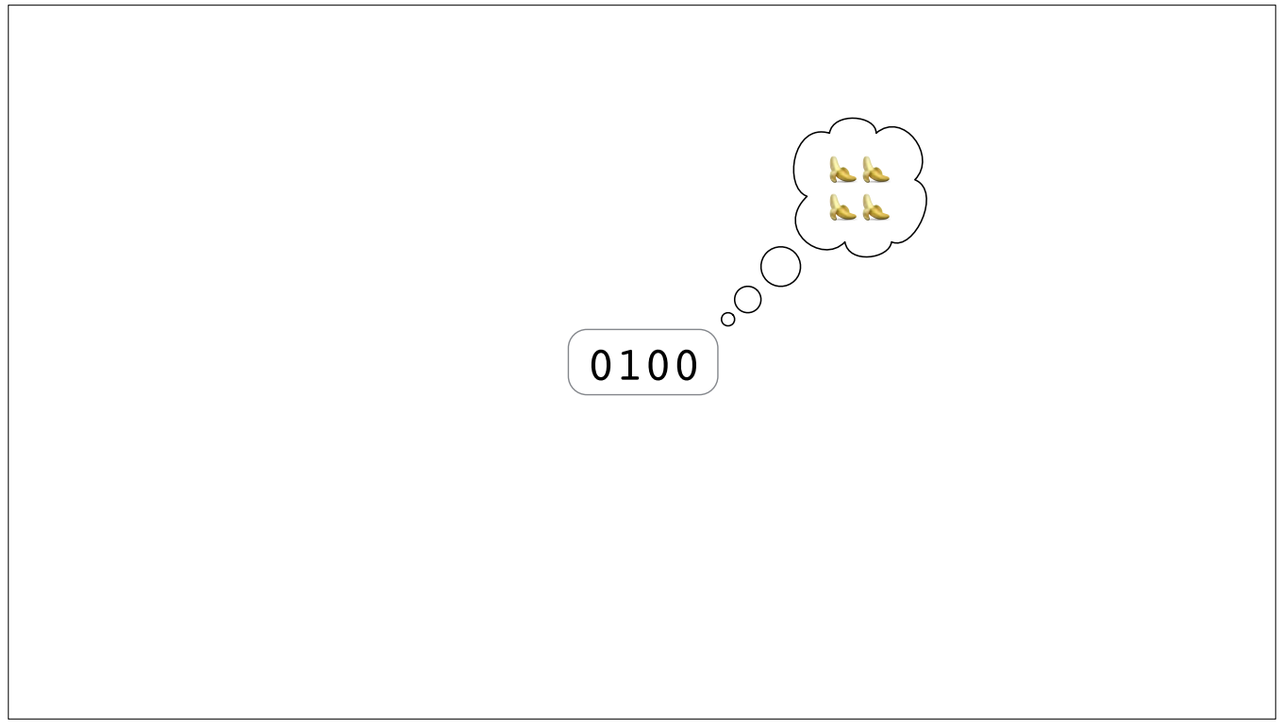


Dimensionless space - not empty, dimensionless. There isn’t even time.

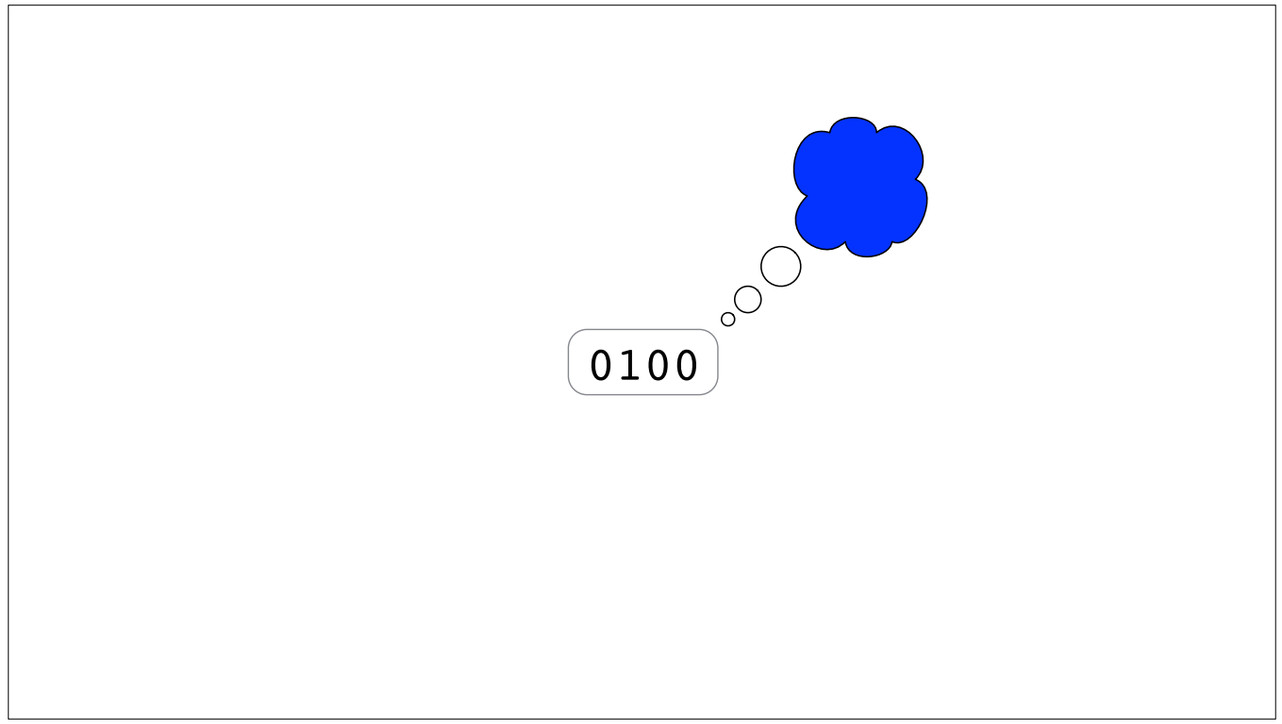




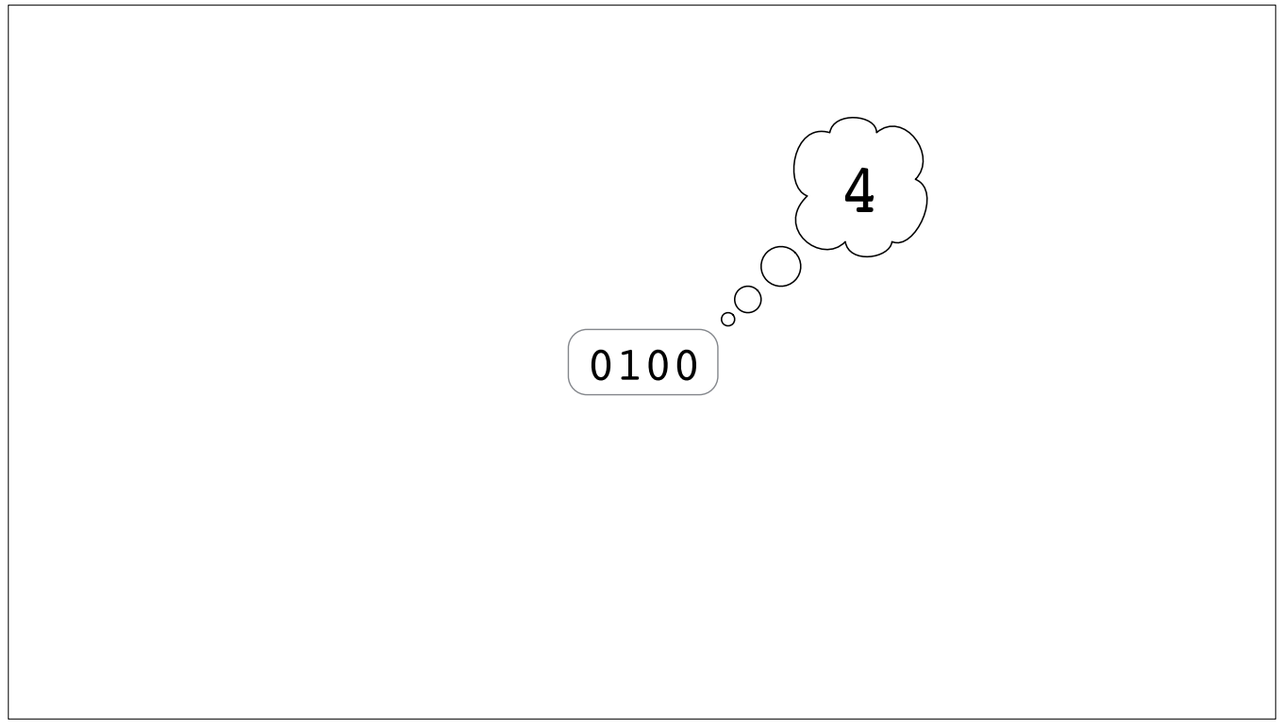
Into this void, we introduce an object to represent the value of 4.
It could be 4 cow, 4 cats, or just an abstract “4”



Into this void, we introduce an object to represent the value of 4.
It could be 4 bananas, the color blue, or just an abstract “4”



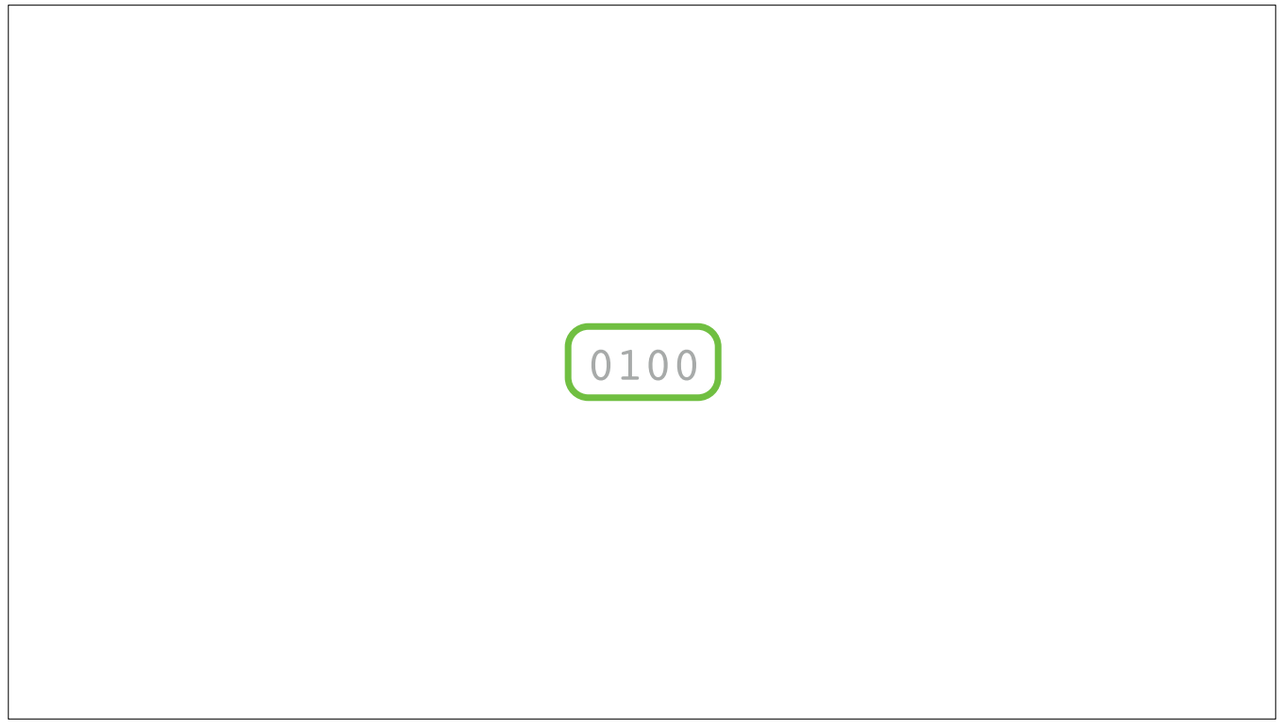
Into this void, we introduce an object to represent the value of 4.



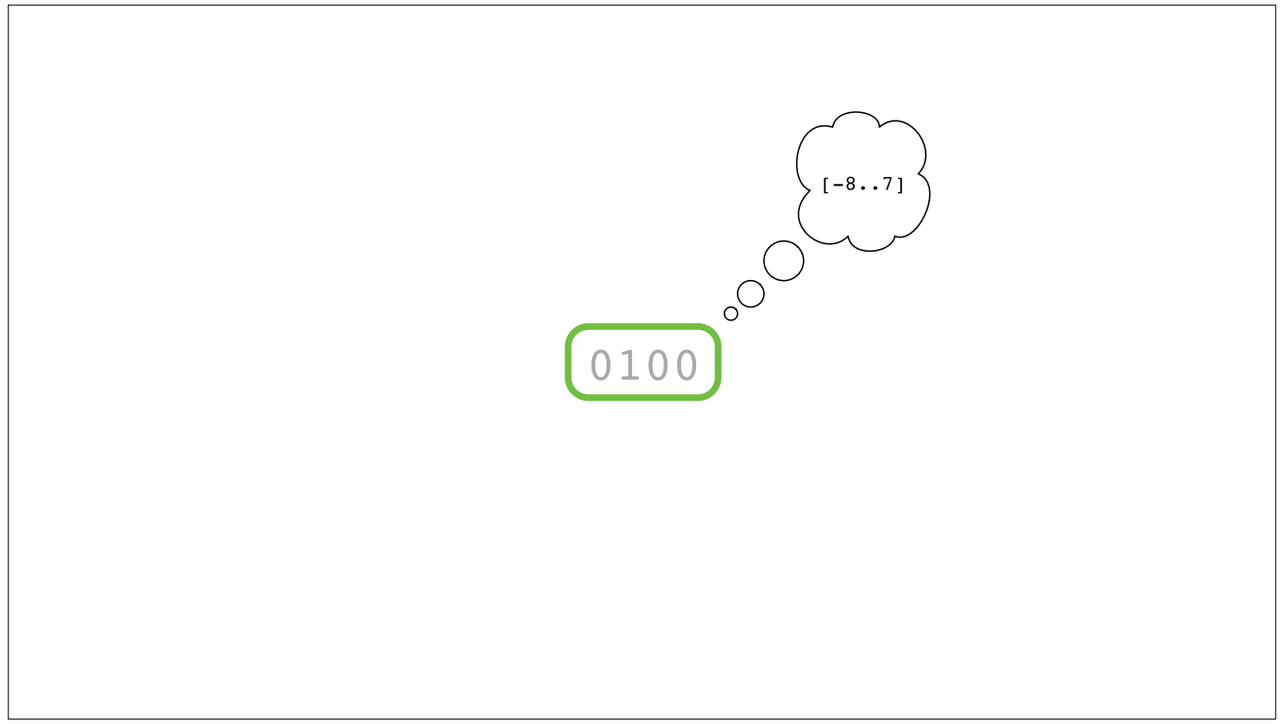
This is a representational relationship

Representational relationships are what we mean by object validity
An object is valid when this relationship holds
Standard has two meanings of valid

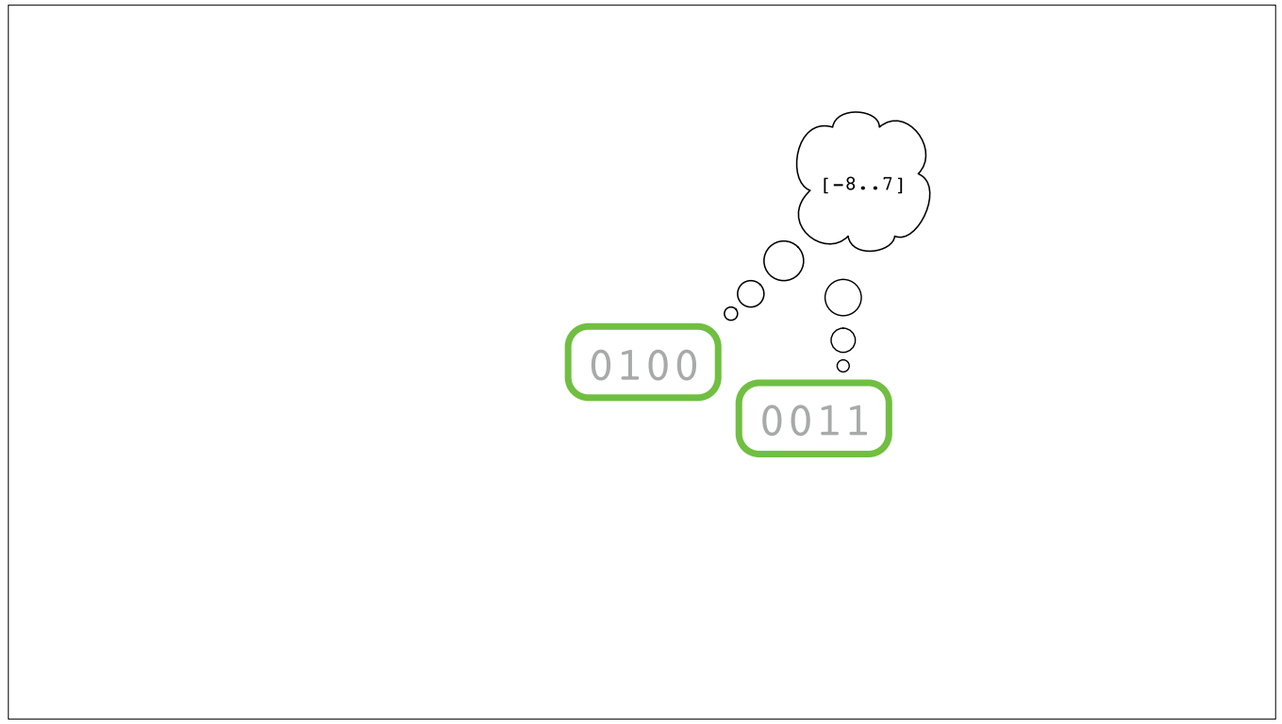
- Iterator validity is this form of valid - iterator represents a position in a sequence, when it doesn’t, it is said to be invalid
- Valid but unspecified - really is invalid - meaning is that the type relationship holds but the value relationship is lost - same as iterator



This is a representational relationship



This is what we mean by *type*



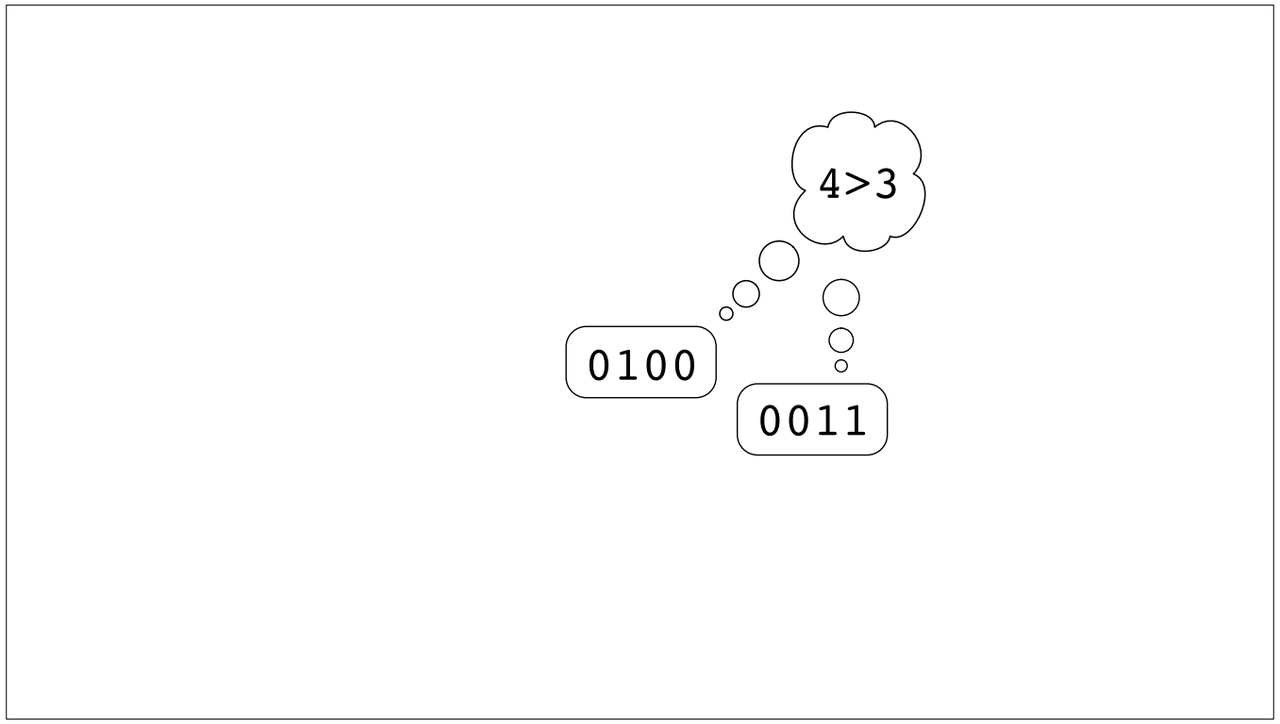
This is the type relationship

Discuss validity, invariants, Safety

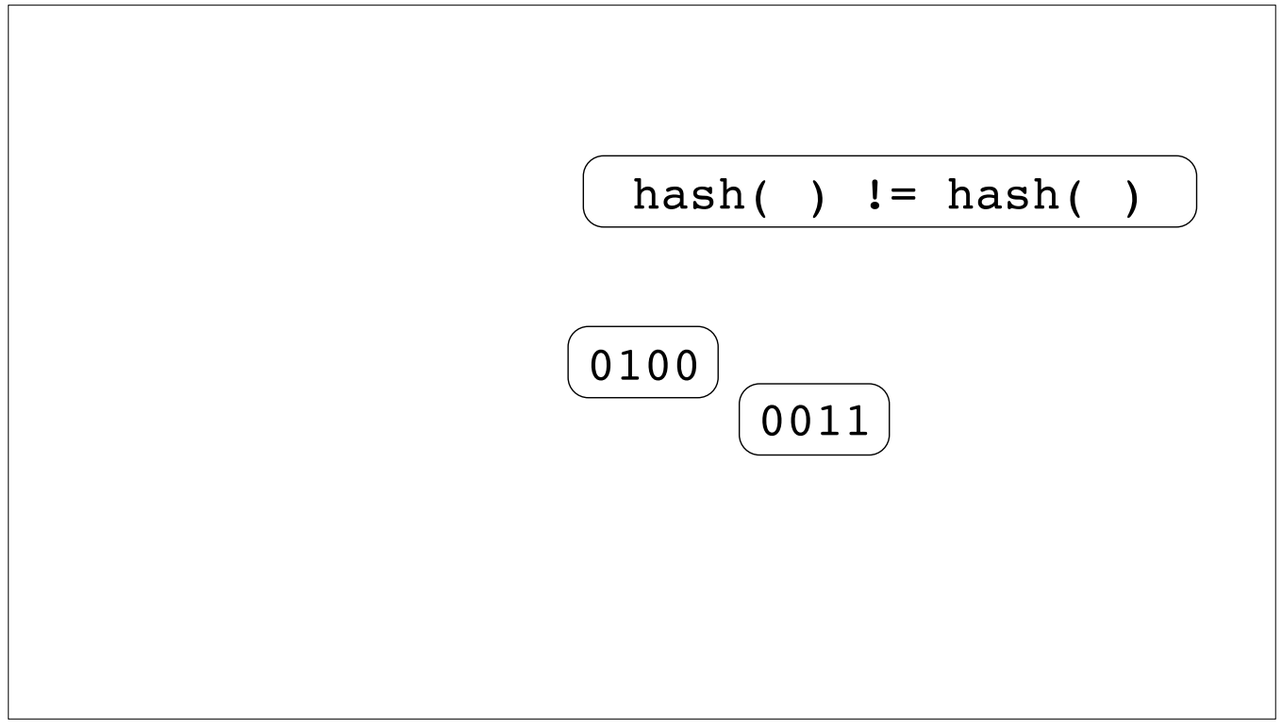
Standard has two definitions of validity

EoP partially formed

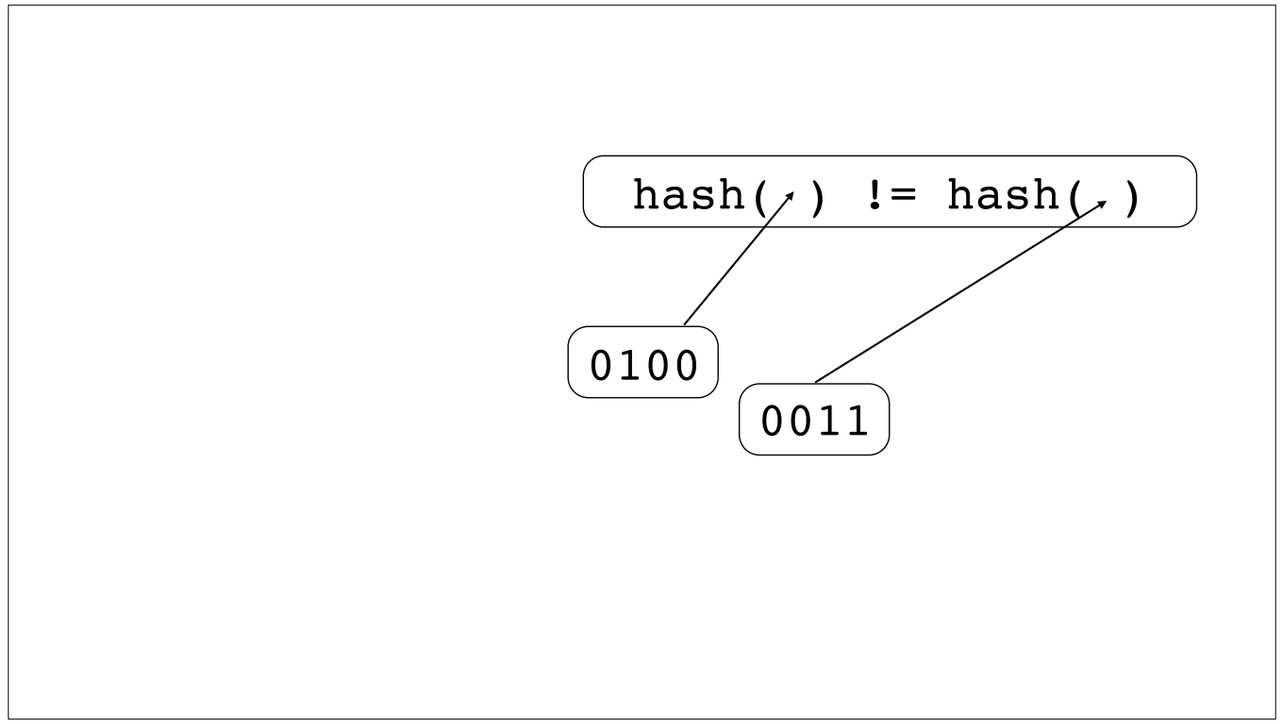
Dangling Pointer…



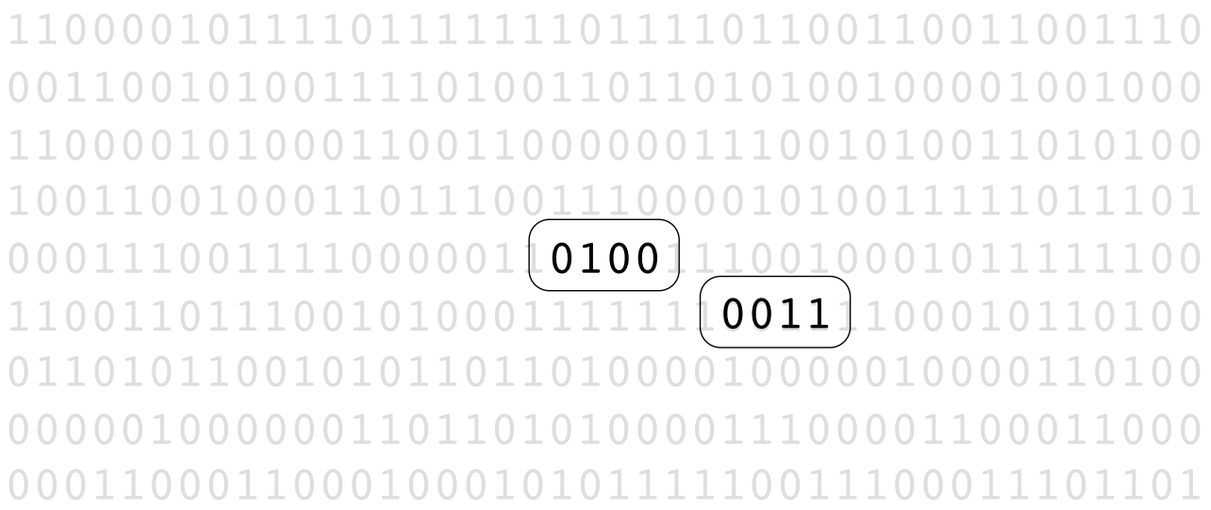
Now we have all kinds of mathematical relationships and structures
This is a ordered relationship



A representational relationship - this relationship connects to semantics because object with the same type and representation have equal values.


hash( ) != hash( )
0100
0011

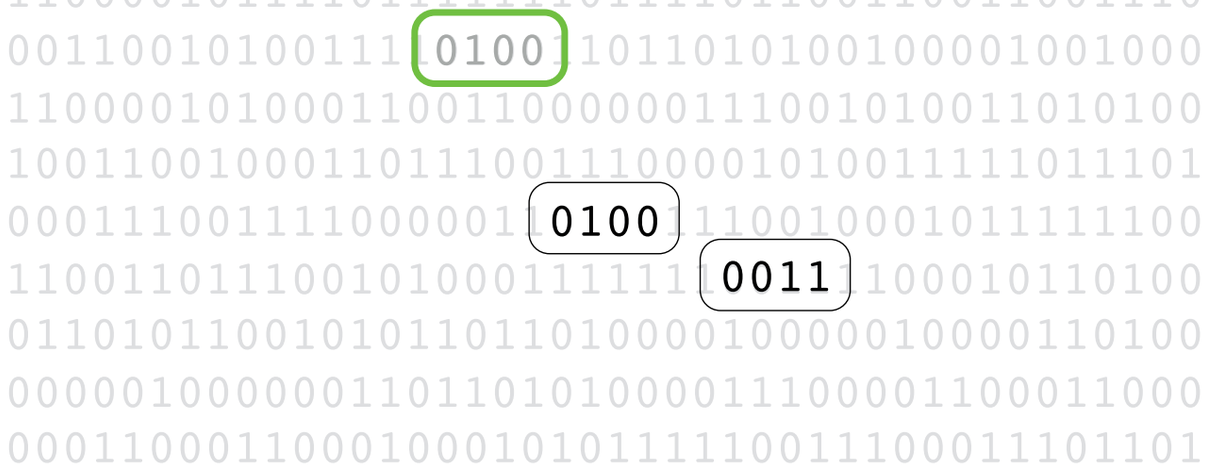
Memory Space



What makes this object different from other bits is the fact that it has a semantic relationship
An object in a physical space has a location - in both time and space
[These are “structural relationships but since all relationships form structures - we refer to these as physical relationship.]
Joke about physical relationships, Jon & Marshall - Haskell - puritan
The object has a relationship with the space within which it resides and with other objects in the same space.
The physical relationship has no inherent meaning - however we can utilize it to encode meaning.

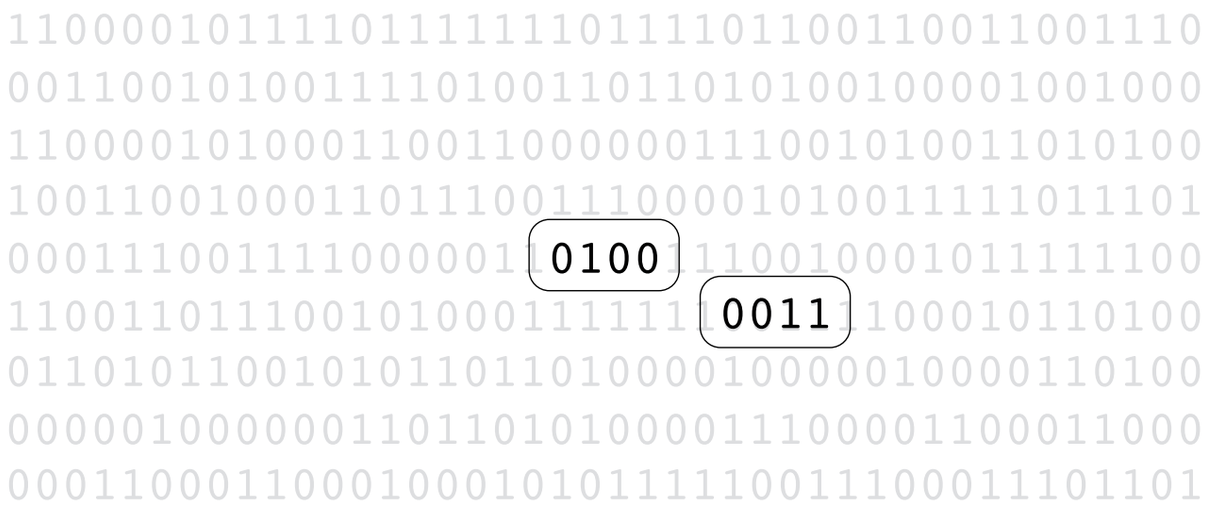
[[SORT VALUES HERE}

Memory Space



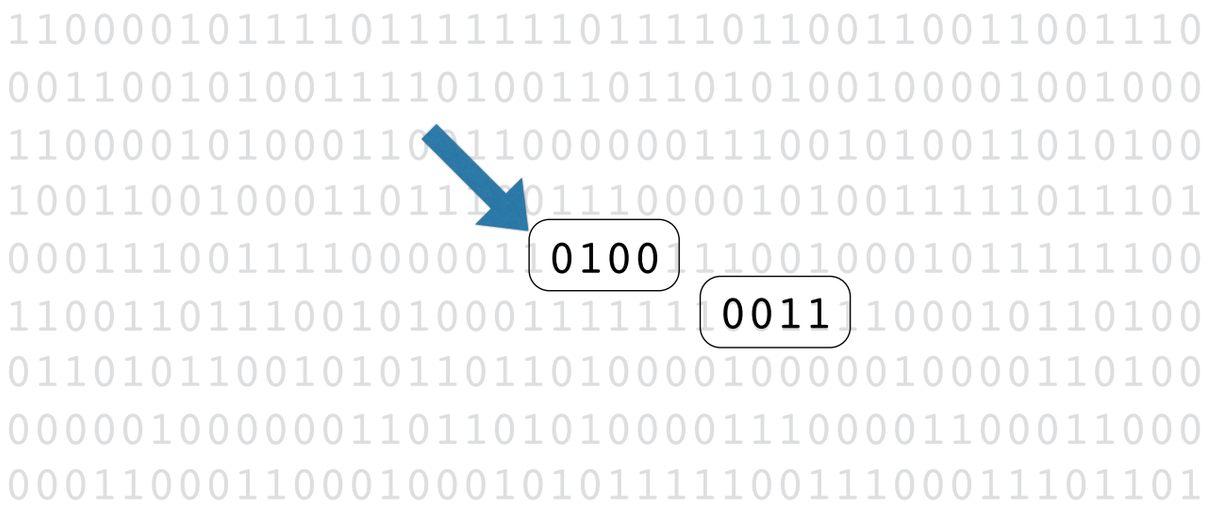
Adobe

Memory Space



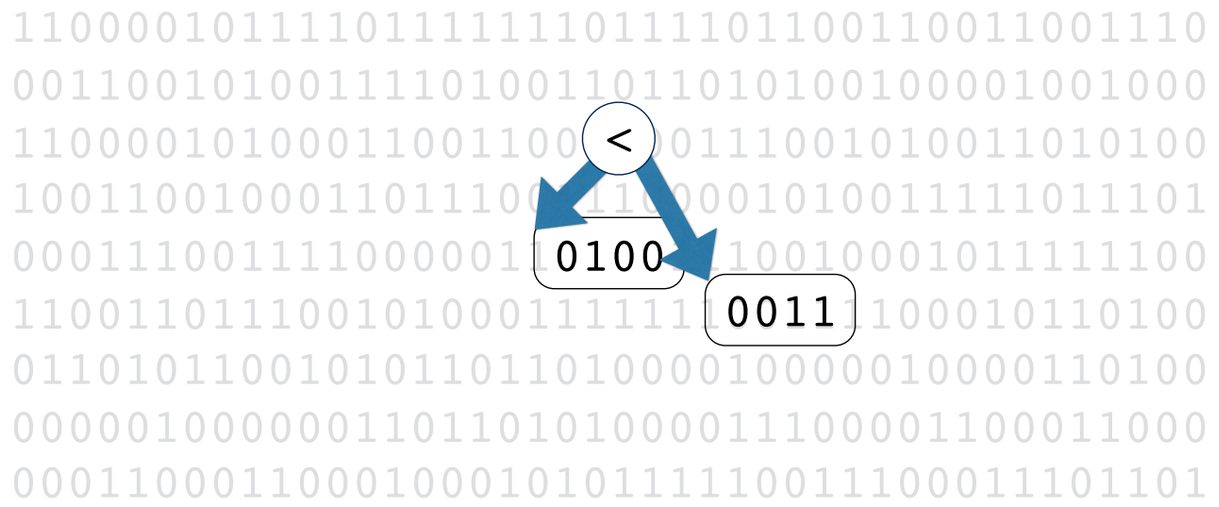
Adobe

Memory Space



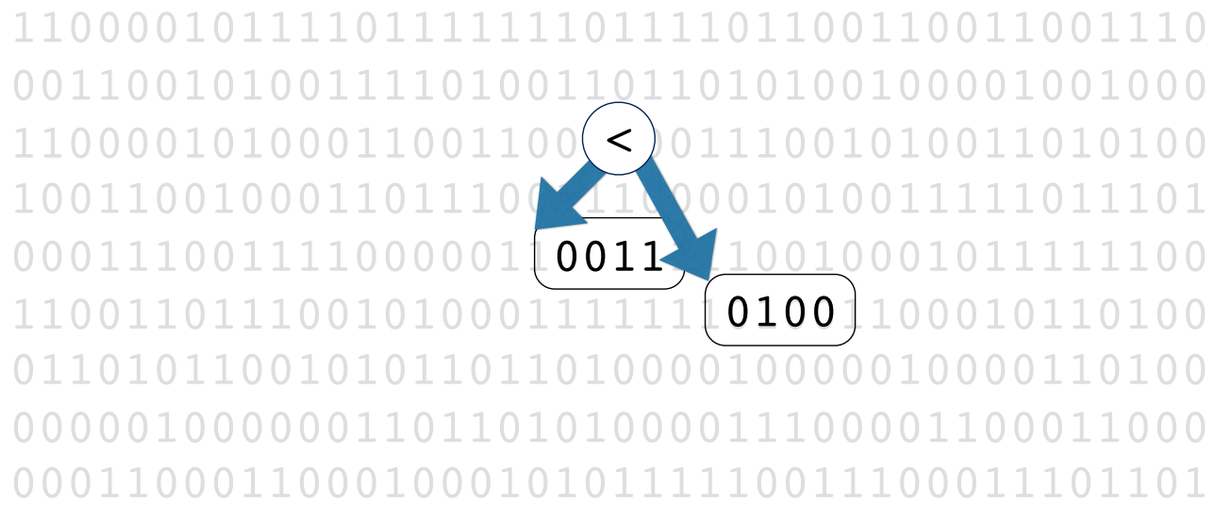
Adobe

Memory Space

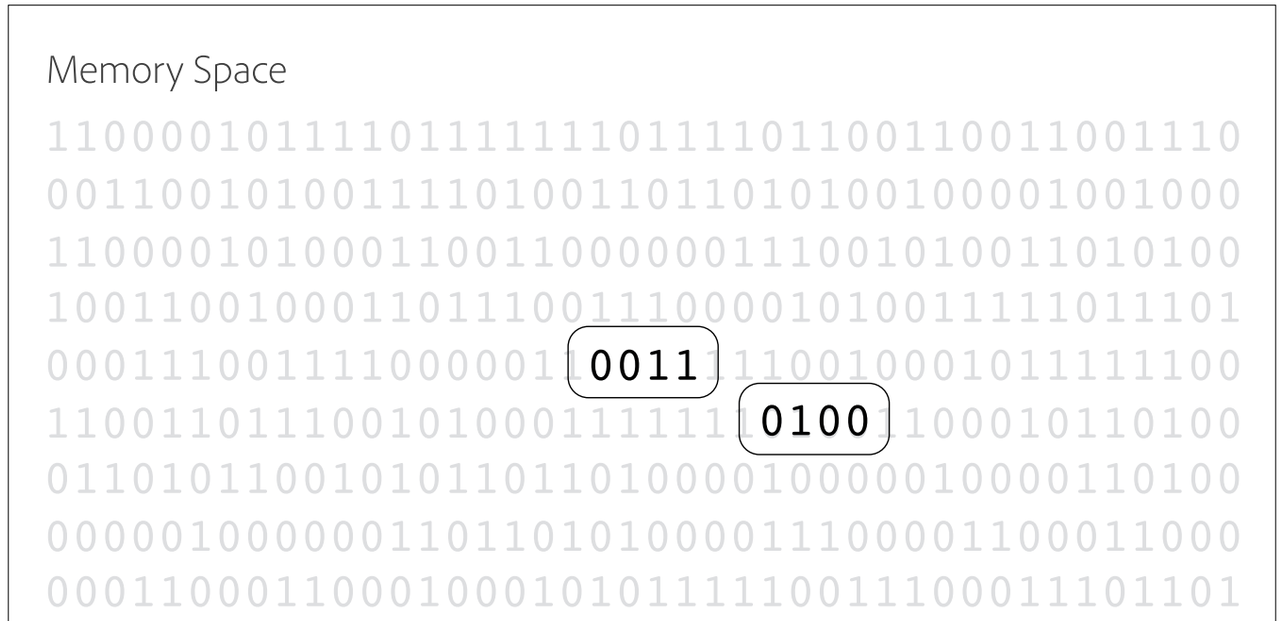


Adobe

Memory Space

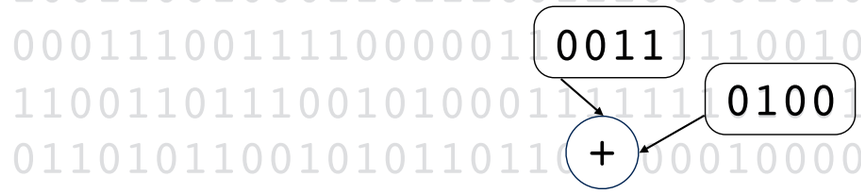


Adobe



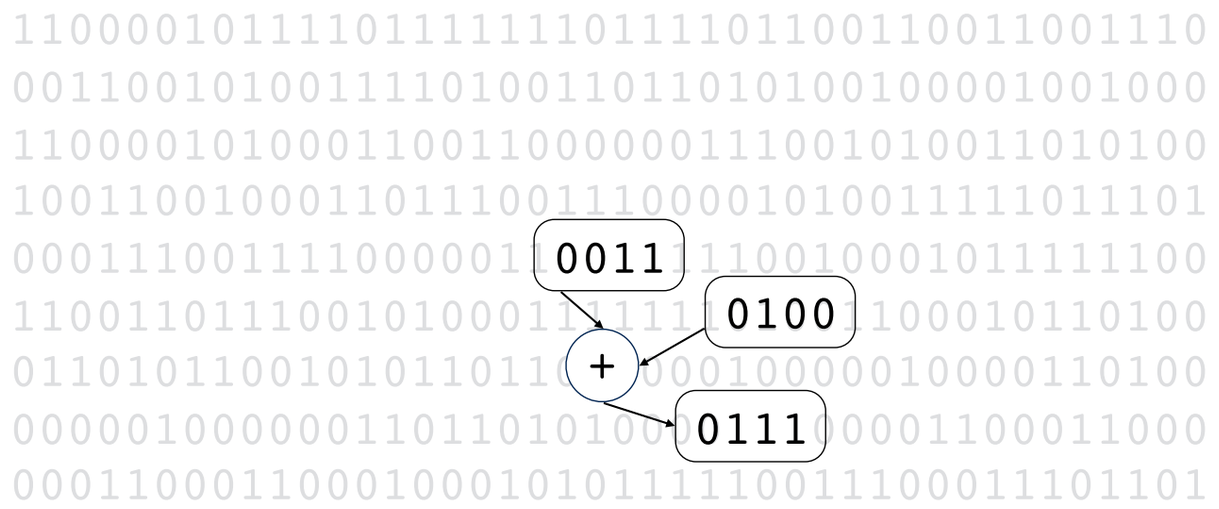
Functional Relationships form a dependency graph - for more on these see my concurrency talk
Cannot not add two numbers that overflow.

Memory Space



Adobe

Memory Space



Adobe

Object Type

- A type is a correspondence between entities with common properties, or *species*, and a set of values
  - Example species: color, integer, dog
- An object type is a pattern for storing a value in memory

See EoP, Chapter 1

C++20

C++20

- Two new features specifically about relationships

C++20

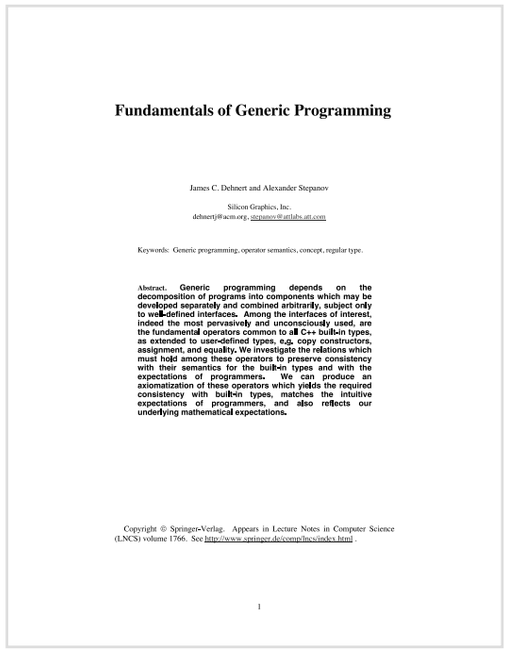
- Two new features specifically about relationships
  - Concepts

C++20

- Two new features specifically about relationships
  - Concepts
  - Contracts

C++20

- ~~Two~~One new feature~~s~~ specifically about relationships
  - Concepts
  - ~~Contracts~~



Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@atlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. **Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.**

1998

1998 - Roots in Hoar Logic

Adobe

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

1

1998

Adobe

"We call the set of axioms satisfied by a data type and a set of operations on it a *concept*."

"We call the set of axioms satisfied by a data type and a set of operations on it a ***concept***."

Algebraic structures
Multi-sorted algebras

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.6, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y < r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leq r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to nonnegative numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value representable.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers representable.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

A10$_I$ $\neg\exists x \forall y \quad (y < x)$,

where all finite arithmetics satisfy:

A10$_F$ $\forall x \quad (x < \max)$

where "max" denotes the largest integer representable.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

A11$_S$ $\neg\exists x \quad (x = \max + 1)$ (strict interpretation)

A11$_B$ $\max + 1 = \max$ (firm boundary)

A11$_M$ $\max + 1 = 0$ (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

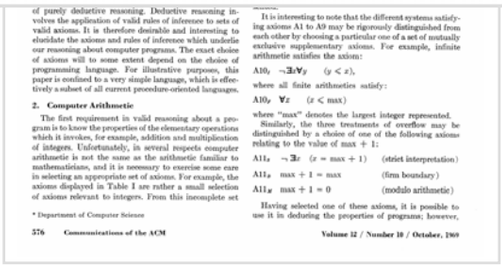
1969

1998 - Roots in Hoar Logic

Adobe

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*



of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

* Department of Computer Science

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_i \quad \neg\exists x \forall y \quad (y \leq x)$,

where all finite arithmetics satisfy:

$A10_f \quad \forall x \quad (x < \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_s \quad \neg\exists x \quad (x = \max + 1)$ (strict interpretation)

$A11_e \quad \max + 1 = \max$ (firm boundary)

$A11_w \quad \max + 1 = 0$ (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

1969

Adobe

Equality

- Two objects are equal iff their values correspond to the same entity
- From this definition we can derive the following properties:

$$(\forall a) a = a. \qquad \text{(Reflexivity)}$$
$$(\forall a, b) a = b \Rightarrow b = a. \qquad \text{(Symmetry)}$$
$$(\forall a, b, c) a = b \wedge b = c \Rightarrow a = c. \qquad \text{(Transitivity)}$$

Read math notation
Same axioms for any equivalence structure
These axioms are necessary but not sufficient
Reflexivity is implied
Connected axioms form an equivalence structure
? Use “Dave” example ?

Equality

- By convention we use = for equality in mathematics and == (C, C++, Java), or equals() (Java classes) for equality in programming
- Properties, or *axioms*, follow from definition
- A collection of connected axioms form an *algebraic structure*
- Connected type requirements form a *concept*

The structure for equivalence relation is a setoid.

Copy and Assignment

- Properties of copy and assignment:

$$b \to a \Rightarrow a = b \qquad \text{(copies are equal)}$$

$$a = b = c \wedge d \neq a, d \to a \Rightarrow a \neq b \wedge b = c \qquad \text{(copies are disjoint)}$$

- Copy is connected to equality

Natural Total Order

- The natural total order is a total order that respects the other fundamental operations of the type
- A total order has the following properties:

$$(\forall a, b) \text{exactly one of the following holds:}$$
$$a < b, b < a, \text{or } a = b. \qquad \text{(Trichotomy)}$$
$$(\forall a, b, c) a < b \wedge b < c \Rightarrow a < c. \qquad \text{(Transitivity)}$$

Back to natural total order
Now we see why understanding equality is important

Natural Total Order

- Example: Integer < is consistent with addition.

$$(\forall n \in \mathbb{Z}) n < (n + 1).$$

[read the math]
This relates less-than to addition

Concepts

- A collection of connected axioms form an *algebraic structure*
- Connected type requirements form a concept

- Qualified axioms are (generally) not actionable
  - Concepts work by associating semantics with the name of an operation

We cannot (currently) express concepts directly in the language in a verifiable way (not that it is impossible but requires an algebraic proof system).

What we can do is associate semantics with names. When we use the same name with different semantics this breaks. At a point we are _only_ requiring names which gives us no assurance that the operation does what is expected.

Alex, Dave Musser, and Deepak Kapur worked on a language to do this, known as Tecton.

Software is defined on Algebraic Structures

Adobe

Need axioms - this is about Concepts
From [2018-11-09-generic-programming.

Applying "Design by Contract"

Bertrand Meyer
Interactive Software Engineering

Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

As object-oriented techniques steadily gain ground in the world of software development, users and prospective users of these techniques are clamoring more and more loudly for a "methodology" of object-oriented software construction — or at least for some methodological guidelines. This article presents such guidelines, whose main goal is to help improve the reliability of software systems. *Reliability* is here defined as the combination of correctness and robustness or, more prosaically, as the absence of bugs.

Everyone developing software systems, or just using them, knows how pressing this question of reliability is in the current state of software engineering. Yet the rapidly growing literature on object-oriented analysis, design, and programming includes remarkably few contributions on how to make object-oriented software more reliable. This is surprising and regrettable, since at least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

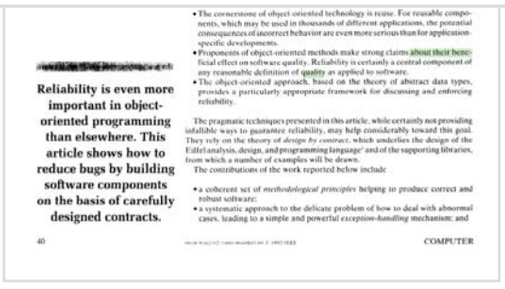
1986 (original)

1998 - Roots in Hoar Logic

Applying “Design by Contract”

Bertrand Meyer

Interactive Software Engineering



Reliability is even more important in object-oriented programming than elsewhere. This article shows how to reduce bugs by building software components on the basis of carefully designed contracts.

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application-specific developments.
- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.
- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

The pragmatic techniques presented in this article, while certainly not providing infallible ways to guarantee reliability, may help considerably toward this goal. They rely on the theory of *design by contract*, which underlies the design of the Eiffel analysis, design, and programming language[1] and of the supporting libraries, from which a number of examples will be drawn.

The contributions of the work reported below include

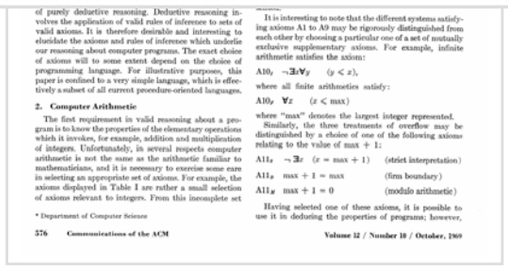
- a coherent set of *methodological principles* helping to produce correct and robust software;
- a systematic approach to the delicate problem of how to deal with abnormal cases, leading to a simple and powerful *exception-handling* mechanism; and

1986 (original)

Adobe

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*



of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set

* Department of Computer Science

576 Communications of the ACM

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I \quad \neg\exists x \forall y \quad (y \leq x)$,

where all finite arithmetics satisfy:

$A10_F \quad \forall x \quad (x < \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S \quad \neg\exists x \quad (x = \max + 1)$ (strict interpretation)

$A11_F \quad \max + 1 = \max$ (firm boundary)

$A11_M \quad \max + 1 = 0$ (modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

Volume 12 / Number 10 / October, 1969

1969

Adobe

1998 - Roots in Hoar Logic

Contracts

- Originally part of the Eiffel language
- Contracts allow the specification of constraints
  - Preconditions (require)
  - Postconditions (ensure)
  - Class Invariants

Contracts

- Contracts are actionable predicates on values

"In some cases, one might want to use quantified expressions of the form "For all *x* of type *T*, *p(x)* holds" or "There exists *x* of type *T*, such that *p(x)* holds," where *p* is a certain Boolean property. Such expressions are not available in Eiffel."

Concepts and Contracts

- Concepts describe relationships between operations on a type
- Contracts describe relationships between values

- The distinction is not always clear
  - i.e. The comparison operation passed to **`std::sort`** must implement a *strict weak ordering relation*

[placeholder]

- Add example for concepts and contracts in C++

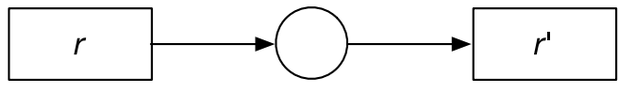
No Raw Loops

```
// Next, check if the panel has moved to the other side of another panel.
const int center_x = fixed_panel->cur_panel_center();
for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
```

No Raw Loops

```
std::rotate(p, f, f + 1);
```

No Raw Loops



No Incidental Data Structures

```
class view {
    std::list<std::shared_ptr<view>> _children;
    std::weak_ptr<view> _parent;
    //...
};
```

No Incidental Data Structures

```
adobe::forest<view>
```

No Incidental Data Structures

views

Whole-Part Relationship

- A part which is referred to indirectly is a *remote part*
- An object with remote parts can be *moved*
  - Moving an object only requires storage for the local parts
  - Any reified relationship can be maintained and *moved*

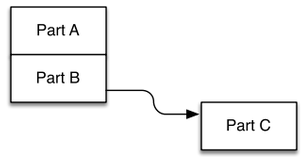
[Put this slide where?]

Moore’s law hasn’t kept up for memory speeds

Each bump in the memory level hierarchy costs us an order of magnitude in performance

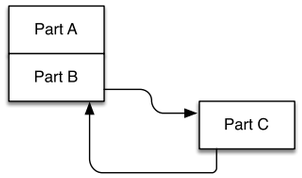
Whole-Part Relationships and Composite Objects

- Connected



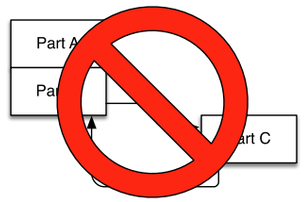
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular



Whole-Part Relationships and Composite Objects

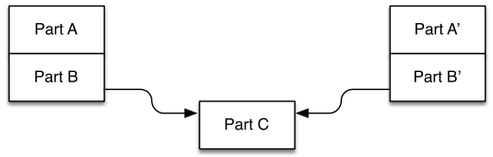
- Connected
- Noncircular



Adobe

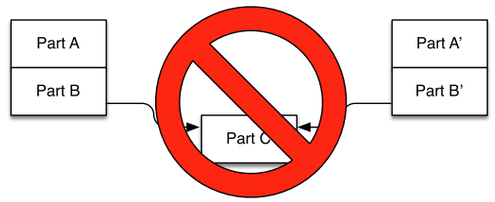
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint



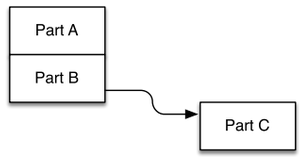
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint



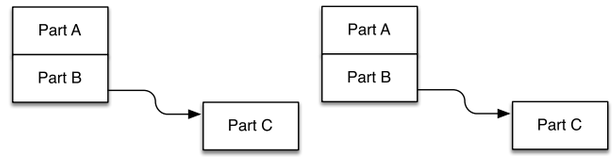
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning



Whole-Part Relationships and Composite Objects

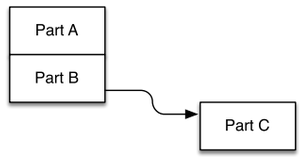
- Connected
- Noncircular
- Logically Disjoint
- Owning



Elements of Programming, Chapter 12

Whole-Part Relationships and Composite Objects

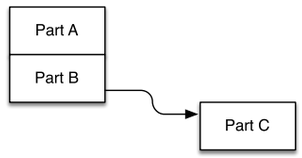
- Connected
- Noncircular
- Logically Disjoint
- Owning



Whole-Part Relationships and Composite Objects

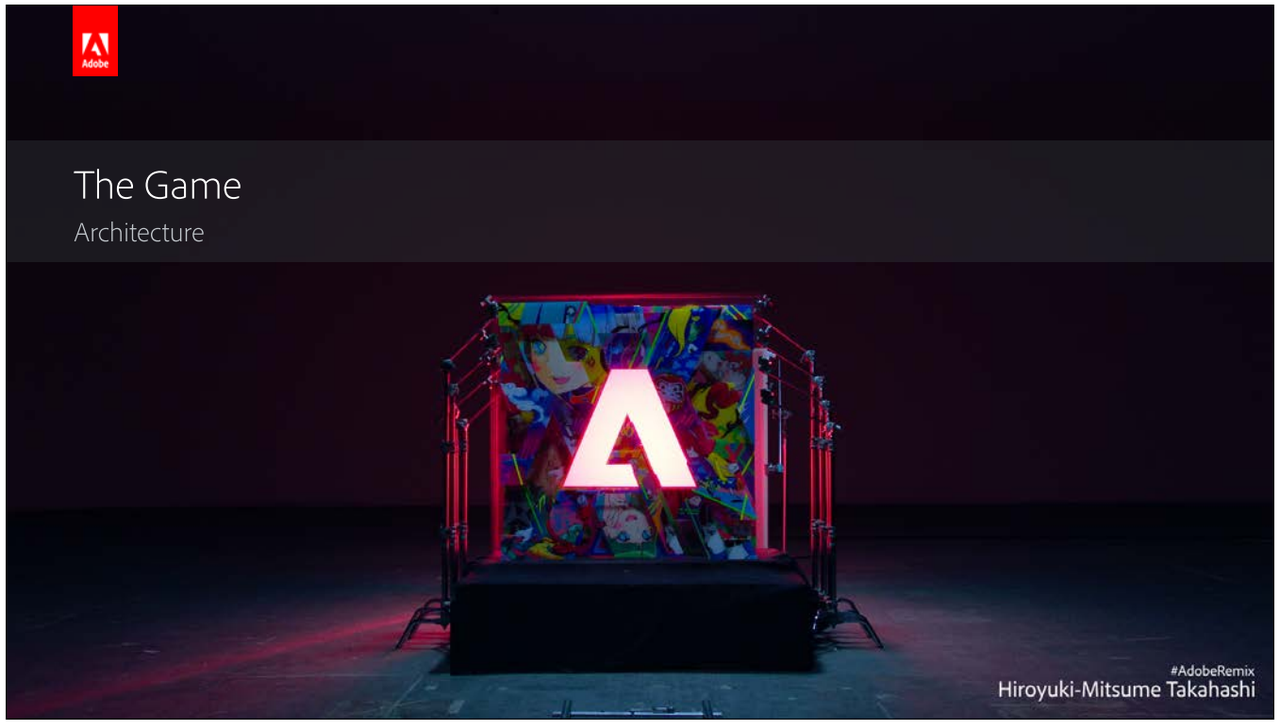
- Connected
- Noncircular
- Logically Disjoint
- Owning

- Standard Containers are Composite Objects



The Game

Architecture



Architecture is the art and practice of designing and constructing structures.

Task

- Save the document every 5 minutes, after the application has been idle for at least 5 seconds.

This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

Task

- **Save the document every 5 minutes**, after the application has been idle for at least 5 seconds.

This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

Task

- ~~Save the document every 5 minutes~~, after the application has been idle for at least 5 seconds.

This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

Task

- After the application has been idle for at least *n* seconds do *something*

Adobe

This code isn’t bad, but has a number of issues, _list_idle should be atomic, but can’t be, needs a mutex but that would complicate this code more than necessary for my point

Task

- After the application has been idle for at least *n* seconds do *something*

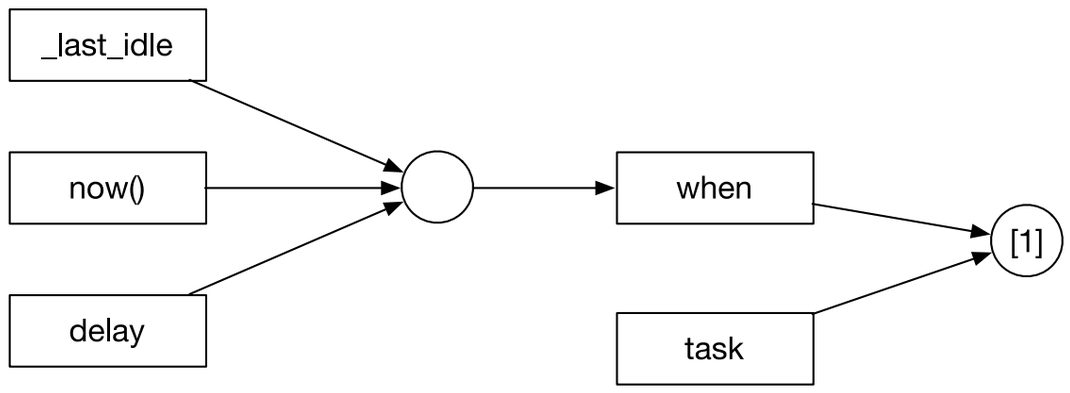
```
extern system_clock::time_point _last_idle;
void invoke_after(system_clock::duration, function<void()>);

template <class F> // F is task of the form void()
void after_idle(F task, system_clock::duration delay) {
    auto when = delay - (system_clock::now() - _last_idle);

    if (system_clock::duration::zero() < when) {
        invoke_after([=] { after_idle(task, delay); });
    } else {
        task();
    }
}
```

Visualizing the Relationships

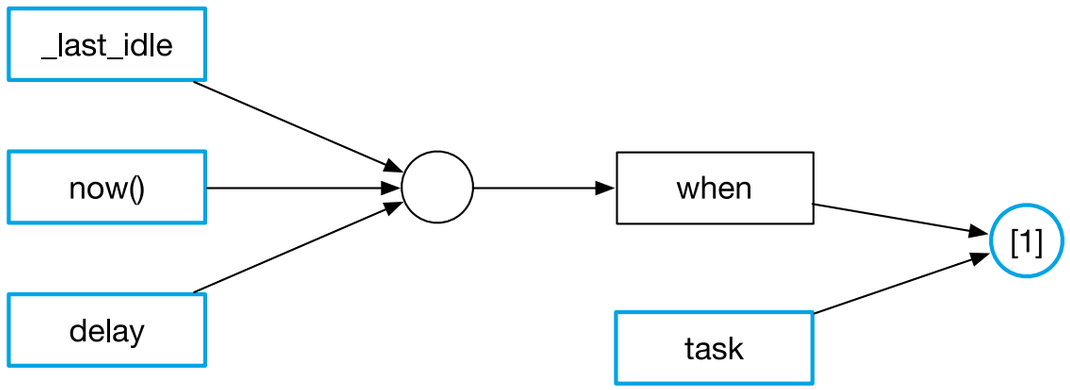
- The structure, ignoring the recursion in **`invoke_after`**[1]



[add a slide about source and derived entities]

Visualizing the Relationships

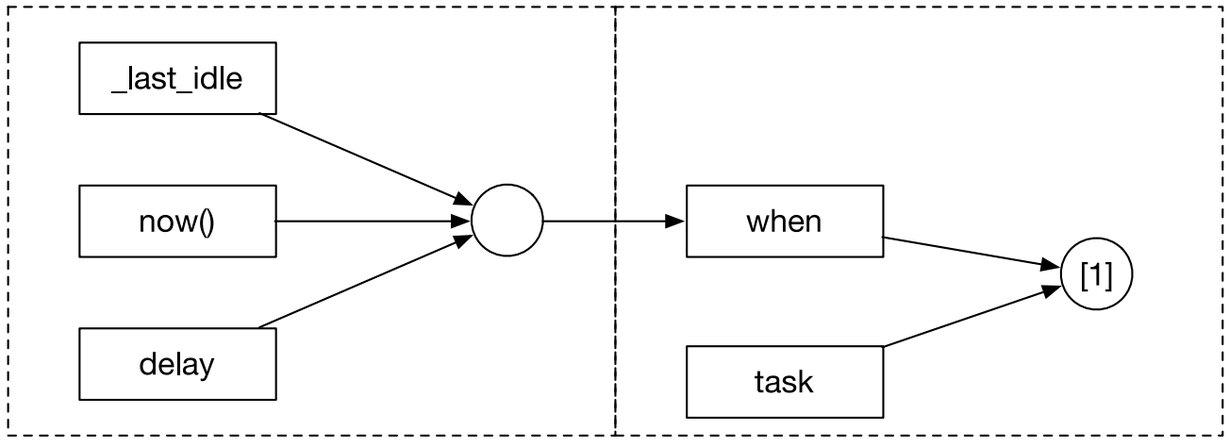
- The arguments and dependencies



This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

Visualizing the Relationships

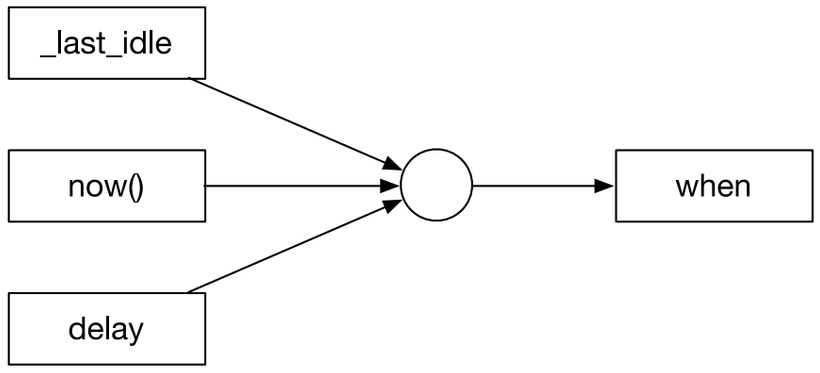
- Two operations



This representation can be very useful for reasoning about relationships -
The intent is not to capture every aspect of the relationship, but to to reason about the _structure_.

Visualizing the Relationships

```
auto when = delay - (system_clock::now() - _last_idle);
```



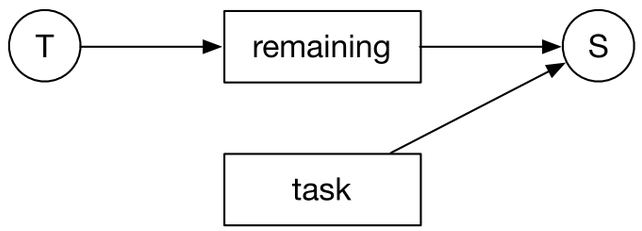
The first relationship is just this line of code - not something we would consider refactoring
But what is this?
A: A timer

Visualizing the Relationships

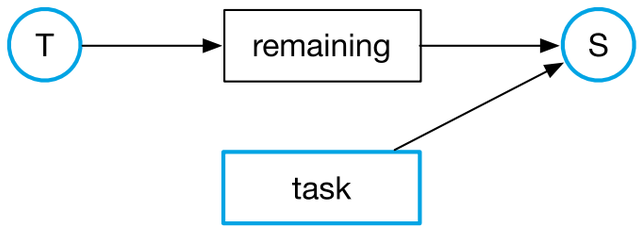
```
auto when = delay - (system_clock::now() - _last_idle);
```

The first relationship is just this line of code - not something we would consider refactoring
But what is this?
A: A timer

On Expiration



On Expiration



On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();
    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] { on_expiration_(scheduler, timer, task); });
    } else {
        task();
    }
}
```


No external dependencies - not even on the standard library
Requirement of the timer result is:
- can be compared against zero (could have also chosen convertible to bool)
- can be passed to the scheduler

Scheduler takes whatever the timer returns and the task
Task is invokable

Issue of task possibly executing immediately…

On Expiration

```
template <class S, class T, class F>
void on_expiration_(S scheduler, T timer, F task) {
    auto remaining = timer();
    if (decltype(remaining){0} < remaining) {
        scheduler(remaining, [=] { on_expiration_(scheduler, timer, task); });
    } else {
        task();
    }
}

template <class S, class T, class F>
void on_expiration(S scheduler, T timer, F task) {
    scheduler(timer(), [=] { on_expiration_(scheduler, timer, task); });
}
```


As an exercise to the reader -
 use std::invoke and std::move as appropriate

[Add a slide showing use?]

Registry

- A registry is a container supporting the following operations
  - Add an object, and obtain a *receipt*
  - Use the receipt to retrieve the object or remove it
  - Operate on the object in the container

  - Example: signal handler

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Registry

```
template <class T>
class registry {
    unordered_map<size_t, T> _map;
    size_t _id = 0;
public:
    auto append(T element) -> size_t {
        _map.emplace(_id, move(element));
        return _id++;
    }

    void erase(size_t id) { _map.erase(id); }

    template <typename F>
    void for_each(F f) const {
        for (const auto& e : _map)
            f(e.second);
    }
};
```

Russian Coat Check Algorithm

- Receipts are **ordered**
  - Coats always appended with stub
  - Binary search to retrieve coat by matching receipt to stub
    - When more than half the slot are empty, compact the coats

- Coats are always ordered by receipt stubs

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

```
template <class T>
class registry {
    vector<pair<size_t, optional<T>>> _map;
    size_t _size = 0;
    size_t _id = 0;

public:
    //...
```


size_t is our stub, optional<T> because we can have empty slots on the coat rack
_size

Russian Coat Check Algorithm

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

Russian Coat Check Algorithm

```
auto append(T element) -> size_t {
    _map.emplace_back(_id, move(element));
    ++_size;
    return _id++;
}
//...
```

size_t is our stub, optional<T> because we can have empty slots on the coat rack
_size

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	3	4	
a	d	e	

Russian Coat Check Algorithm

```
void erase(size_t id) {
    auto p = lower_bound(
        begin(_map), end(_map), id,
        [](const auto& a, const auto& b) { return a.first < b; });
    if (p == end(_map) || p->first != id) return;
    p->second.reset();
    --_size;
    if (_size < (_map.size() / 2)) {
        _map.erase(remove_if(begin(_map), end(_map),
                             [](const auto& e) { return !e.second; }),
                   end(_map));
    }
}
//...
```

size_t is our stub, optional<T> because we can have empty slots on the coat rack
_size

Russian Coat Check Algorithm

0	3	4	
a	d	e	

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	3	4	8	9	
a	d	e	i	j	

Russian Coat Check Algorithm

```
    template <typename F>
    void for_each(F f) {
        for (const auto& e : _map) {
            if (e.second) f(*e.second);
        }
    }
};
```


size_t is our stub, optional<T> because we can have empty slots on the coat rack
_size

Double-entry bookkeeping

$$assets = liabilities + equity$$

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention

$$assets = liabilities + equity$$

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- By ensuring all transactions are made against two separate accounts

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- By ensuring all transactions are made against two separate accounts
  - The probability of error is significantly reduced

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- By ensuring all transactions are made against two separate accounts
  - The probability of error is significantly reduced
  - The account gains *transparency;* making it easier to audit, and detect fraud

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- By ensuring all transactions are made against two separate accounts
  - The probability of error is significantly reduced
  - The account gains *transparency;* making it easier to audit, and detect fraud

Double-entry bookkeeping

- *Double-entry bookkeeping* is an accounting tool for error detection and fraud prevention
- Relies on the accounting equation

$$assets = liabilities + equity$$

- By ensuring all transactions are made against two separate accounts
  - The probability of error is significantly reduced
  - The account gains *transparency;* making it easier to audit, and detect fraud

- An example of *equational reasoning*

Double-entry bookkeeping

Pronounced “Luca Pacholi”

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy
    - Leading to the rise of one oft he most powerful family dynasties in history

Double-entry bookkeeping

- Pioneered in the 11th century by the Jewish banking community
  - Likely developed independently in Korea in the same time period
- In the 14th century, double-entry bookkeeping was adopted by the Medici bank
  - Credited with establishing the Medici bank as reliable and trustworthy
    - Leading to the rise of one oft he most powerful family dynasties in history
- Double-entry bookkeeping was codified by Luca Pacioli (the Father of Accounting) in 1494

Luca Pacioli

The dude photobombing the portrait is known as the “perpetual student”, may be Dürer.

Double-entry bookkeeping

[Put on different slide - What is the value of unit testing? What is the value of the largest bank in Europe in the 15th century?”

Double-entry bookkeeping

- Every transaction is entered into twice, into at least two separate accounts

Double-entry bookkeeping

- Every transaction is entered into twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses

Double-entry bookkeeping

- Every transaction is entered into twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses
- This ensures the mechanical process of entering a transaction is done in two distinct ways

Double-entry bookkeeping

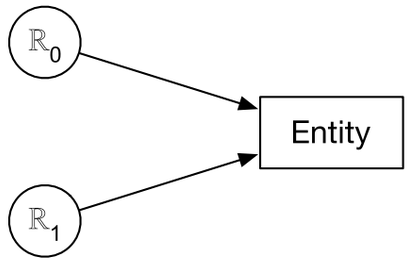
- Every transaction is entered into twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses
- This ensures the mechanical process of entering a transaction is done in two distinct ways

Double-entry bookkeeping

- Every transaction is entered into twice, into at least two separate accounts
- There are 5 standard accounts, Assets, Capital, Liabilities, Revenues, and Expenses
- This ensures the mechanical process of entering a transaction is done in two distinct ways

- If the accounting equation is not satisfied, then we have a *contradiction*

Contradictions



Same structure of a race condition
  or a shared pointer
Even with a mutex, the _implicit_ relation is “last one in wins” - which may not be wrong, but it is an unusual relation.

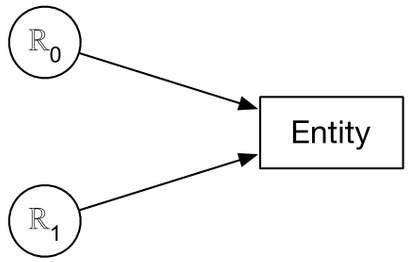
[Discuss mutex, “last one in” is really “any of”, null ptr (?), shared pointer, last on in? bridge to multiple writes to the same location, “Not saying it is wrong, but it’s probably wrong.”

cases:
  code represents different relationship which are somehow mutually exclusive - with non-local control
  code represents different sides of the same relationship (multiply example) - non-local
  Or entity is “most recent of” from implicit join
  Or incidental algorithm (via an incidental structure) - entity will converge to correct value?
  Or just wrong.

Cover self referential loops.]

Contradictions

- When two relationship imply the same entity has different values



References

- References are a many to 1 relationship

And pointers

References

- Properties of copying a reference:

$$b \to a \Rightarrow a = b \qquad \text{(copies are equal)}$$
$$a = b = c \wedge d \neq a, d \to a \Rightarrow a = b = c = d \qquad \text{(copies are not disjoint)}$$

Syntax isn’t important - same holds for pointers or references to java objects.

Reference as Proxy

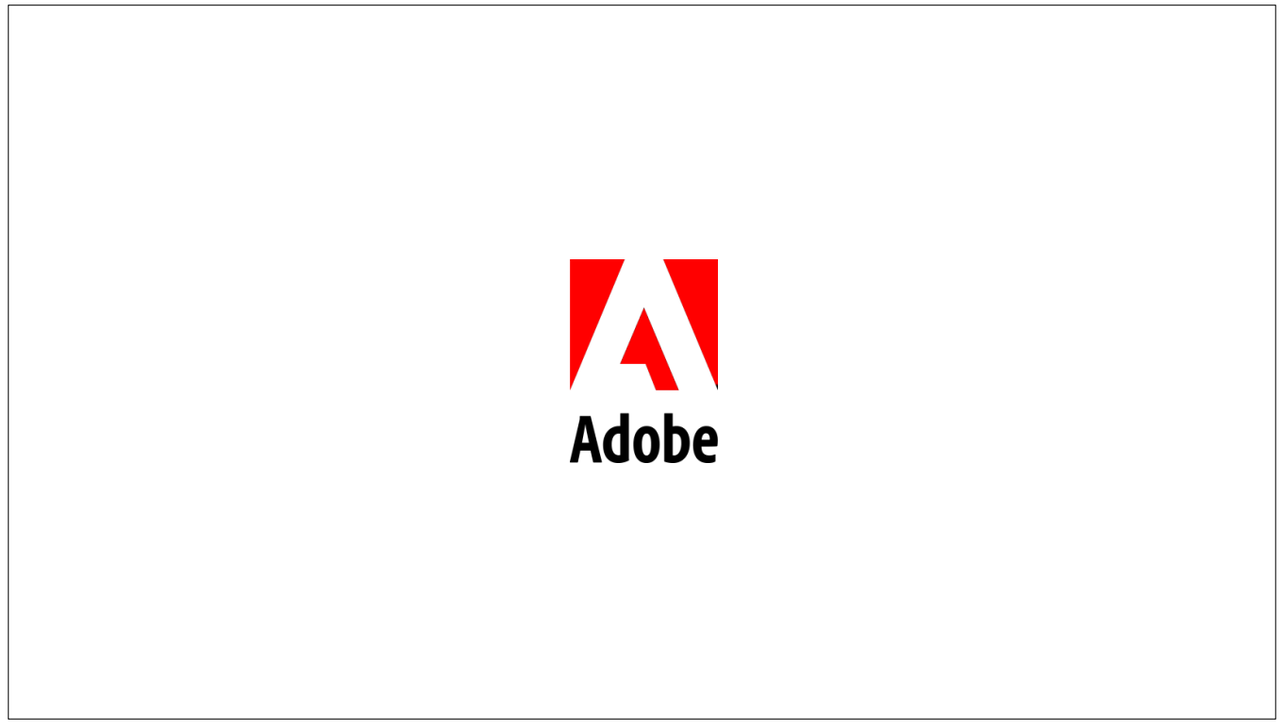
- A reference can act as a proxy to an object so long as the object is not modified or the cardinality of the relationship is one, and the lifetime of the object is greater than that of the reference

Break it down
Hence the const reference -

Reference as Proxy

- Functional, or value semantic, languages work by removing assignment and ensuring the lifespan of the objects, typically with garbage collection
- A shared reference to a mutable object breaks our ability to reason locally about code

A shared pointer is as good as a global variable


Adobe