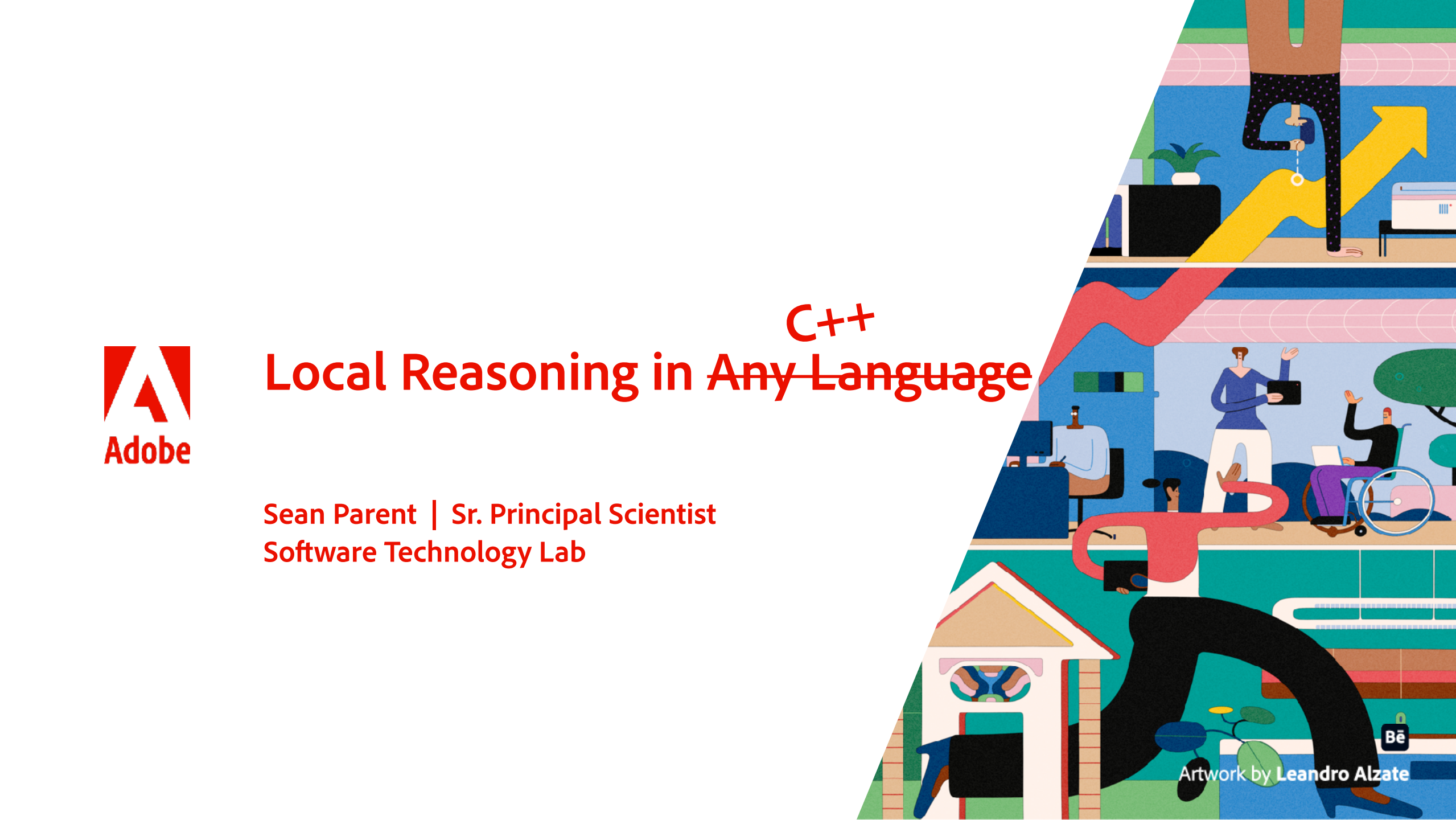


# Local Reasoning in ~~Any Language~~ C++

Sean Parent | Sr. Principal Scientist
Software Technology Lab

Artwork by Leandro Alzate

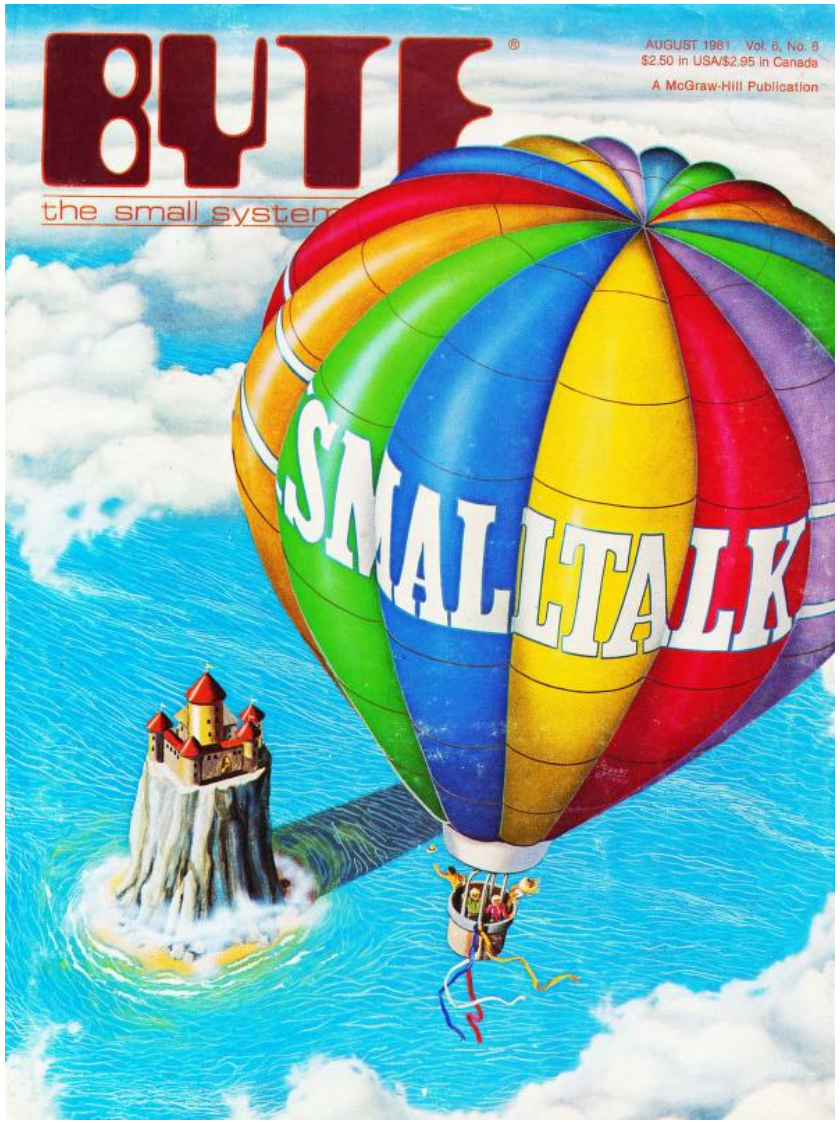
# Some Links

- https://developer.adobe.com/cpp/
- https://sean-parent.stlab.cc/papers-and-presentations/
- https://www.hylo-lang.org/

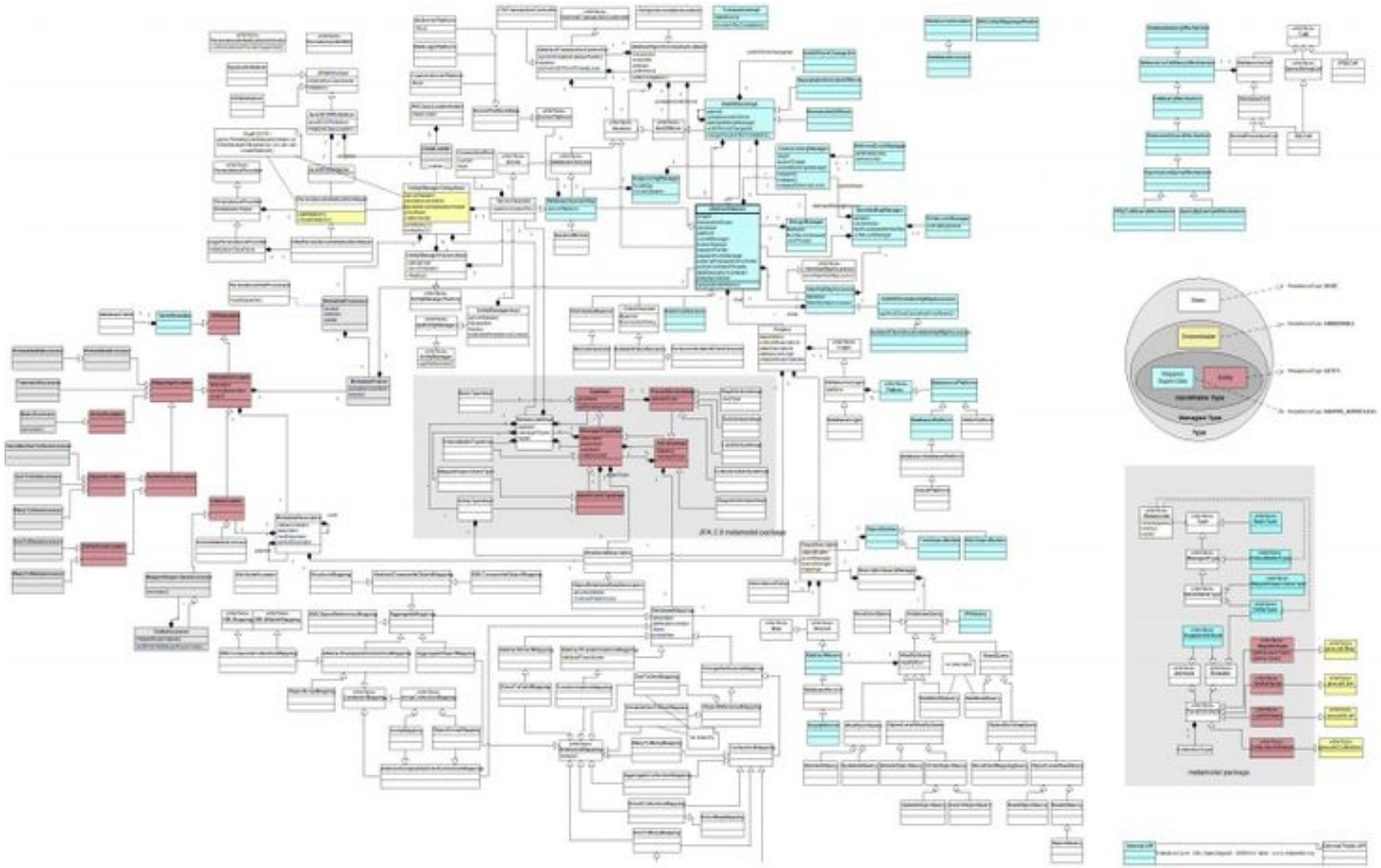
# Lokalt Resonnement ~~på Alle Språk~~ i C++

Sean Parent | Sr. Principal Scientist
Software Technology Lab

# Network of Object



# Network of Object



# Software Crisis

- OOP was supposed to solve the software crisis
- Wikipedia lists 17 major failed software projects totaling billions of dollars in losses since 1980

List of failed and overbudget custom software projects. (2024, August 28). In Wikipedia.

# Failed Software Projects

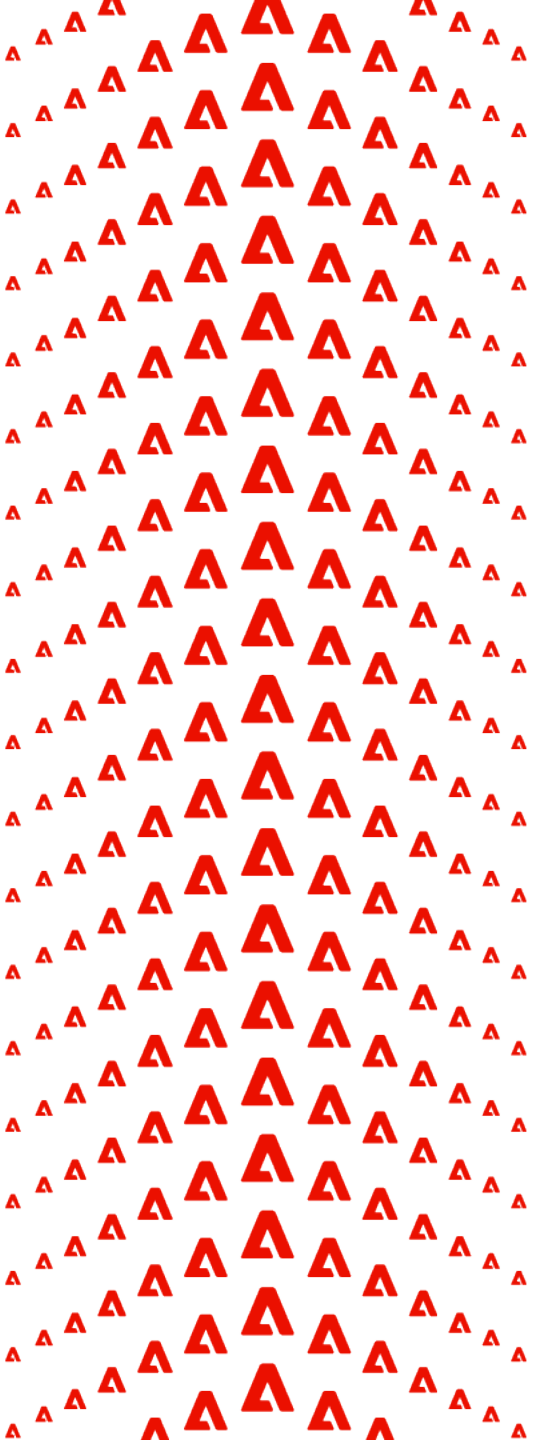
| Started | Terminated | System name | Type of system | Country or region | Type of purchaser | Problems | Cost *(expected)* | Outsourced or in-house? | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| 2017[11] | 2023[12] | Distributed Ledger Technology (generic name) | Electronic trading platform | Australia | Australian Stock Exchange | System was too complex and only 60% completed | $AU 170m expended | Outsourced | Cancelled |
| 2012 | 2014 | Cover Oregon | Healthcare exchange website | United States | State government | Site was never able to accept online enrollments, so users were instructed to mail in paper enrollments instead. | approx $200m | Outsourced | Cancelled, then client and supplier both sued each other |
| 2011 | 2014 | Pust Siebel | Police case management | Sweden | Police | Poor functioning, inefficient in work environments.[9] | SEK 300m ($35m)[10] | Outsourced | Scrapped |
| 2007 | 2014 | e-Borders | Advanced passenger information programme | United Kingdom | UK Border Agency | A series of delays. | over £412m (£742m) | Outsourced | Cancelled |
| 2009 | 2013 | The Surrey Integrated Reporting Enterprise | Crime & criminal intelligence logging | United Kingdom (Surrey) | Police Force | Not fit for purpose[8] | £14.8m | Outsourced | Scrapped |

# Software Crisis

- In all cases, mismanagement and development processes are blamed for the failures
- Software practice, available languages, libraries, tools, and fundamental algorithms and types are ignored

List of failed and overbudget custom software projects. (2024, August 28). In Wikipedia.

# Why Software Projects Fail



# Why Software Projects Fail

"The greatest limitation in writing software is our ability to understand the systems we are creating."

– *A Philosophy of Software Design*, John Osterhaut

# Local Reasoning

- *Local Reasoning* is the ability to reason about a defined unit of code and verify its correctness without understanding all the contexts in which it is used or the implementations upon which it relies.
- The two units of code this talk is concerned with are:
  - Functions
  - Classes
- The API is the key to local reasoning

# Terminology

- Local Reasoning is concerned with both sides of an API
  - The *client* code is the code calling a function or holding an instance of a class
  - The *implementor* code is the implementation of a function or class

# Functions

```
void f();
```

# Functions

```
// Does nothing.

void f();
```

# Functions

```
// Does nothing.

void f() { }
```

# Functions

```
// Returns the successor of `x`.

int f(int x) { return x + 1; }
```

# Functions

```
// Returns the successor of `x`.
// Precondition: `x < INT_MAX`.

int f(int x) { return x + 1; }
```

# Function Arguments

# Function Arguments

```
// Increments the value of `x` by `1`.
// Precondition: `x < INT_MAX`.

void a(int& x) { x += 1; }
```

# Function Arguments

```
// Increments the value of `x` by `1`
// Precondition: `x < INT_MAX`.
// Precondition: no other thread of execution is accessing `x`
//     during this operation.

void a(int& x) { x += 1; }
```

# General Preconditions:

- Arguments passed to a function by non-const reference cannot be accessed by other threads during the operation
- Arguments passed to a function by const reference cannot be written by another thread during the operation
- Unless otherwise specified

# Function Arguments

```
// Increments the value of `x` by `1`.
// Precondition: `x < INT_MAX`.

void a(int& x) { x += 1; }
```

# Why Mutation?

- Mutation is space efficient
- Mutation *may* be:
  - more performant.
  - simpler to reason about.

# Transformations and Actions

- A *transformation* is a regular unary function.
- Changing the state of an object by applying a transformation to it defines an *action* on the object.

```
x = f(x);
```

# Transformations and Actions

There is a duality between transformations and the corresponding actions: An action is defined in terms of a transformation, and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

– *Elements of Programming, Section 2.5*

# Argument Passing

- *let* arguments
  - `const T&`
- *inout* arguments
  - `T&`
- *sink* arguments
  - `T&&`, use a constraint when T is deduced

```
template <class T>
void f(T&&) requires std::is_rvalue_reference_v<T&&>;
```

# Argument Qualifiers

- *let* arguments
  - Postcondition: The argument is not modified
- *inout* arguments
  - Postcondition: The argument may be modified
- *sink* arguments
  - Postcondition: The argument is (assumed to be) consumed
  - The client can subsequently assign to the argument, or destruct it.

# A more complex action

```
// Offsets the value of `x` by `n`
// Precondition: `(x + n) < INT_MAX`

void offset(int& x, const int& n) {
  x += n;
}
```

- What if this is called as:

```
int x{2};
offset(x, x);

println("{}", x);
```

**4**

# A more complex action

```
// Offsets the value of `x` by `n`
// Precondition: `(x + n) < INT_MAX`

void offset(int& x, const int& n) {
  for (int i = 0; i != n; ++x) { }
}
```

- What will this print?

```
int x{2};
offset(x, x);

println("{}", x);
```

# A more complex action

```
vector a{ 0, 1, 1, 0 };

erase(a, a[0]);

println("{}", a);
```

- What will this print?

**[1, 0]**

– *https://godbolt.org/z/qM8Teos5h*

# Invalid References and References to Uninitialized Objects

```
vector<int> a{a};

terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Program terminated with signal: SIGSEGV
```

https://godbolt.org/z/6zqM8neax

# General Preconditions:

- Referenced objects must be within the objects lifetime
- inout and sink arguments cannot be accessed except directly by the implementation for the duration of the call
- let arguments passed by reference cannot be mutated for the duration of the call
  - Unless otherwise specified

# Swift Law of Exclusivity

To achieve memory safety, Swift requires exclusive access to a variable in order to modify that variable. In essence, a variable cannot be accessed via a different name for the duration in which the same variable is being modified as an `inout` argument or as `self` within a mutating method.

– *Swift 5 Exclusivity Enforcement*

# Rust Borrowing

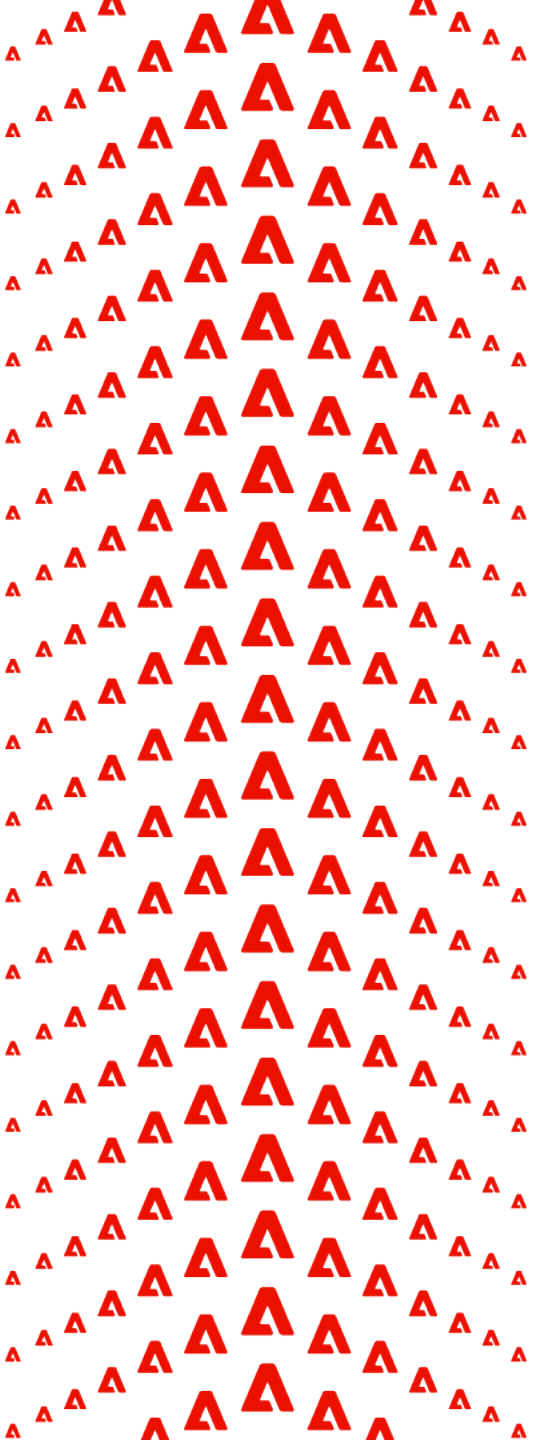
Mutable references have one big restriction: if you have a mutable reference to a value, you can have no other references to that value.

– *The Rust Programming Language: References and Borrowing*

# The Law of Exclusivity Applies to C++

- Upholding it is left as an execise for the developer

# Projections



# Function Results

```
// Returns the successor of `x`.
// Precondition: `x < INT_MAX`

int f(int x) { return x + 1; }
```

# Return-by-reference

```
vector a{0, 1, 2, 3};
a.back() = 42;

println("{}", a);
```

**[0, 1, 2, 42]**

# Projection Qualifiers

- Projections qualifiers mirror argument qualifiers
  - *Mutable* (T&) projections allows the projected objects to be modified
  - *Constant* (`const` T&) projections do not allow the projected object to be modified
  - *Consumable* (T&&) projections allow the projected objects to be consumed

# Projection Qualifiers

- Returning consumable projections are uncommon
  - Usually return by-value is used but consumables may be more efficient when extracting a value from an rvalue:

```
T&& extract() &&;
```

  - Mutable projections may also be consumed but require an additional operation to restore invariants on the owning object. i.e.

```
auto e{std::move(a.back());}
a.pop_back(); // erase the moved-from object
```

# Projection Validity

- A projection is invalidated when:
  - The object they are projected from is modified other than through the projection.

```
vector a{0};
int& p{a[0]};   // p is a projection
a.push_back(1); // p is invalidated
```

# Projection Validity

- A projection is invalidated when:
  - The object they are projected from is modified other than through the projection or through another non-overlapping projection

```
vector a{0, 1, 2, 3};
const e& = a.back();
a.clear(); // invalidates e
```

```
vector a{0, 1, 2, 3};
const e& = a.back();
a[2] = 42; // e is not invalidated
```

  - The lifetime of the object they are projected from ends

```
int& p{vector{0}[0]}; // p is invalidated right after creation!
```

# Projecting Multiple Values

- Iterator pairs, views, and spans project a collection of values from an object
- They follow the same rules as reference projections

```
vector a{3, 2, 1, 0};
copy(begin(a), begin(a) + 2, begin(a) + 1); // Invalid – overlapping

vector a{3, 2, 1, 0};
copy(begin(a), begin(a) + 2, begin(a) + 2); // OK – not overlapping
```

# Objects

# Objects

```
void f(shared_ptr<widget> p);
```

- What is the *type* of the argument for `f()`?

- To understand `f()` we need to understand the *extent* `p`

# Equational Reasoning

- *Equational reasoning* is proving that expressions are equal by substituting equals for equals.
- Equational reasoning explains how code works and is a component part of larger proofs.

- To know if two values are equal, we need to know the *extent* of the values.

# Equality

- *Equality* is an equivalence relation (reflexive, symmetric, and transitive)
- Equality connects to *copy* (equal and disjoint)

# Transformations and Actions

There is a duality between transformations and the corresponding actions: An action is defined in terms of a transformation, and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

*– Elements of Programming, Section 2.5*

# Composite Objects and Whole-Part Relationships

- A *composite object* is made up of other objects, called its *parts*.
- The whole–part relationship satisfies the four properties of *connectedness*, *noncircularity*, *disjointness*, and *ownership*

```
vector a{ 0, 1, 2, 3 };

struct {
  string name{ "John" };
  int id{0}
} b;
```

# Objects

```
void f(widget& p);
```

- This should only modify an instance of `widget`
- It should be possible to express this as:

```
widget a(widget&& p);
```

# Objects, Copies, and Argument Independence

- Objects used as arguments must be independent under mutation to uphold the Law of Exclusivity.
- Copies are equal and logically disjoint.

# Achieving Independence

- No mutation
- No sharing
- Copy-on-write (no mutation unless not shared)

# Extending Independence with Mutation

- A mutable object may extend permission for mutation to its parts through projections
  - So long as those projections do not overlap

# whole/part examples

```
class whole {
    part _part;
public:
    whole() = delete;
    explicit whole(state s) : _part{s} { }

    explicit whole(const whole&) = default;
    whole(whole&&) noexcept = default;

    whole& operator=(const whole&) = default;
    whole& operator=(whole&&) noexcept = default;

    bool operator==(const whole&) const = default;
};
```

# whole/part examples

```
class whole {
    shared_ptr<const part> _shared_part;
public:
    whole() = delete;
    explicit whole(state s) : _shared_part{make_shared<part>(s)} { }

    explicit whole(const whole&) = default;
    whole(whole&&) noexcept = default;

    whole& operator=(const whole&) = default;
    whole& operator=(whole&&) noexcept = default;

    // bool operator==(const whole&) const = default; // OK
    bool operator==(const whole& w) const {
        return *_shared_part == *w._shared_part;
    }
};
```

# whole/part examples

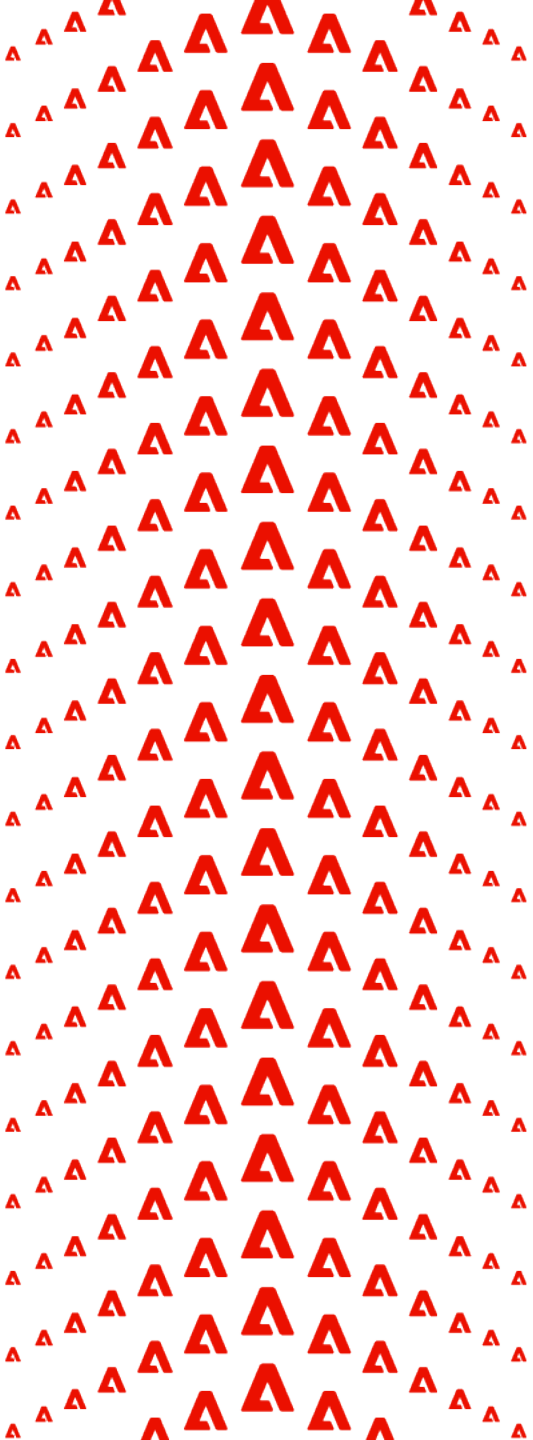
```
class whole {
    unique_ptr<part> _remote_part;
public:
    whole() = delete;
    explicit whole(state s) : _remote_part{make_unique<part>(s)} { }

    explicit whole(const whole& w) : _remote_part{make_unique<part>(*w._remote_part)} { }
    whole(whole&&) noexcept = default;

    whole& operator=(const whole& w) { return *this = whole{w}; }
    whole& operator=(whole&&) noexcept = default;

    // bool operator==(const whole&) const = default; // NOT OK
    bool operator==(const whole& w) const {
        return *_remote_part == *w._remote_part;
    }
};
```

# Extrinsic Relationships



# Extrinsic Relationships

- An *extrinsic relationship* is a relationship that is not a whole-part relationship

```
vector a{0, 1, 2, 3};
```

  - *`a[0]` is before `a[1]`* is an extrinsic relationship

# Relationships

- A relationship is a connection between elements of two sets
  - For every relationship, there is a corresponding binary predicate. i.e., `is_married(a, b)`
- A relationship between objects may be severed by modifying or destroying either object
- A relationship may be *witnessed* by an object such as a pointer or **index**
  - An object that is a witness to a severed relationship may be *invalid*

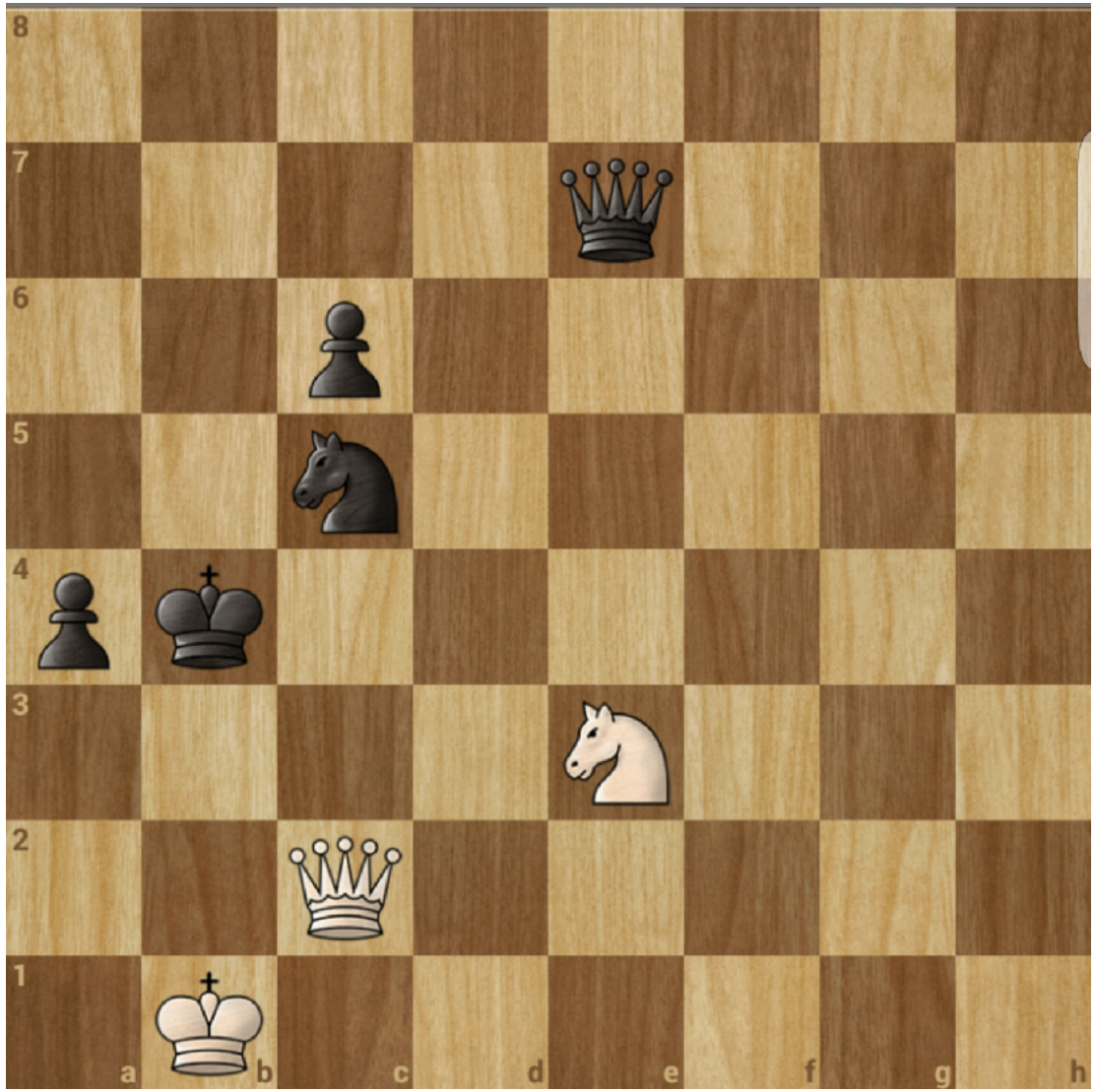
# You Have an Extrinsic Relationship If...

- Your class stores a non-owning pointer or any pointer that doesn't witness a whole/part relation.
- Your class stores a key or index.
- You reference a global variable.
- You use any synchronization primitive (mutex, atomic, etc.).

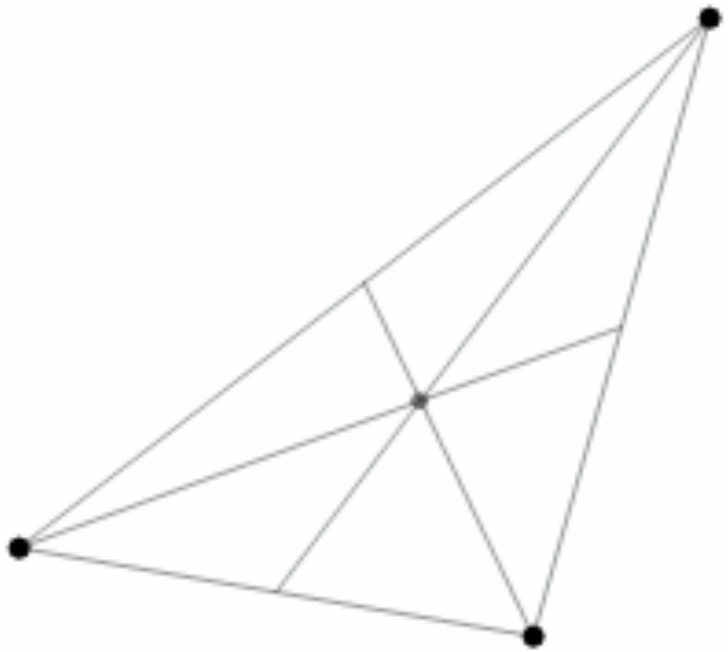
# Local Reasoning and Extrinsic Relationship

- To reason *locally* about extrinsic relationships they should be encapsulated into a class
- The relationships are maintained between parts by the class
- The class ensures the validity and correctness of the relationships by controlling access to the related objects
- An intrusive witness in a part should only be manipulated by the owning class, and explicitly severed if the object is moved or copied outside the whole
- Containers are examples of classes that manage extrinsic relationships between their parts

# An Analogy

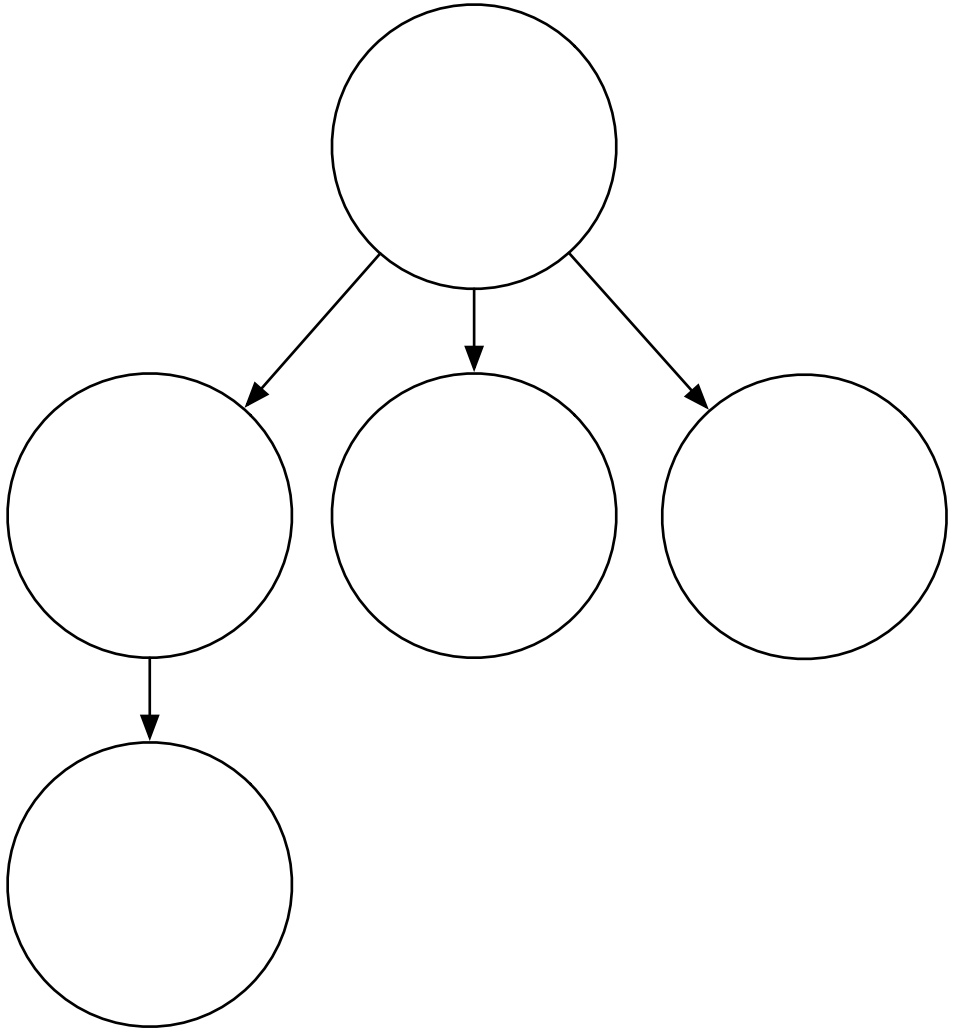


# Object Independence

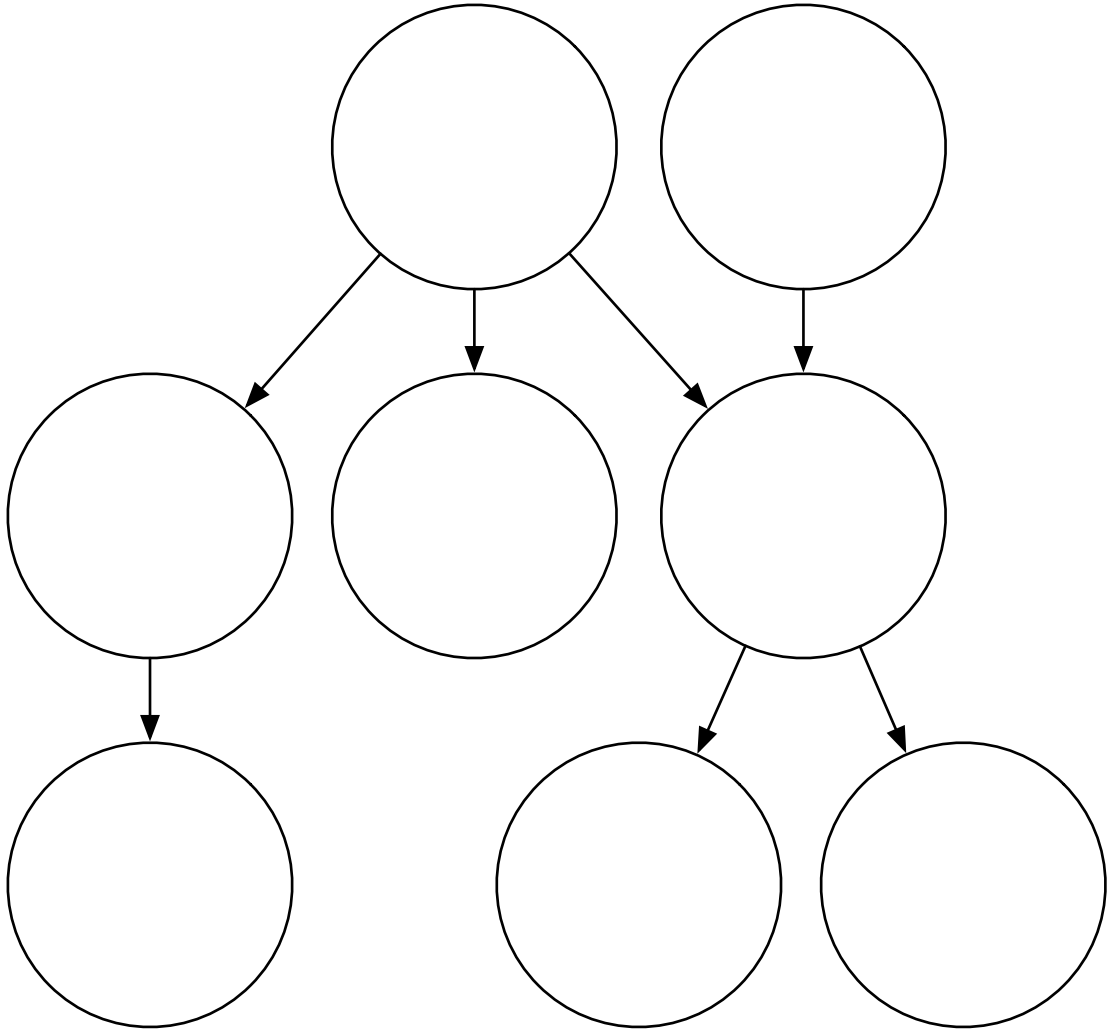


# Object Independence

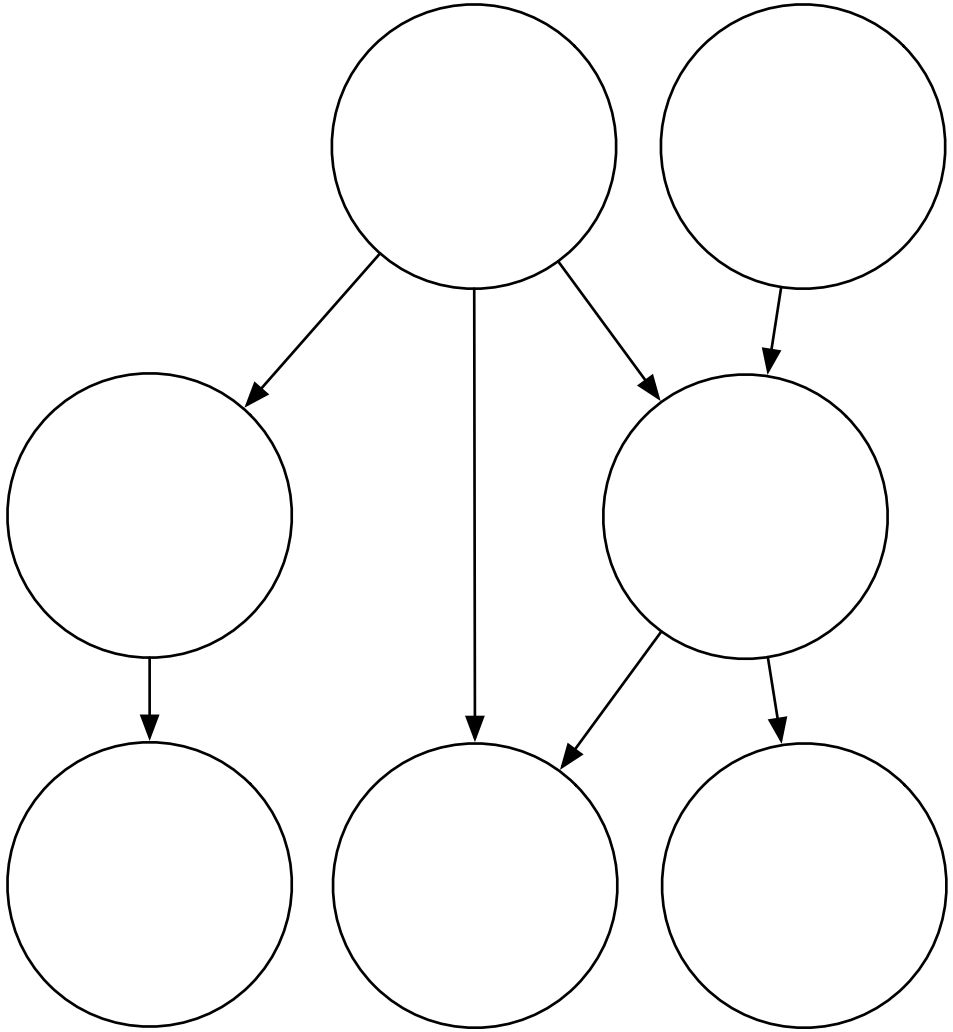
# Structural Complexity - Hierarchies and Trees



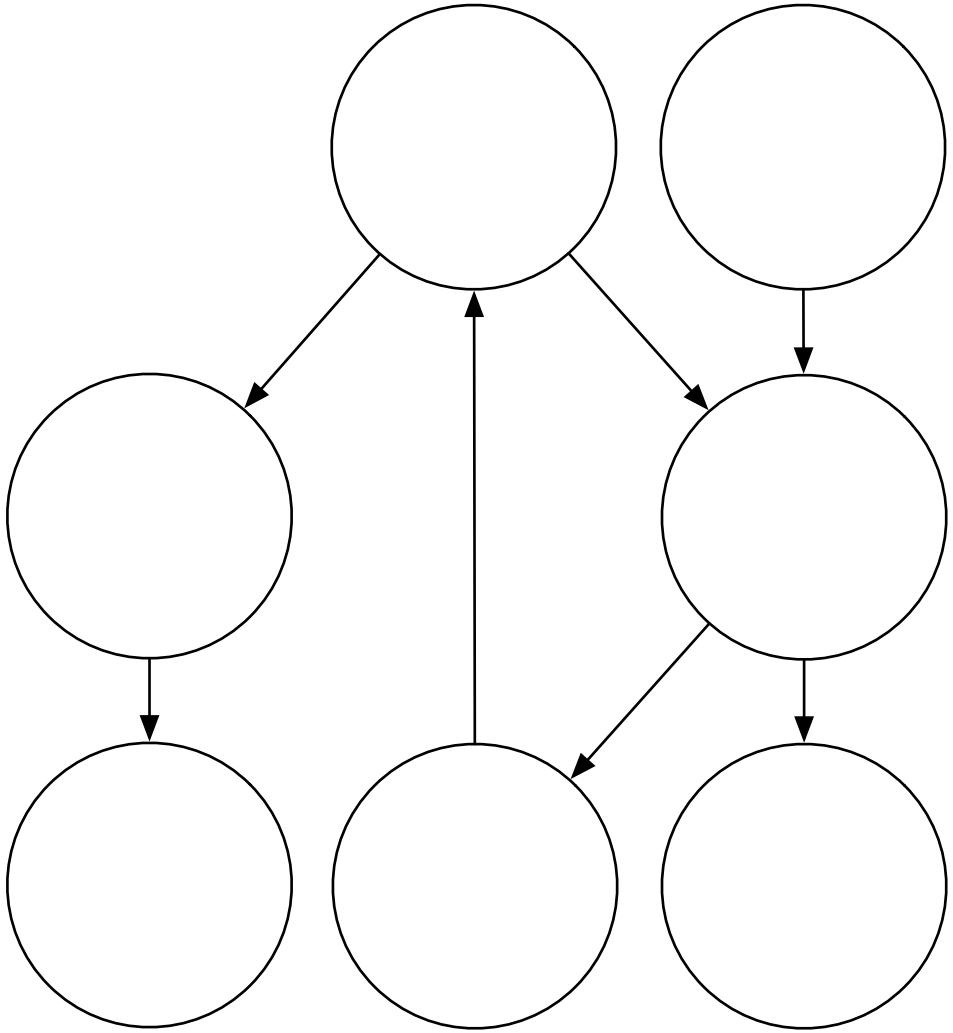
# Structural Complexity - Polytrees



# Structural Complexity - DAGs



# Structural Complexity - Directed Graphs



# Free Relationships

# Free relationships

- A *free relationship* is an extrinsic relationship that is not managed between parts of an object.
- If we assume local reasoning what meaningful structures can we build?

# CALM

"Question: What is the family of problems that can be consistently computed in a distributed fashion without coordination, and what problems lie outside that family?"

– *Keeping CALM: WhenDistributed Consistency is Easy*

# CALM

"A program has a consistent, coordination-free distributed implementation if and only if it is monotonic."

– *Keeping CALM: WhenDistributed Consistency is Easy*

# CALM

- Conflict-free replicated data types(CRDTs) provide a framework for monotonic programming patterns
- An immutable variable is a monotonic pattern that transitions from undefined to its final value and never returns. Immutable variables generalize to immutable data structures

# Russian Coat Check Algorithm

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

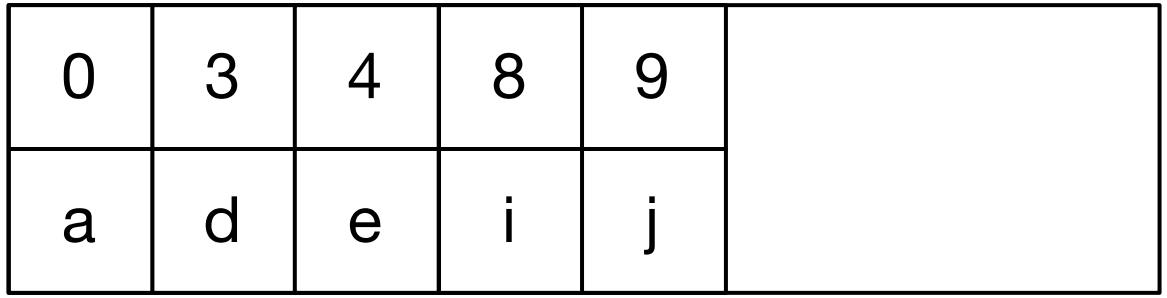
# Russian Coat Check Algorithm

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

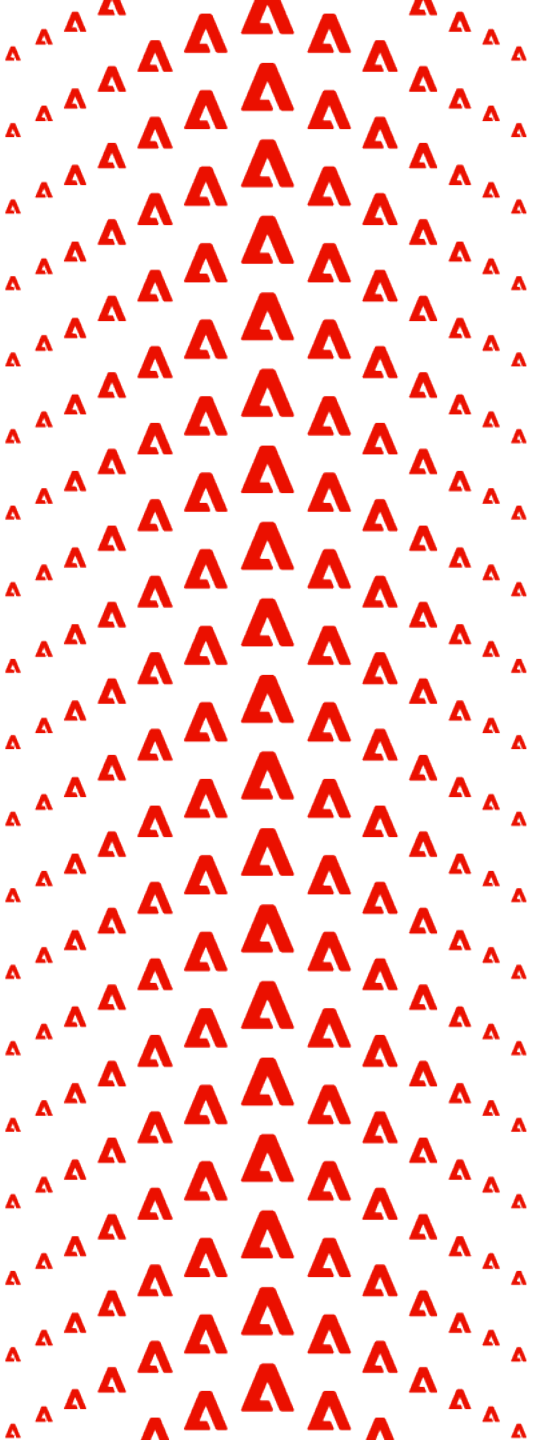
# Russian Coat Check Algorithm

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

# Russian Coat Check Algorithm



# Summary



# Existing Code

- Be conservative
- Avoid modifying shared data
  - If you don't know if it is shared, consider it immutable
- Avoid creating new sharing
  - Don't hold a member by a shared reference if you didn't create it
- If dealing with reference semantics
  - Make it clear if you are returning a new object or a reference to an existing one
- Remember the power of preconditions and push responsibility to the caller

# Summary

- Interfaces should make the scope of the operation clear
- Projections provide an efficient way to achieve value semantics and manipulate parts
- It is the client's responsibility to uphold the Law of Exclusivity
  - Don't pass projections that overlap a mutable projection
- Implementors provide types with value semantics
- Confine extrinsic relationships between parts within a class
  - As the relationships between parts scale, seek a general solution

## About the artist

### Leandro Alzate

Berlin-based illustrator Leandro Alzate mixes bright color palettes and stylized characters in his fanciful work for editorial and advertising clients. He draws inspiration from observing the ways people interact, and combines that with his passion for architectural shapes and spaces. He created this piece for the German Ministry of Economy to encourage people to explore work-from-home career opportunities. Working with brushes and vector shapes, Alzate created this piece entirely in Adobe Photoshop.

Made with

Adobe Photoshop


Adobe
Artwork by Leandro Alzate