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

Adobe

Sean Parent | Sr. Principal Scientist
Software Technology Lab

The original idea for this talk was "Local Reasoning in Any Language" to document how I think about code, regardless of the language I'm programming in. The talk gets mired in the C++ details, so doing a set of languages seemed too much. Except for the details, the rules in this talk apply to all languages. I present here how I map these ideas into C++; it isn't the only mapping for C++, and if you program in a different language, figure out a set of conventions to map the ideas into that language.

Some Links

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

The first link has information on C++ careers at Adobe, links to our careers website, and videos of seminars my team gave (and in the future, there should be more contributions).

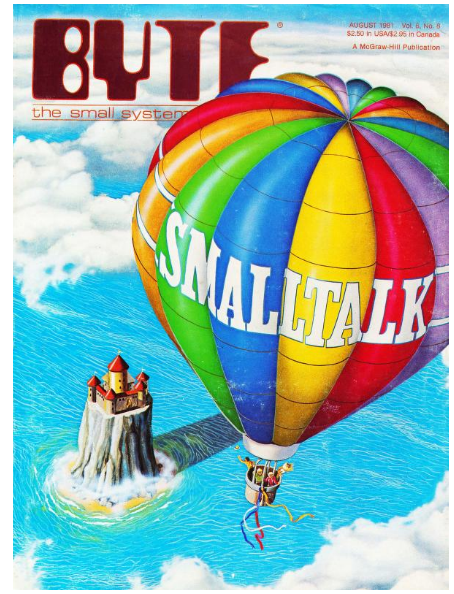
Adobe has development organizations in Hamburg and Bucharest, and almost 300 engineering positions are open, with about 35 in Europe.

The second link is to my vanity page - you'll find links to nearly all of my talks (at least all the ones I've been able to hunt down). This talk will be posted there soon.

The original idea for this talk was "Local Reasoning in Any Language" to document how I think about code, regardless of the language I'm programming in. The talk gets mired in the C++ details, so doing a set of languages seemed too much. Except for the details, the rules in this talk apply to all languages. I present here how I map these ideas into C++; it isn't the only mapping for C++, and if you program in a different language, figure out a set of conventions to map the ideas into that language.

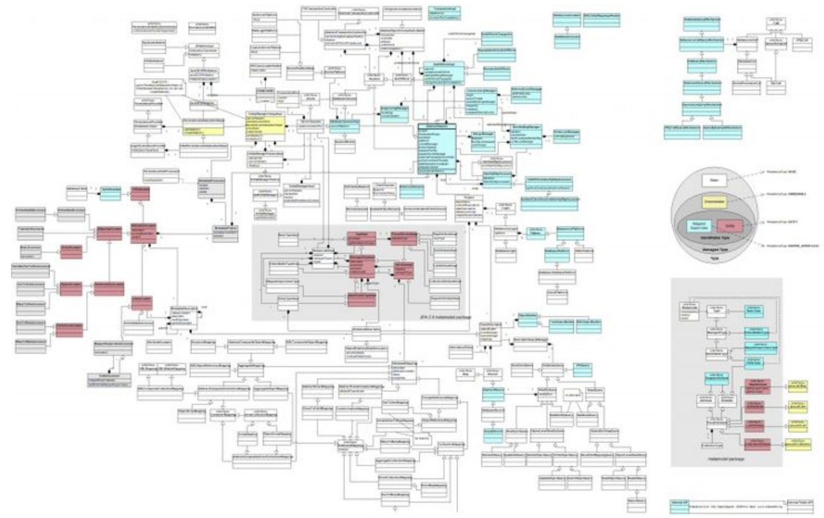
My team is trying to build a culture of correctness - how to engineers write correct code, not how do you go about proving code is correct after-the-fact.

Network of Object



In the 1980s and '90s, when object-oriented programming was in its heyday, the idea that we could build complex systems from large networks of objects was the rage. This is an issue of Byte magazine from 1981 - it had a massive influence on the entire industry and launched "Object Oriented Programming." Even though Smalltalk never gained widespread adoption, it was a very influential language.

Network of Object



The seductive idea was that, as long as each component had a well-defined interface, we could build complex systems by simply connecting them like Lego bricks. This is a UML diagram - does anyone here currently use UML? This is a relatively small system - but it fits on a slide. A collection of classes all interconnected, what could go wrong?

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.

NATO coined the term "software crisis" in 1968, and Dijkstra referenced it in his 1972 Turing Award Lecture.

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

Adobe

"Failed" here means failed to ship or shipped but was so riddled with defects it was scrapped or so late it was irrelevant.

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.

I'm sure someone will say, "If only these people knew about Agile and story points!"

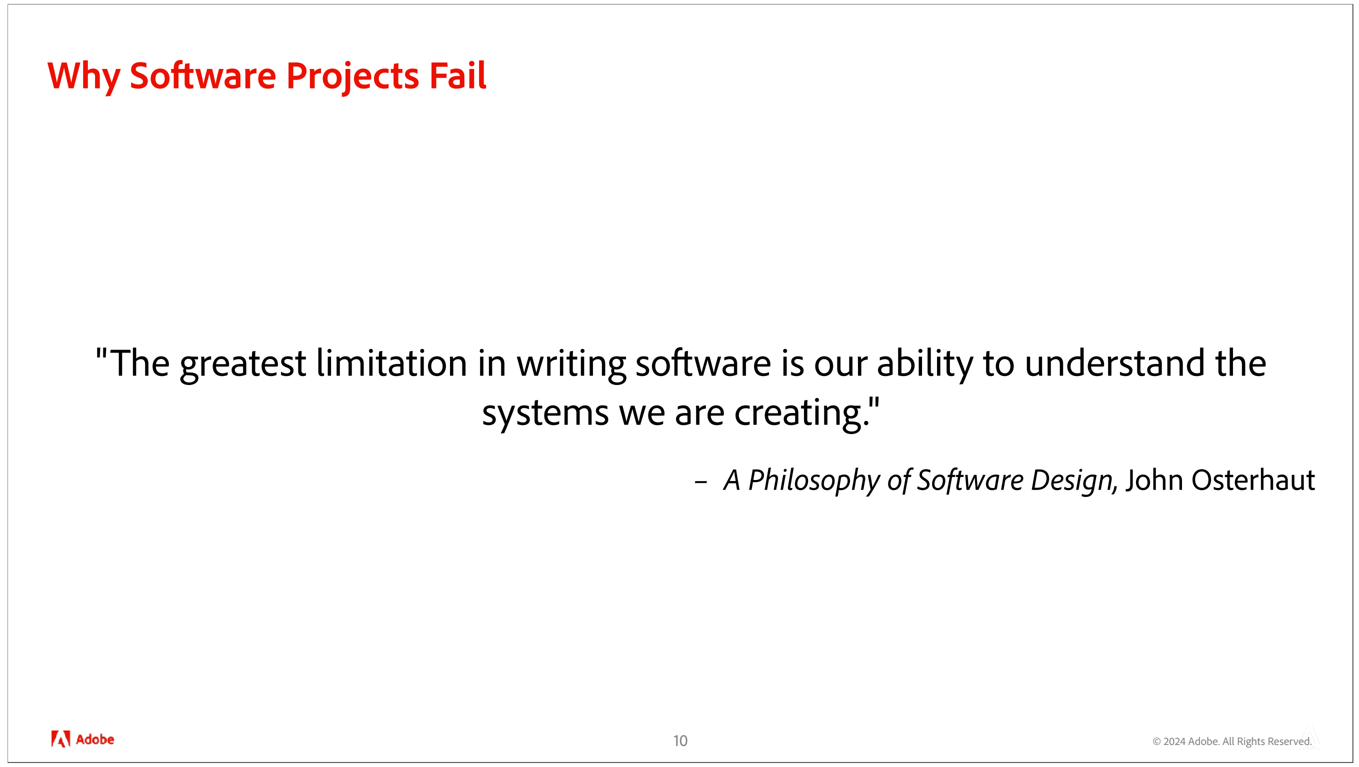
I can't find a reference that considers the _code_ or the system's design.

These failures are as much an engineering failure as a management failure. Are there any university students here today? If you want a thesis, write a report on the *engineering* failures for some of these projects.

Why Software Projects Fail

That is a big topic. I spent significant time looking for engineering analysis about why software projects fail. There are a _lot_ of management analyses, but I could not find a single paper with an engineering analysis. I will argue there is a specific point where engineering will, and often does, fail.

The failure occurs at the point where we lose the ability to reason _locally_ about code.



We already know the answer -

At some point, our ability to reason about the system breaks down. We can no longer understand what we are building, what effect a change will have, and we lose sight of the goal from the details in the code. Who here has worked on a software project that failed? Who has worked on a project where they felt lost? Uncertain of how the effect any change in the code will have. I work on a project that has over 50M lines of code… I can tell you, that the uneasiness is felt by the engineering team every day.

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

The solution is to construct systems that can be reasoned about locally. That is what this talk is about. I attended Patricia's workshop on insecure software at the start of this conference. I learned a lot. But I didn't learn how to write _correct_ code. Even formal methods focus on how you prove the code you write is correct - it doesn't focus on the construction of correct code. I want my team at Adobe to build a culture of correctness. Local reasoning is a key part of that.

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

I'll sometime use _caller_ and _callee_ when discussing functions, but client and implementor generalize to classes.

Functions

```
void f();
```


Let's start with a simple function interface. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we should document it. <click>
Now we can implement `f` <click>

Functions

```
// Does nothing.
void f();
```


Let's start with a simple function interface. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we should document it. <click>
Now we can implement `f` <click>

Functions

```
// Does nothing.
void f() { }
```

Adobe

I hope everyone is convinced that `f()` is implemented correctly. A requirement for local reasoning is a specification, a contract.

Suppose a piece of code has a contract, and everything it invokes also has one. In that case, we can read the implementation and verify that the function body is correct and fulfills the contract. But this isn't a talk about contracts; instead, it is about general principles for constructing code that is _simple_ to reason about and that we can verify.

Let's make our function a little more complicated<click>

Functions

```
// Returns the successor of `x`.
int f(int x) { return x + 1; }
```


Still very simple, this code is easy to understand at a glance. It doesn't have a great name—we'll get to that—but it does what the specification says. There is a simple precondition. If preconditions are satisfied and a function cannot satisfy the postconditions, the function should return an error… we can keep the code simple by just stating the precondition. <click>

Functions

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

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


Let's add a little more complexity <click>

Function Arguments

Function Arguments

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

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


This function is still simple; is it correct? We introduced a second precondition (maybe a third). What is it I'm looking for?

What if another thread is readying `x` when we update it? That would be a data race. There is an implicit precondition <click>

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; }
```


This precondition cannot be tested or verified by `a()`. The client must ensure it. By introducing indirection (passing the argument by reference), we raise the prospect of _aliasing_ in the interface, having more than one way to access an object. The rest of this talk is about techniques to control aliasing and confine the effect of an operation so it can be reasoned about locally.

[Make bigger point - and in summary - aliasing of mutable state is the whole thing...]

We certainly don't want to write preconditions like this with every function. So, instead, we're going to develop a set of general preconditions that must be upheld for all operations unless otherwise specified.

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

And now we can remove our precondition. Everyone already makes these assumptions - but they need to be written down. Otherwise, you forget the "unless otherwise specified" and it gets out of control.

Function Arguments

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

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


We don't normally pass an `int` by reference; we pass an object by reference as an optimization to avoid unnecessary copies. But for types where the cost of taking the reference is as much as passing the value, we pass the value. By convention, arithmetic types and pointers are passed by value. In generic code, iterators and invocable (function objects) are passed by value because they are likely small and trivial.

Why Mutation?

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

How would I replace the battery in my car, compared to rebuilding an equivalent vehicle with a new battery?

In situ operations are more space-efficient and may be more time-efficient.

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);
```


Let's define a couple of more terms.

An action is sometimes referred to as a functional update.

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
```

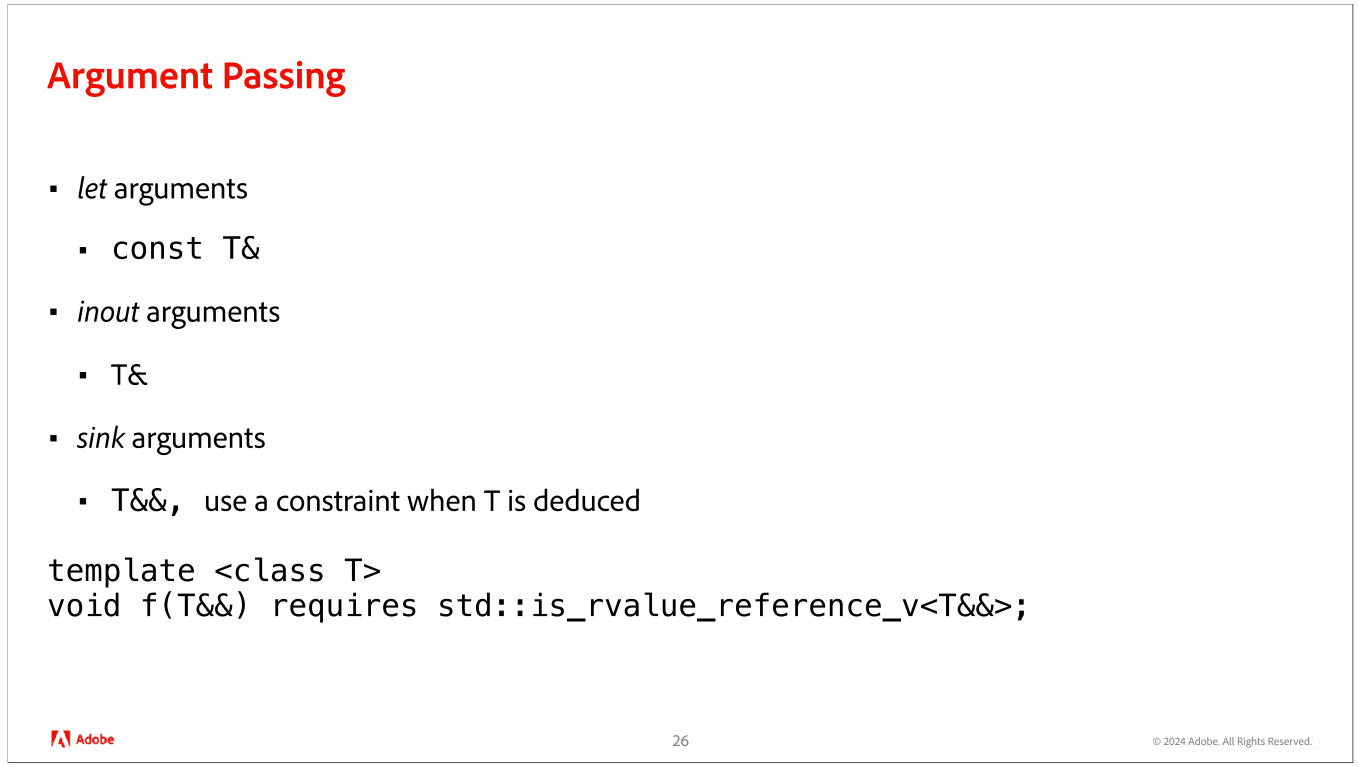
– Elements of Programming, Section 2.5

For a given operation, either an action or transformation may be a more efficient implementation. All other things being equal prefer transformations.

Because of the duality - you only need to write one. But the nature of the operation may tell you which one to implement. And we'll see later that sometimes there is value in writing both forms.

sort is an example that is more efficient in situ, and partition is more efficient as a transformation.

The transformation here is taking the argument by value, but we need to say a little more about passing arguments <click>



I'm borrowing the argument terminology from Hylo because C++ doesn't have a good nomenclature for this.

[regarding if T is deduced] We want the sink to be a non_const rvalue reference, I leave it as an exercise to write is_sink_v constraint. Unfortunately, I don't see a way to do it as a concept where you could say `auto sink a`

Consider `!std::is_reference_v<T> && !std::is_const_v<T>`.

```
template <class T>
inline constexpr bool is_sink_v{std::is_const_v<std::remove_reference_t<T>> && 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.

We want each of these to behave like the corresponding transformation form was used. We already found we cannot alias the value across threads. Are there other preconditions?

Regarding sink - In C++ you still have a named object if it was an lvalue.

sink arguments are used when the argument is escaped - either stored or returned, possibly with modification.

Pass by value is a let argument from the caller side, and consumable (sink) by the implementor. For small (<= sizeof(void*)) basic types (move and copy are equivalent) pass by value is used.

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

Will this print `4`, or `2`, or something else? We can see from the implementation that the answer is `4`. This is breaking the client contract that the second argument is not modified. The postconditions conflict - a contradiction. But maybe this is what we "expected." But what if offset was implemented this way <click>

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);
```

Adobe

the print statement is never reached. Because of the aliasing between arguments, where one is under mutation, the implementation cannot satisfy either postcondition.

This may seem like a contrived example. But here is a real one <click>

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

What will this print… It depends on the implementation, but here is one answer<click>

Why? After the code removes the first element matching a[0] (0), a[0] holds a 1, so the remaining 1s are removed, leaving the trailing 0. According to the standard, the answer is unspecified.

If arguments are aliased with mutation, local reasoning is broken for both the client and implementor.

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

We do have to say one more thing about references in C++. What does this line of code do?


NOBODY EXPECTS
SELF-INITIALIZATION
imgflip.com

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

I want to say that initializing a reference from an object outside of its lifetime (either before construction or after destruction) was forbidden, but it is not.

These preconditions appear in various forms in several [modern? safe?] languages

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*

In Swift, this is known as "The Law of Exclusivity", a term coined by John McCall.

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*

In Rust, the borrow checker enforces this restriction. C++ does not have such a restriction. We must rely on conventions and diligence.

The Law of Exclusivity Applies to C++

- Upholding it is left as an execise for the developer

Projections

We haven't talked about function results yet. So let's start our discussion of projections there…

Function Results

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

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


Let's go back to an early simple function. Here, we are returning a new value. Would it ever make sense to return a reference from a function?

Return-by-reference

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

println("{}", a);

[0, 1, 2, 42]
```


vector::back() is an example of returning a reference. There are many examples of returning references in the standard library, all assignment operators, indexing, and the min and max algorithms (by const reference, unfortunately)…

When we return a reference to a _part_ of something (and the whole is a part of the whole), we refer to it as a _projection_.

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

The fact that projection qualifiers mirror argument qualifiers is not a coincidence -
By reference arguments _are_ projections.

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
```


These are the general rules, a specific operation my provide stronger guarantees. It is the client responsibility to only pass valid projections to an operation

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!
```

Adobe

These are the general rules, a specific operation my provide stronger guarantees. Unless otherwise specified.

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
```


The copy algorithm has specific rules about overlapping ranges and copying to the left - as with our other rules there is an "unless otherwise specified" clause. If you rely on "otherwise specified" behavior - note it in a comment with a link to the documentation.

Objects

Argument independence allows us to reason about a function in isolation, but for that to work, our objects must be independent.

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`

This seems like a ridiculous question - of course, the type is a shared widget pointer!
It could be a let or sink argument since it is pass by-value…
Do you think `f` is just operating on the pointer?
Maybe the type of the argument is the widget. And the widget is mutable so this could be an inout widget argument. It could be a nullptr, so it could be an optional inout widget argument!

f has exclusive access to the pointer (pass by-value). Am I confident f has exclusive mutable access to the widget for the duration of the call? Maybe the widget contains other child widgets held as shared pointers.

Why is the extent important?

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.

Quick refresher on equality -

Equality

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

Equality also connects to move…
Recall the duality between transformations and actions -

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*

This is an example of equational reasoning. Projections are a proxy for a value, with rules governing the validity of the proxy.

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;
```


a is a composite object with 4 integer part
b is a composite object with two named parts

disjointness - logically disjoint under mutation from other wholes - not necessarily other parts. The whole is responsible for not providing projections that alias. immutable and copy-on-write objects may share storage.

Pointers, shared, unique or otherwise, witness a relationship. Which may, or may not, be a whole-part relationship. In an interface, their meaning is ambiguous and they are best avoided. Alone, they are disconnected from any whole.

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);
```


I prefer the sink/return-by-value form over mutation. It also allows for more concise code

But we need to talk a little about non-whole part relationships

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.

[Say more here… To reason locally…]

Achieving Independence

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

Adobe

No mutation is the functional programming approach.

Swift relies heavily on copy-on-write. It uses function bundles to associate a transform and action. When modifying an object, if the object is uniquely owned, the action is used. Otherwise, the transform is used, and the "copy" is free.

For example, if we have a copy-on-write dynamic array that is shared, and we insert an element, instead of copying, then inserting, we copy up to the insertion point, then move in the elements to be inserted, the copy the remaining elements.

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

In Rust this is down with ownership and borrowing, in Hylo this is done through whole/part relationships and projections. The advantage is less annotation.

In C++, the developer must manage projections.

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;
};
```


This is the canonical template for a class. Well-behaved parts compose into well-behaved wholes. Copy and equality are part-wise. We don't need a default constructor unless there is a meaningful default value. It is nice that now we get != for free.

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;
    }
};
```

Adobe

Here the part-wise equality is okay - representational equality implies equality.

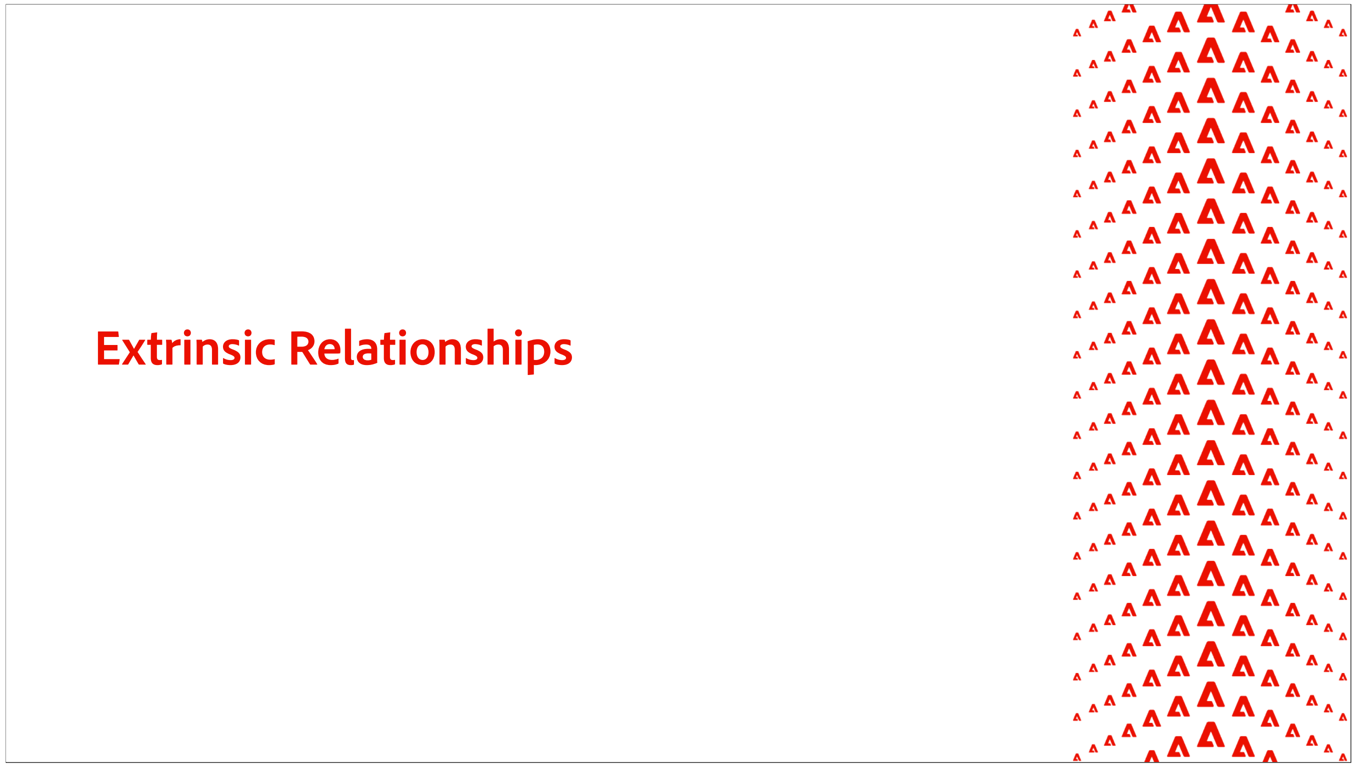
whole/part examples

```
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;
    }
};
```

Adobe

We can't use the default equality because it violates the axioms for copy - copies are equal.
If we use unique_ptr we need to implement copy explicitly.

There is a standard proposal for indirect_value that would encapsulate this. One could also imagine a shared_value (const). The stlab libraries have a copy-on-write abstraction, but it is overdue for revisiting.



But there is more to a system than just a bunch of objects - the objects are often somehow *related*.

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 exist all over in the code - the main challenge in programming isn't in functions or classes, but in finding and managing the essential relationships.

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*

I'm emphasizing index - sometimes memory-safe or functional languages are described as solving the problems with pointers. They only solve the memory-safety problems, not correctness (because correctness doesn't compose), and surprisingly (in the case of functional languages), not the problem of local reasoning.

In any Turing complete language, you can build a C machine and write buggy C code.

If I have an index to the largest element of an array, and I change the element such that it is no longer the largest, my index, as a witness to the relationship, is invalid.

This is the severing of relationships and invalidation of witnesses that we describe as "spooky action at a distance."

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

Class invariants are extrinsic relationships on or between parts that always hold (for some definition of always).

Private access is used to protect the class invariants.

By encapsulating, we mean managing all the elements involved in a relationship. Those elements are _parts_.

"explicitly severed," such as by nulling a pointer or optional, or assigning a sentinel value such as a negative index to represent severed.

Linked list example. Splicing doesn't entangle lists—a container view of the world.

I started this talk by claiming that software projects fail because local reasoning breaks down. Failing to manage extrinsic relationships is where that happens.

This is why it is essential to avoid creating _potential_ extrinsic relationships where a whole/part relationship would suffice. This is why we don't want incidental data structures, but we want data structures encapsulated in a class, so we can reason about the relationships locally.

Even if we are well principled in doing this. If every component we write has a well-defined contract. Extrinsic relationships are _hard_. Computer scientists are bad at relationships.

To illustrate - consider this chess board. Not any chess board or chess game, just _this_ board. There are four distinct pieces. King, queen, knight, pawn. Seven classes if we encode color in the class. We can represent these as very simple types.

Chess pieces do not have state. You might say "position is their state" - but that is false. The relationship is extrinsic to the part, even if I use an intrusive witness to represent the relationship.

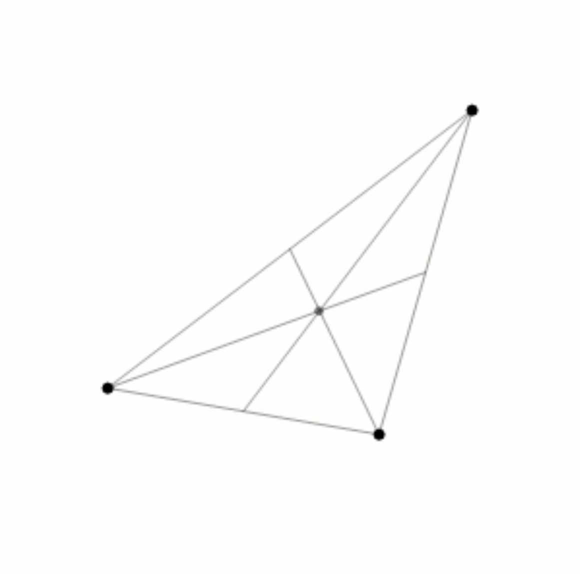
This is an end-game. We are not concerned with pawns moving 1 or 2 spaces, en passant, or castling. The board is only 8x8 - just 64 cells containing one eight pieces or nothing.

The relationship between the piece and the board is explicit. But that creates relationships and potential relationships between the pieces. Is another piece along a line of immediate attack? How do those relationships change if I move a piece. Is a king in check? In mate?

Even with this simple example I cannot reason through the relationship considering every possible move independently. There are just 30 allowed moves for white to take _next_ ruling out invalid moves, and about 30 responses to consider for each...

Now consider a system with shared pointers to mutable objects state-owned by more than one class and multiple instances… If you think, "it isn't that hard" add concurrency.

Object Independence



Here we have three objects, and three relationships. All have well-defined contracts (the physics are known). The resulting system is chaotic.

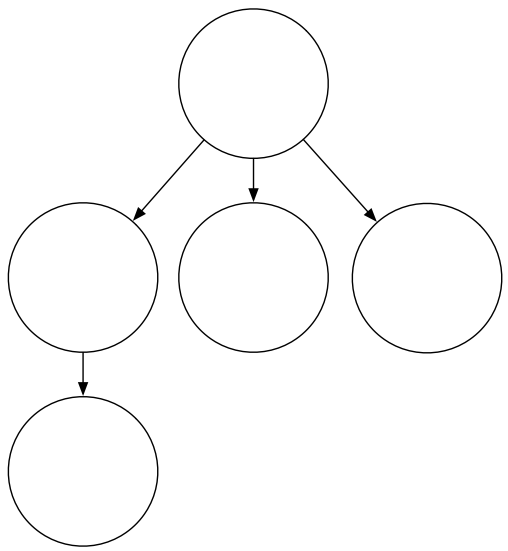
Object Independence

This is a Hénon map—a chaotic system with just two relationships.

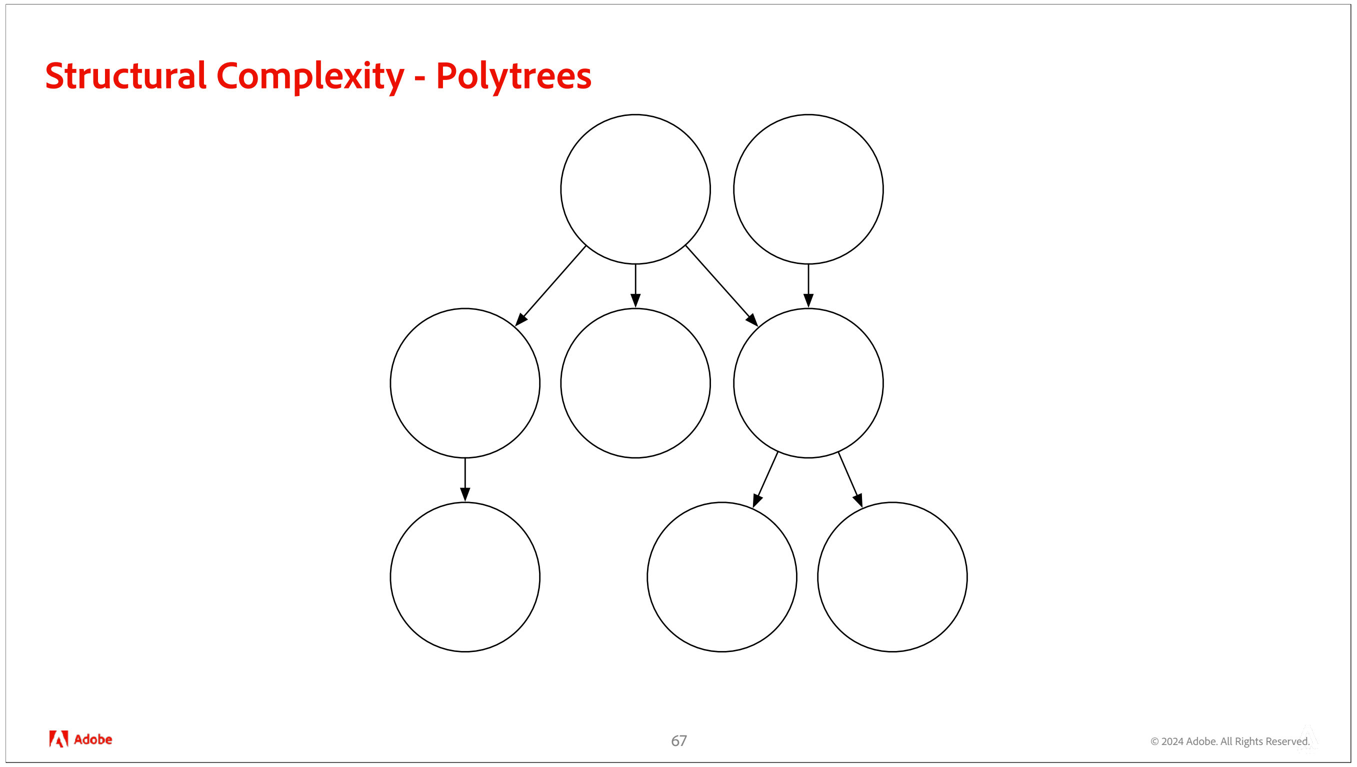
[- To build systems we can reason about, we need to:
- make whole/part (strict hierarchical) relationships explicit in the code so we don't have to worry about them.
- encapsulate extrinsic relationships in an explicit class
- limit extrinsic relationships to a _small number_ (two might be too many if cyclical), up to about 12 if heterogeneous and acyclic.
- Larger numbers of relationships must be homogenous and solved algorithmically.]

Chaotic software - a small change in input leads to a dramatic change in output.

Structural Complexity - Hierarchies and Trees

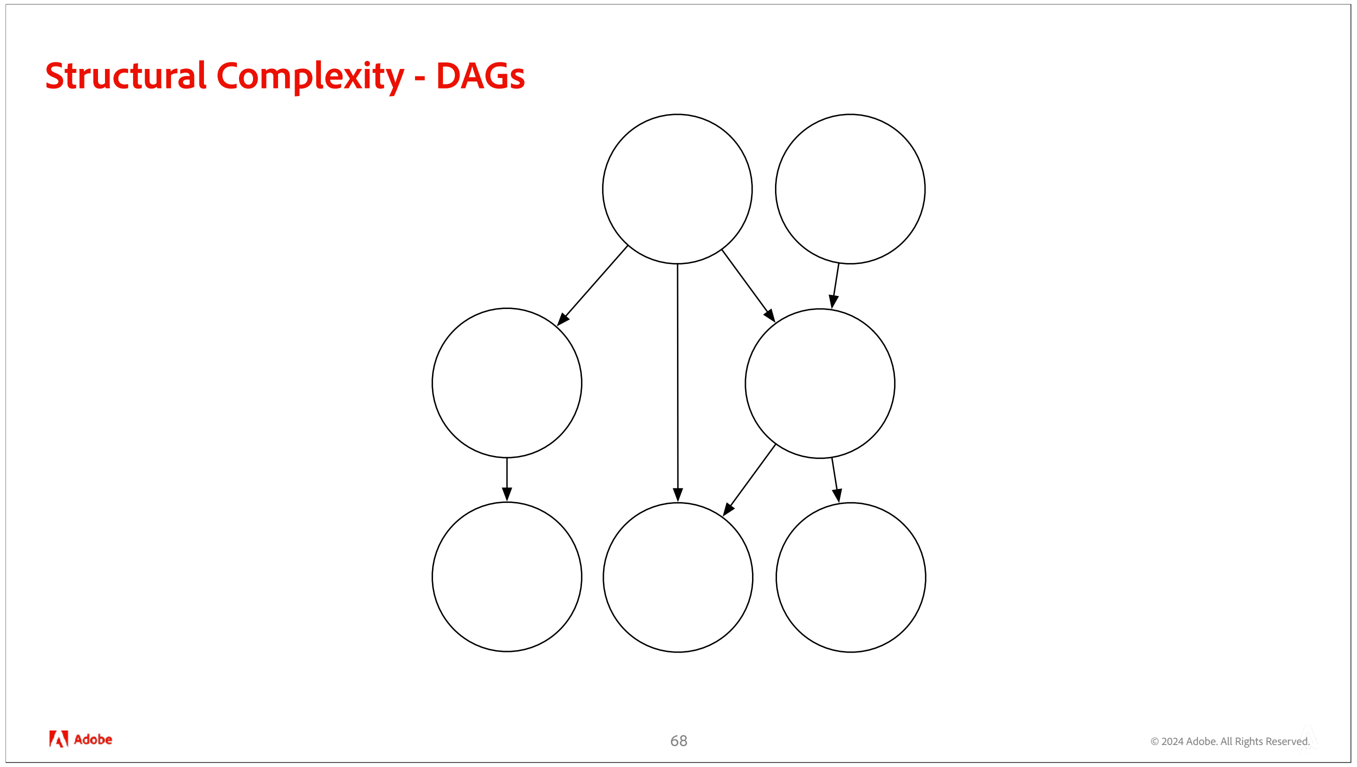


This is the whole/part relationship - little can go wrong here. Projections allow us to reason about local hierarchical structure.

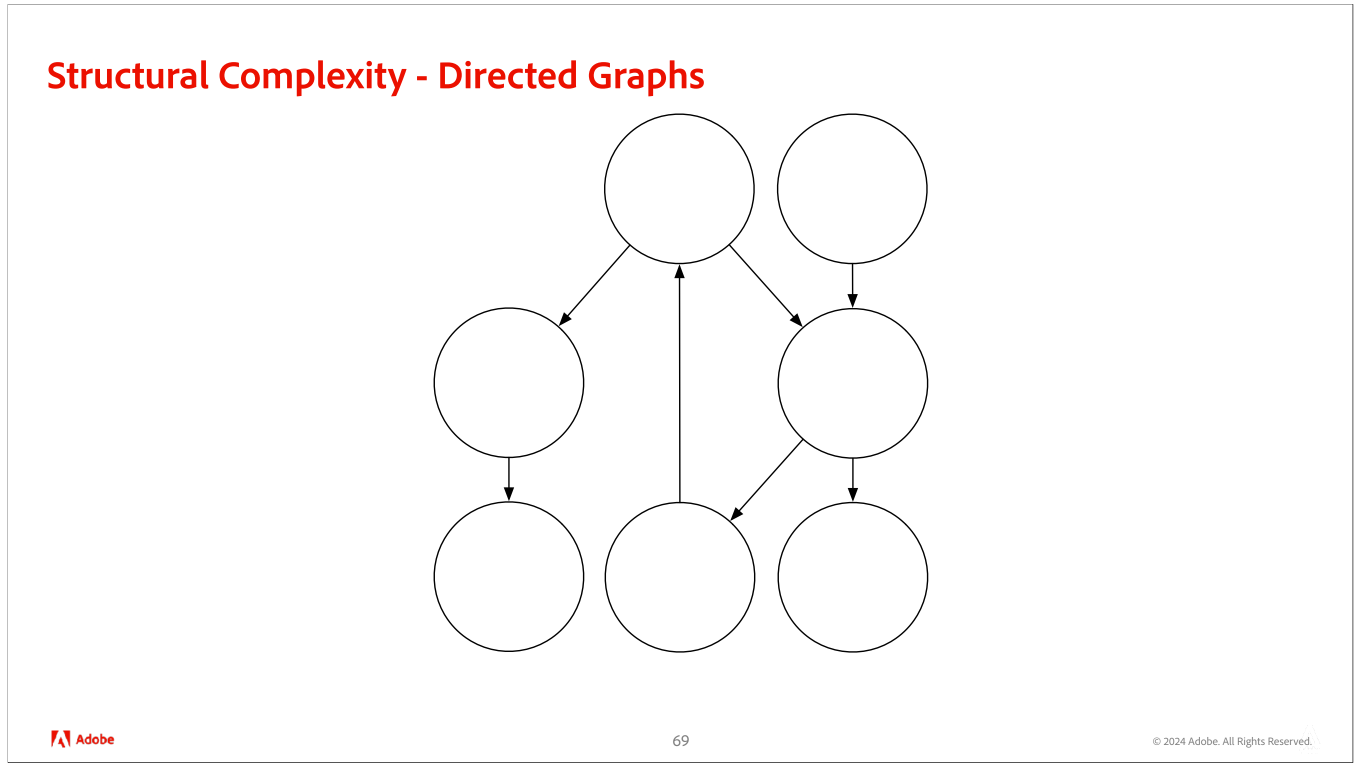


Joins must be managed - this is the structure of race conditions and LOE violations. Joins are potential contradictions. Joins should be explicit and managed (last-one-in wins is almost always a bad join).

This isn't "bad" just more complex. Isolate polytrees between parts of a class.



DAGs can rejoin - if not directed, they would contain cycles. They introduce consistency concerns (is the data calculated on this path consistent with this other path). The s-combinator is a diamond-shaped relationship. S-combinators allow us to build a system effectively Turing complete without iteration or recursion.



Cycles should be factored out and replaced with a single node. "No raw loops" includes structural loops. Reasoning about cycles requires proving termination, convergence, loop invariants, limits and gates.

Chess boards are filled with essential cyclic relationships - the queen is related to the king and the king to the queen. Chaotic systems are cyclic relationships. A chaotic node in your graph means you cannot predict the output from the input unless you know how exactly how you arrived at the current state. Bugs that don't reproduce are chaotic.

Free Relationships

In the 80s and into the 90s, there was a view that you could build systems at scale consisting of networks of objects. The entire OOP ethos was built around this idea. The view was always flawed but persists in reference-semantic languages.

The 80s programmers were the hippies from the 60s and 70s. Free the objects!

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?

We only have local knowledge of each object, which follows a set of rules.

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*

This paper is from 2019. Relatively recent.

CALM

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

– *Keeping CALM: WhenDistributed Consistency is Easy*

Consistency As Logical Monotonicity (CALM).

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

Immutable globals are okay. They don't require any additional coordination. A monotonic system can never repeat its state. This is related to the class ABA problem in concurrent programming.

In 2008, I gave a Google tech talk on a possible future of software development. I conjectured that _at_ some scale, we require coordination-free computation. That scale is determined by the latency required for coordination. Significant progress has been made in recent years in this space (see CRDT and operational transforms), but many open issues remain. But we now know the bounds within which we are working.

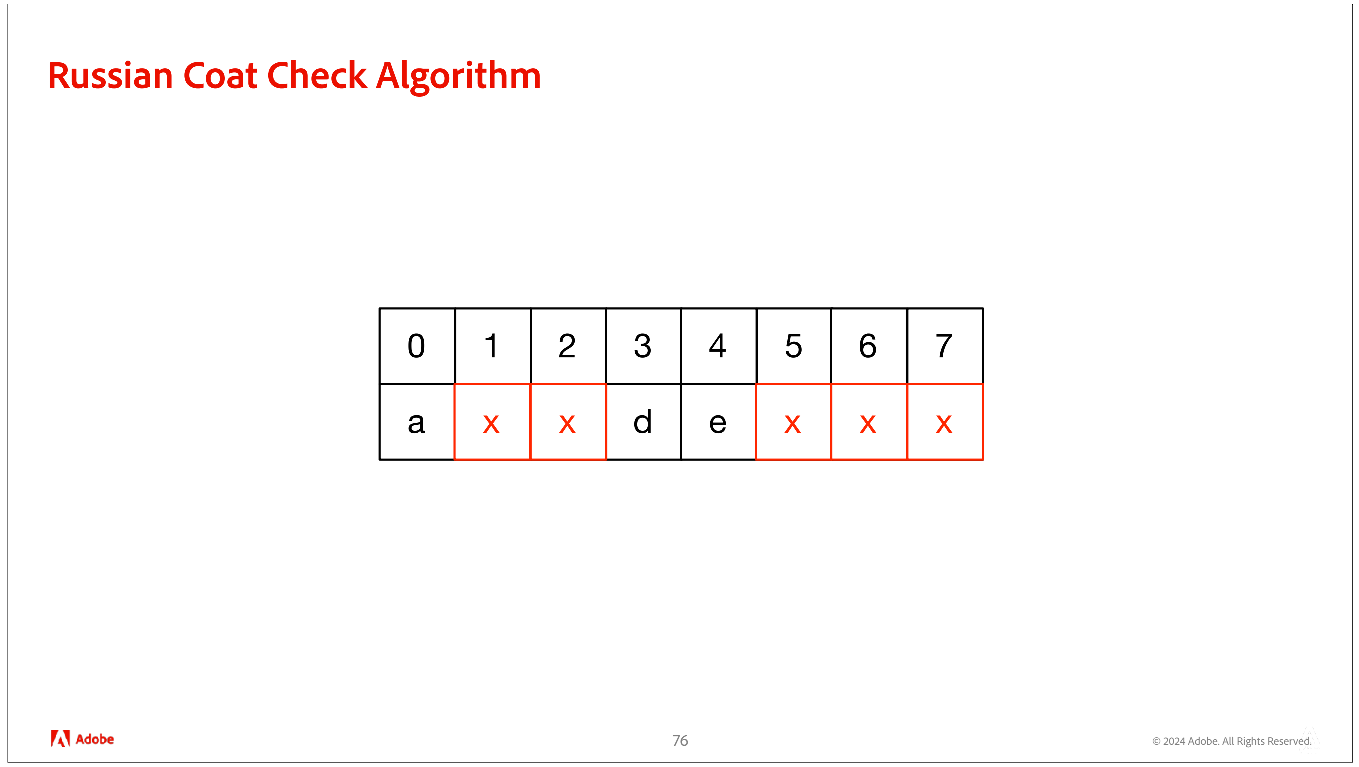
Russian Coat Check Algorithm

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

In my relationships talk I presented this structure. The Russian Coat Check Algorithm (so named because I thought of it while watching how a woman managed coats at a coat check in Russia and as a nod to the Russian Peasant Algorithm - aka Egyptian Multiplication).

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]



In my relationships talk I presented this structure. The Russian Coat Check Algorithm (so named because I thought of it while watching how a woman managed coats at a coat check in Russia and as a nod to the Russian Peasant Algorithm - aka Egyptian Multiplication).

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

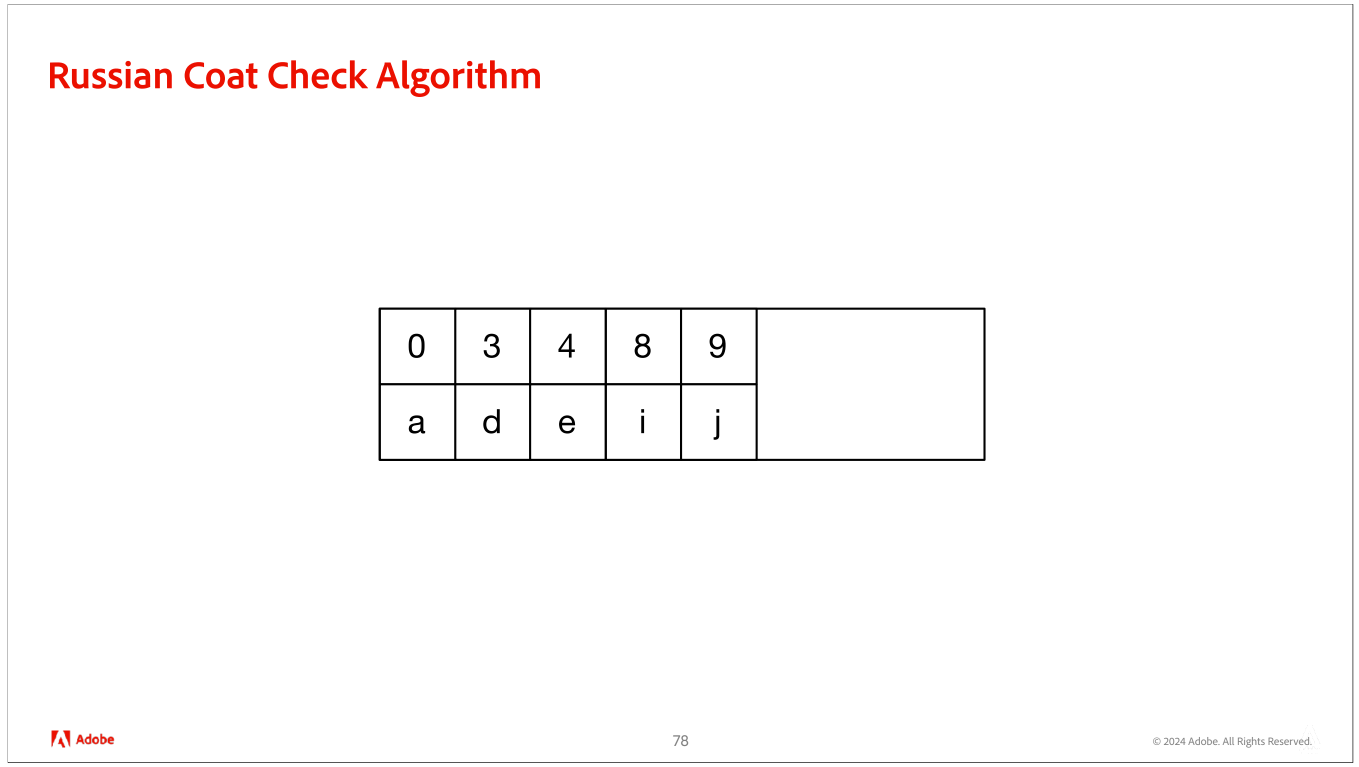
[every add is a wide gap]

Russian Coat Check Algorithm

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

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]



This structure is monotonic. It will never repeat the same state. A given element has 3 states and never cycle.

A Russian Coat Check can be safely shared without coordination.

- Has not been there
- Is there
- Was there

CALM is a tool to help you reason about what can be meaningfully shared and provides a framework for how to reason about objects required to be shared.

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
Bē
Artwork by Leandro Alzate