Keynote: Better Code: Relationships

Sean Parent
Better Code: Relationships
Sean Parent | Senior Principal Scientist, Photoshop
Goal: No Contradictions
“A novice sees only the chessmen. An amateur sees the board. A master sees the game.”

– Unknown
“Computer scientists are bad at relationships.”

– Me
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

\[
\{(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots\}
\]
Predicates

- A relation 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 a 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

\[ a \implies b \]

_(a implies b)_

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>( a \implies b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
A simple, but incomplete, notation

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

- Implication is represented with directional edges

- This is shorthand for *given entities b and c, a is any entity such that R holds*
- Read as, *b and c imply a*
Relationships and Objects

- As soon as we have two entities we have implicit relationships
- A memory space is an entity
- When an object is copied or moved, any relationship that object was involved in is either *maintained* or *severed* with respect to the destination object
- When an object is destructed, any relationship that object was involved in is *severed*
Witnessed Relationships

- A *witnessed* relationship is a relationship represented by an object
  - As an object, a witnessed relationship is copyable and equality comparable
  - When an object is copied or moved, any witnessed relationship that object was involved in is either maintained, severed, or *invalidated*.
  - When an object is destructed, any witnessed relationship that object was involved in is either severed, or *invalidated*.

- We may choose not to implement copy or move for witnessed relationships
- This is how we get iterator invalidation “at a distance”
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.
[ This slide intentionally left void ]
0100

4
0100

[-8..7]
4 > 3
\texttt{hash( )} \neq \texttt{hash( )}
Memory Space

0100 < 0100 0011
Memory Space
Safety

- An object instance, without meaning, is *invalid*
- An object in an invalid state, must either be restored to a valid state, or destroyed
- This is related to the idea of a *partially formed* object
- An operation which leaves an object in an invalid state is *unsafe*

- `std::move()` is an unsafe operation
C++20

- Two new features specifically about relationships
  - Concepts
  - Contracts
C++20

- Two new features specifically about relationships
  - Concepts
  - Contracts
Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

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

1998
“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*.\"
An Axiomatic Basis for Computer Programming

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

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it involves, for example, addition and multiplication of integers. Unfortunately, in several aspects 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 depicted in Table I are a rather small selection of axioms relevant to integers. From this incomplete set

It is interesting to note that the different systems satisfying axioms A1 to A5 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:

\[ \forall y \exists x \quad (y < x) \]

where all finite arithmetics satisfy:

\[ \forall x \quad (x \leq \text{max}) \]

where “\text{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 \( \text{max} + 1 \):

\begin{align*}
A1a & \quad \Rightarrow (x = \text{max} + 1) \quad \text{(strict interpretation)} \\
A1a & \quad \Rightarrow \text{max} + 1 = \text{max} \quad \text{(firm boundary)} \\
A1a & \quad \Rightarrow \text{max} + 1 = 0 \quad \text{(modulo arithmetic)}
\end{align*}

Having selected one of three axioms, it is possible to use it in defining the properties of programs; however,
Equality

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

\[(\forall a) a = a. \quad \text{(Reflexivity)}\]
\[(\forall a, b) a = b \Rightarrow b = a. \quad \text{(Symmetry)}\]
\[(\forall a, b, c) a = b \land b = c \Rightarrow a = c. \quad \text{(Transitivity)}\]
Concepts

- Axioms follow from the definition
- A collection of connected axioms form an *algebraic structure*
- Connected type requirements form a *concept*
Copy and Assignment

- Properties of copy and assignment:

\[ b \rightarrow a \Rightarrow a = b \]  
\[ a = b = c \land d \neq a, d \rightarrow a \Rightarrow a \neq b \land b = c \]  

(copies are equal)  
(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:

\[ (\forall a, b) \text{exactly one of the following holds:} \]
\[ a < b, b < a, \text{ or } a = b. \]  \hspace{2cm} \text{(Trichotomy)}
\[ (\forall a, b, c) a < b \land b < c \Rightarrow a < c. \]  \hspace{2cm} \text{(Transitivity)}
Natural Total Order

- Example: Integer $<$ is consistent with addition.

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

- Quantified axioms are (generally) not actionable
- Concepts in C++20 work by associating semantics with the name of an operation
Software is defined on Algebraic Structures
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 a strong claim about their benefi- cial 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 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
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 1 are a rather 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 A5 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, the following arithmetic satisfies the axiom:

\[ \forall x \forall y \left( x < y \right) \]

where all finite arithmetic satisfy:

\[ \forall x \left( x \leq \text{max} \right) \]

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:

\[ \forall x \left( x = \text{max} + 1 \right) \] (strict interpretation)

\[ \text{max} + 1 = \text{max} \] (firm boundary)

\[ \text{max} + 1 = 0 \] (modulo arithmetic)

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

1969
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* over the values being sorted
Pattern Matching

- Concepts are used as a compile time constraint to select an appropriate operation
- Contracts assert at runtime if an operations preconditions are not met

- A runtime constraint to select an appropriate operation is known as *pattern matching*

```c
void f(auto i) requires { !(i < 0) }
void f(int i) [[expects !(i < 0)]]
void f(int i) requires !(i < 0) // Not yet in C++...
```
Whole-Part Relationships and Composite Objects

- Connected
- Noncircular
- Logically Disjoint
- Owning

- Standard Containers are Composite Objects
- Composite objects allow us to reason about a collection of objects as a single entity
class view {
    std::list<std::shared_ptr<view>> _children;
    std::weak_ptr<adobe::view> _view;
    //...
};
// 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 = ex::rot(f, f + 1);
            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;
        }
    }
}
The Game

Architecture
Clients

Product

Dependencies
Architecture is the art and practice of designing and constructing structures.
Large structures are built by combining smaller structures which are built by combining even smaller structures.
Task

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

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

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

- After the application has been idle for at least \( n \) seconds do something

```cpp
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(when, [=]{ after_idle(task, delay); });
    } else {
        task();
    }
}
```
Visualizing the Relationships

- The structure, ignoring the recursion in `invoke_after`

```
_last_idle
now()
delay
when
[1]
task
```
Visualizing the Relationships

- The arguments and dependencies

- \_last\_idle
- now()
- delay
- when
- [1]
- task
Visualizing the Relationships

- Two operations

```
_last_idle

now()

delay

when

[1]

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

_last_idle

now()

when

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

T → remaining → S

remaining

S

task
On Expiration

T → remaining → S

task
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) {
    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); });
}
Architecture

- By looking at the structure of the function we can design a better function
- Note that on_expiration has no external dependencies
  - No `std::chrono`
  - No `std::function`
  - Or `invoke_after` or `last_idle`
- Requirements are the semantics of the operations and the relationship between arguments
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 objects in the registry

- Example: signal handler
template <class T>
class registry {
    unordered_map<
        size_t, id
    > public:
        auto append(
            _map.e,
        )
            return
    }
    void erase

    template <
    void for_e,
        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
- As an additional useful properties coats are always ordered by insertion
template <class T>
class registry {
    vector<pair<size_t, optional<T>>> _map;
    size_t _size = 0;
    size_t _id = 0;

public:
    //...

Russian Coat Check Algorithm

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

//...
```
Russian Coat Check Algorithm

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
</tbody>
</table>
```cpp
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 || !p->second) 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));
    }
    //...
}
Russian Coat Check Algorithm

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x</td>
<td>x</td>
<td>d</td>
<td>e</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Russian Coat Check Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>x</td>
<td>x</td>
<td>d</td>
<td>e</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Russian Coat Check Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
<td>e</td>
<td>i</td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>
template <typename F>
void for_each(F f) {
    for (const auto & e : _map) {
        if (e.second) f(*e.second);
    }
};
Russian Coat Check Algorithm

![Graph showing ratio (CPU time / Noop time) for reg_unordered_map and reg_vector. Lower is faster.]
Russian Coat Check Algorithm

![Bar Chart]

- **reg_unordered_map**: 3000
- **reg_vector**: 640
- **reg_for_each**: 0

**Ratio (CPU time / Noop time)**
Lower is faster
Russian Coat Check Algorithm

[Diagram showing a scatter plot with the x-axis labeled 'Elements' and the y-axis labeled 'Allocations'. A single data point is marked with a triangle.]
Architecture

- Relationships can be exploited for performance
- Understanding the relationship between the cost of operations is important
Goal: No Contradictions
Double-entry bookkeeping

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

\[ \text{assets} = \text{liabilities} + \text{equity} \]

- An example of equational reasoning
- 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 of the most powerful family dynasties in history
- Double-entry bookkeeping was codified by Luca Pacioli (the Father of Accounting) in 1494
Double-entry bookkeeping

- Every transaction is entered 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

- When two relationships imply the same entity has different values
- Relationships are consistent if they imply the same entity has the same value
Data Race

- When two or more threads access the same object concurrently and at least one is writing
Data Race

- We can resolve the race with a mutex
- But what does it mean?
No Raw Synchronization Primitives
Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

```cpp
p->member();  // SIGABRT
```

```
~*0  \  \\
   \|  \\
    \  \\
     UB
```

```
*0  \  \\
   \|  \\
    \  \\
     UB
```
Null Pointer Dereference

- C++ Specification: dereferencing a null pointer is *undefined behavior*

```
if (p) p->member();
```
Null Pointers or Optional Objects

- The graceful handling of *nothing* as a limit is important
- empty ranges, 0, etc.
- Removing sections of code to avoid a crash is likely only moving the contradiction
Pro Tip

- Use strong preconditions to move the issue to the caller

```c
void f(type* p) {
    //...
    if (p) p->member();
    //...
}
```
Pro Tip

- Use strong preconditions to move the issue to the caller

```c
void f(type& p) {
    //...
    p.member();
    //...
}
```
Setting a Property

- Two functions setting the same value through a shared pointer

```
p->set_property(value);
// Someplace else...
p->set_property(other_value);
```
Setting a Property

- Possible meanings:
  - Code is redundant
  - Different aspects of the same relationship, represented in disparate sections of code
    - `value` is `a * b` when `a` changes
    - `other_value` is `a * b` when `b` changes
  - Different, mutually exclusive, relationships with non-local control
  - Implied “last in wins” relationship
  - An incidental algorithm - property will converge to the correct value
  - Property is not a simple property but a stream, trigger, or latch
  - Or, it is just wrong
No Raw Pointers
Play the Game

- Consider the *essential* relationships
- Learn to see structure
- Architect code
sean-parent.stlab.cc
photoshopishiring.com