Exceptions the Other Way Around

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab

Parental Talk

Parental Talk™

Playing with exceptions

`errno()`

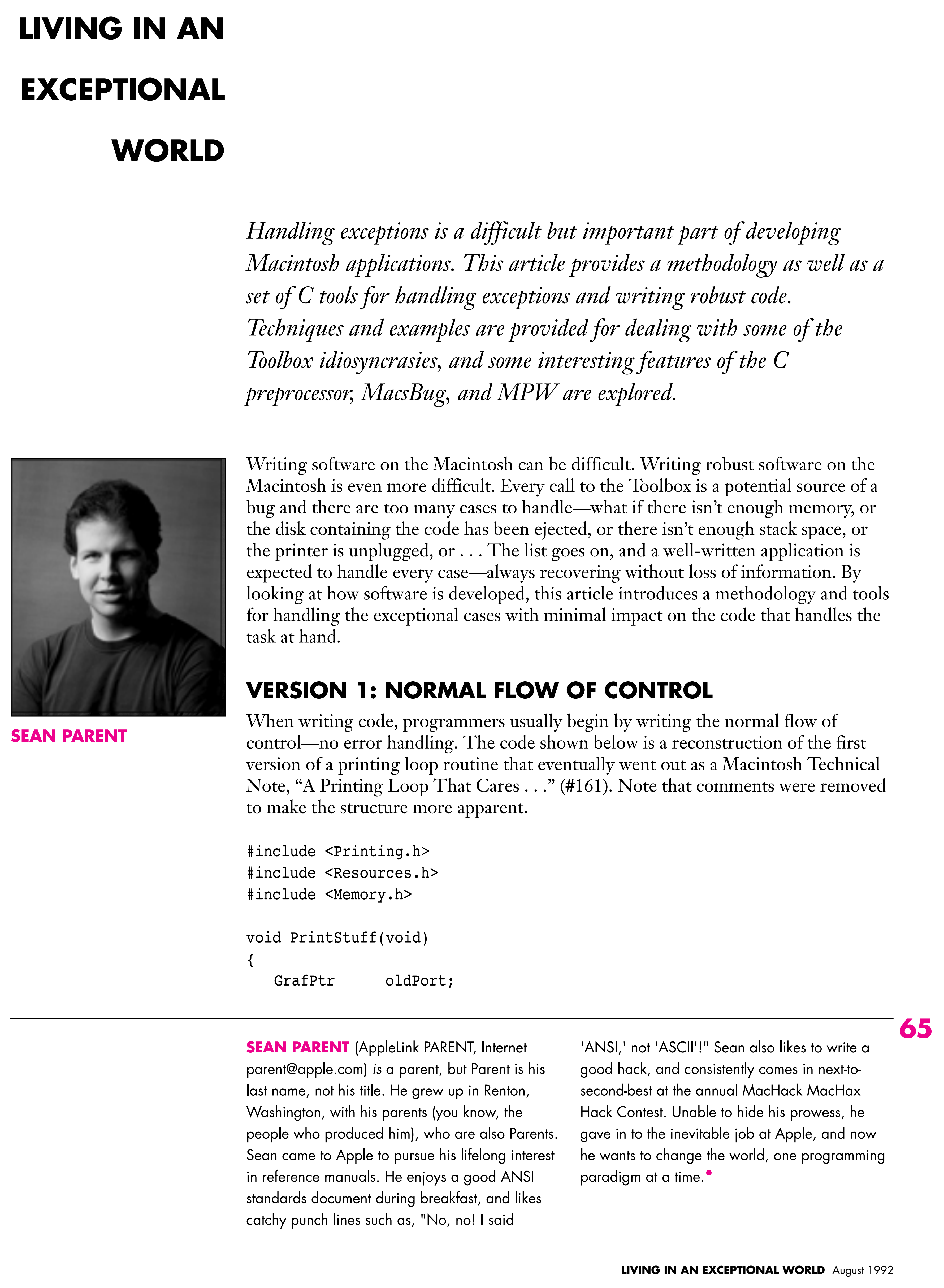
`std::expected<>`

`std::optional<>`

`boost::outcome`

`defer`

`std::error_code`



LIVING IN AN EXCEPTIONAL WORLD

Handling exceptions is a difficult but important part of developing Macintosh applications. This article provides a methodology as well as a set of C tools for handling exceptions and writing robust code. Techniques and examples are provided for dealing with some of the Toolbox idiosyncrasies, and some interesting features of the C preprocessor, MacsBug, and MPW are explored.

SEAN PARENT

Writing software on the Macintosh can be difficult. Writing robust software on the Macintosh is even more difficult. Every call to the Toolbox is a potential source of a bug and there are too many cases to handle—what if there isn't enough memory, or the disk containing the code has been ejected, or there isn't enough stack space, or the printer is unplugged, or . . . The list goes on, and a well-written application is expected to handle every case—always recovering without loss of information. By looking at how software is developed, this article introduces a methodology and tools for handling the exceptional cases with minimal impact on the code that handles the task at hand.

VERSION 1: NORMAL FLOW OF CONTROL

When writing code, programmers usually begin by writing the normal flow of control—no error handling. The code shown below is a reconstruction of the first version of a printing loop routine that eventually went out as a Macintosh Technical Note, "A Printing Loop That Cares . . ." (#161). Note that comments were removed to make the structure more apparent.

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>

void PrintStuff(void)
{
   GrafPtr      oldPort;
```

65

SEAN PARENT (AppleLink PARENT, Internet parent@apple.com) *is* a parent, but Parent is his last name, not his title. He grew up in Renton, Washington, with his parents (you know, the people who produced him), who are also Parents. Sean came to Apple to pursue his lifelong interest in reference manuals. He enjoys a good ANSI standards document during breakfast, and likes catchy punch lines such as, "No, no! I said 'ANSI,' not 'ASCII'!" Sean also likes to write a good hack, and consistently comes in next-to-second-best at the annual MacHack MacHax Hack Contest. Unable to hide his prowess, he gave in to the inevitable job at Apple, and now he wants to change the world, one programming paradigm at a time.•

LIVING IN AN EXCEPTIONAL WORLD August 1992

1992

https://vintageapple.org/develop/pdf/develop-11_9208_August_1992.pdf

AssertMacros.h

```
/*
  File:       AssertMacros.h

  Contains:   This file defines structured error handling and assertion macros for
       programming in C. Originally used in QuickDraw GX and later enhanced.
       These macros are used throughout Apple's software.

       See "Living In an Exceptional World" by Sean Parent
       (develop, The Apple Technical Journal, Issue 11, August/September 1992)
       <http://developer.apple.com/dev/techsupport/develop/issue11toc.shtml> or
       <http://www.mactech.com/articles/develop/issue_11/Parent_final.html>
       for the methodology behind these error handling and assertion macros.

 *  Macro overview:
 *
 *      check(assertion)
 *         In production builds, pre-processed away
 *         In debug builds, if assertion evaluates to false, calls DEBUG_ASSERT_MESSAGE
 *
 *      require(assertion, exceptionLabel)
 *         In production builds, if the assertion expression evaluates to false, goto exceptionLabel
 *         In debug builds, if the assertion expression evaluates to false, calls DEBUG_ASSERT_MESSAGE
 *                          and jumps to exceptionLabel
 *
```

Adobe

AssertMacros.h

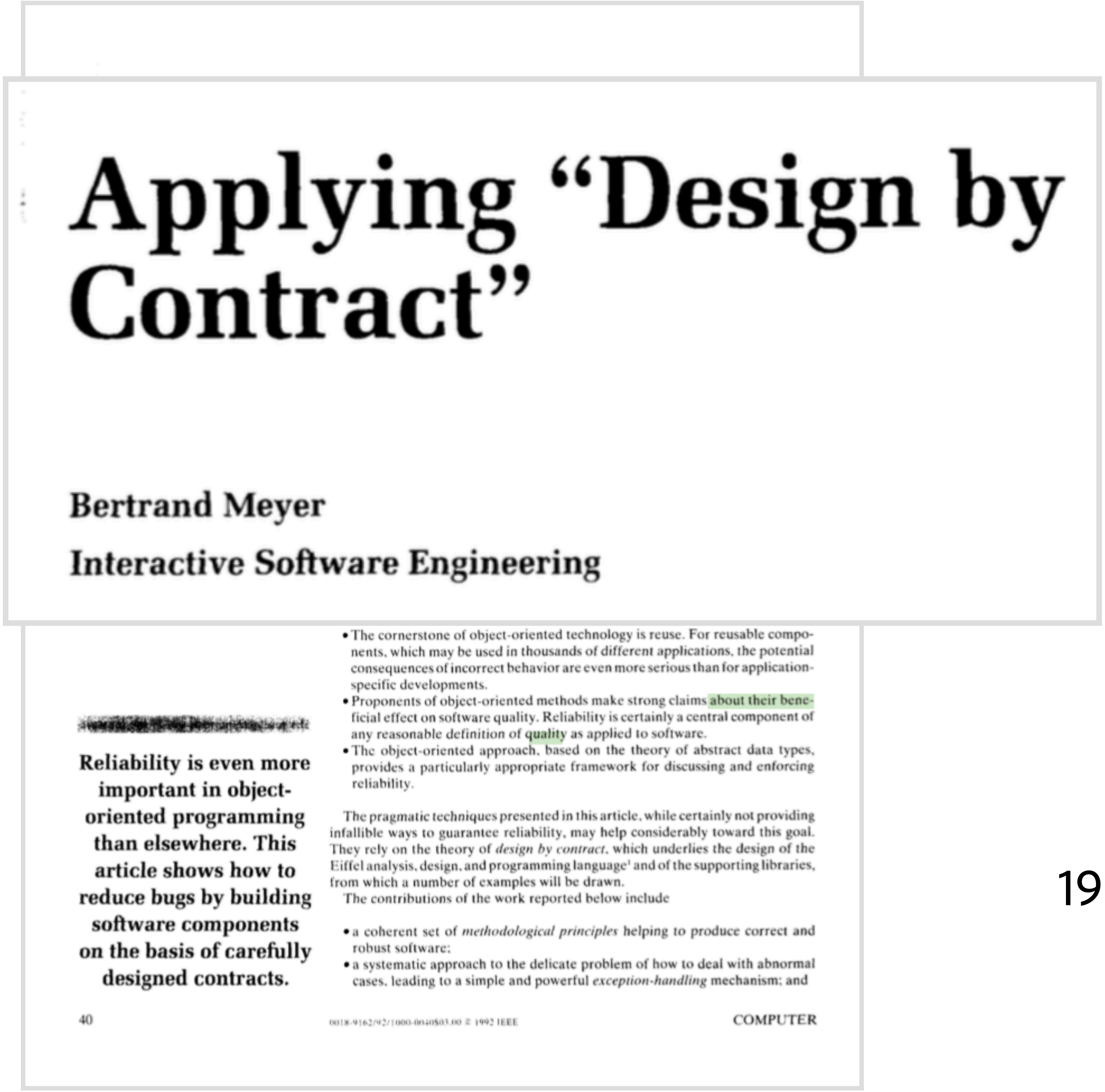
```
/*
 File:      AssertMacros.h

 Contains:   This file defines structured error handling and assertion macros for
      programming in C. Originally used in QuickDraw GX and later enhanced.
      These macros are used throughout Apple's software.

      See "Living In an Exceptional World" by Sean Parent
      (develop, The Apple Technical Journal, Issue 11, August/September 1992)
      <http://developer.apple.com/dev/techsupport/develop/issue11toc.shtml> or
      <http://www.mactech.com/articles/develop/issue_11/Parent_final.html>
      for the methodology behind these error handling and assertion macros.

 *  Macro overview:
 *
 *      check(assertion)
 *         In production builds, pre-processed away
 *         In debug builds, if assertion evaluates to false, calls DEBUG_ASSERT_MESSAGE
 *
 *      require(assertion, exceptionLabel)
 *         In production builds, if the assertion expression evaluates to false, goto exceptionLabel
 *         In debug builds, if the assertion expression evaluates to false, calls DEBUG_ASSERT_MESSAGE
 *                        and jumps to exceptionLabel
 *
```

What is an error?

- Every operation has a set of *preconditions*
- **When preconditions are satisfied, an operation must:**
  - Complete successfully, satisfying ***postconditions***
  - Or, return an error with an indication as to why the postconditions could not be satisfied

- There is a logically isomorphic system of thought where postconditions *include* the error state

Errors are about *Postconditions*



Applying "Design by Contract"

Bertrand Meyer

Interactive Software Engineering

[illegible]

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

40 0018-9162/92/1000-0040$03.00 © 1992 IEEE COMPUTER

1986 (original)

Adobe

Preconditions

- A *Precondition* is an assertion that must be true before an operation

`sort(first, last, compare)`

- *`[first, last)`* is a valid range (implying *`first <= last`*)
- For all *p* in the range *`[first, last)`*, *p* is dereferenceable
- For all *p*, let *v* equal the set of values $*p$;
  - For all pairs *`(`*v_a*`,`* v_b*`)`*, *`compare`* is a predicate establishing a strict-weak-order relation

- The *domain of an operation* is the set of values satisfying all preconditions

Postconditions

- A *Postcondition* is an assertion that must be true just after an operation
  - Unless there is an error

- The postcondition of `sort()` is that all the elements in the range `[first, last)` are in non-decreasing order as defined by `compare`

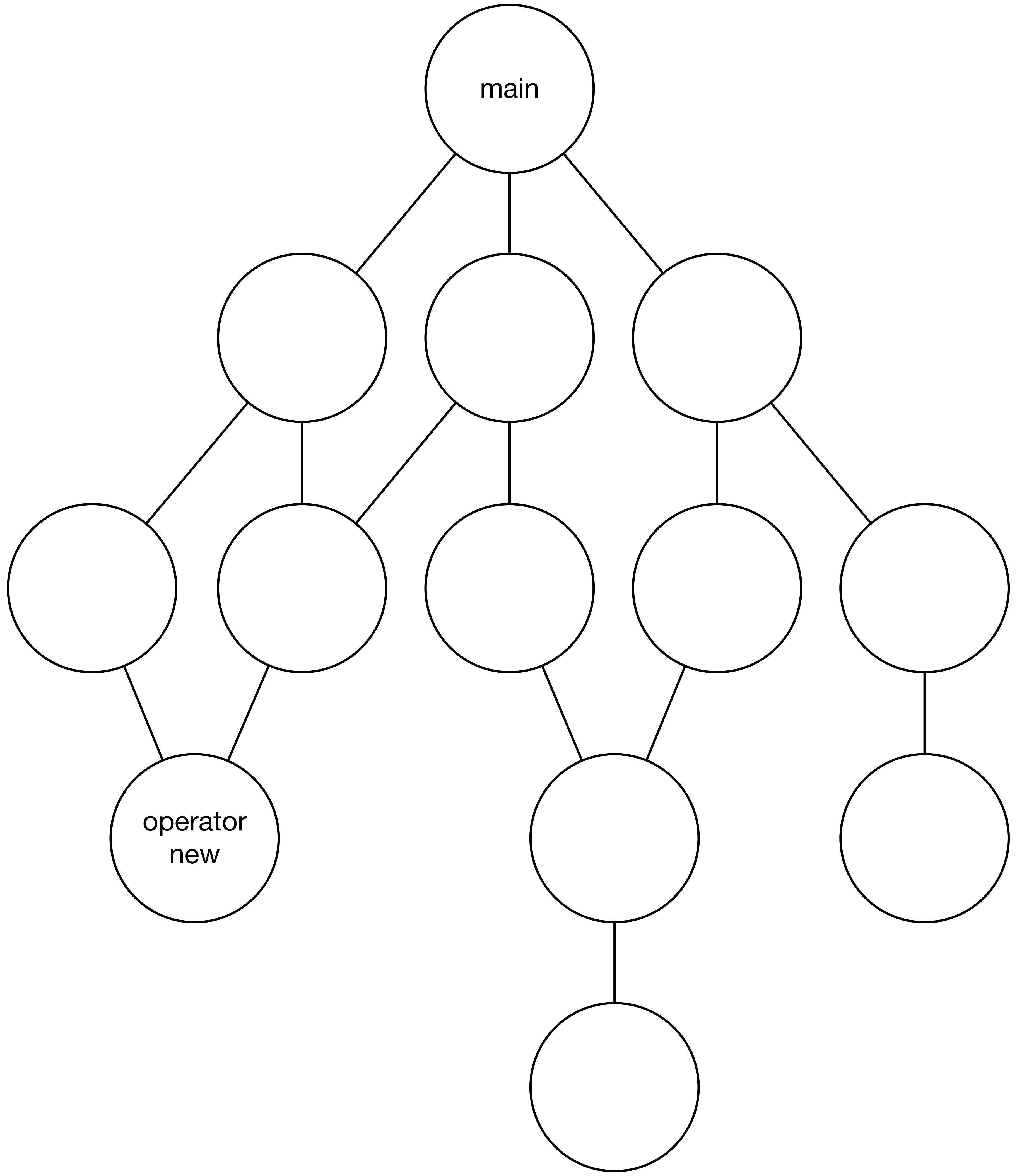
Class Invariants

- *Class invariants* are postconditions that hold for all operations on a type
  - As such, they can be counted on to hold as a precondition for all operations and don't need to be explicitly stated.

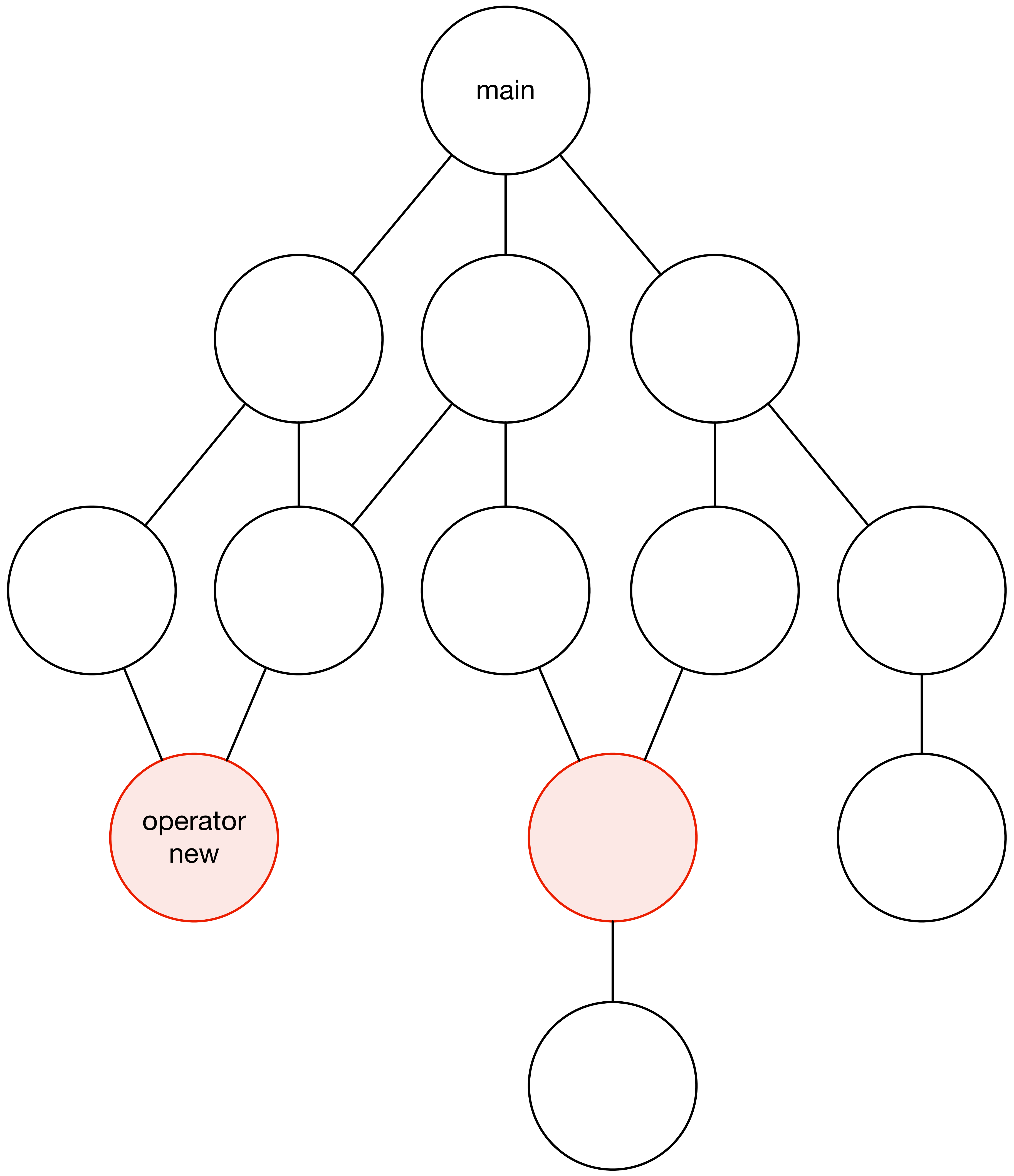
When to report an error

- An error is appropriate when a **postcondition** cannot be satisfied
  - An error is a *recoverable* event
  - Programming errors are not recoverable because you can't tell where they came from
- Resource exhaustion (i.e. out-of-memory)
- I/O failure
- Validating external data
- Cancellation
- Implementation and representation limits

Errors tend to happen at a low level



Errors tend to happen at a low level



Key Points

- Errors tend to occur at a low-level
- A significant amount of code may be in the path from an error to the point where it can be handled

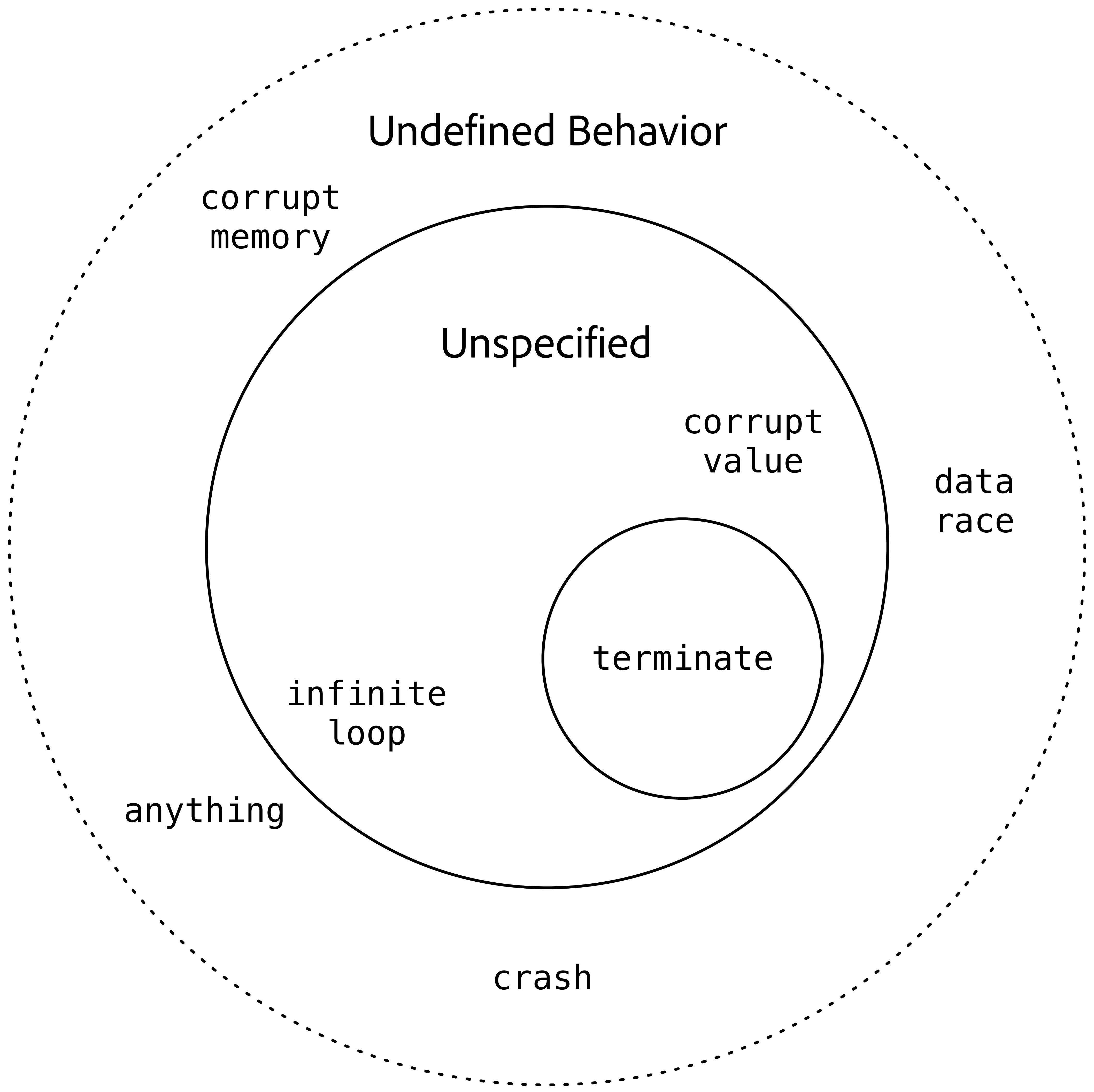
Preconditions

- **When preconditions are not satisfied:**
  - An operation may lead to *undefined behavior*
  - The result may be *unspecified* and may violate *class invariants*
  - It may lead to program *termination*

That's a bug!

Adobe

Possible Effects of Precondition Violation



Safety

- An operation is *safe* if it cannot lead to undefined behavior
  - Directly or indirectly
  - Even if the operation preconditions are violated
- An *unsafe* operation may lead to undefined behavior if preconditions are violated
  - Either directly or during subsequent operations, safe or not

  - We refer to an operation that terminates on a precondition violation or has no preconditions, as *strongly safe*

Safety

- Safety is about incorrect code and the scope of damage it may cause
- Errors are about correct code and recoverable situations

- Safety is a transitive property
- Correctness is not transitive
- Strong safety is not transitive

What is *valid*?

```
int* p{nullptr};
{
    int x{0};
    p = &x;
}
```

- p is *valid* and dereferencable

What is *valid*?

```
int* p{nullptr};
{
    int x{0};
    p = &x;
    ++p;
}
```

- p is *valid* but not dereferencable
- *p is undefined behavior

What is *valid*?

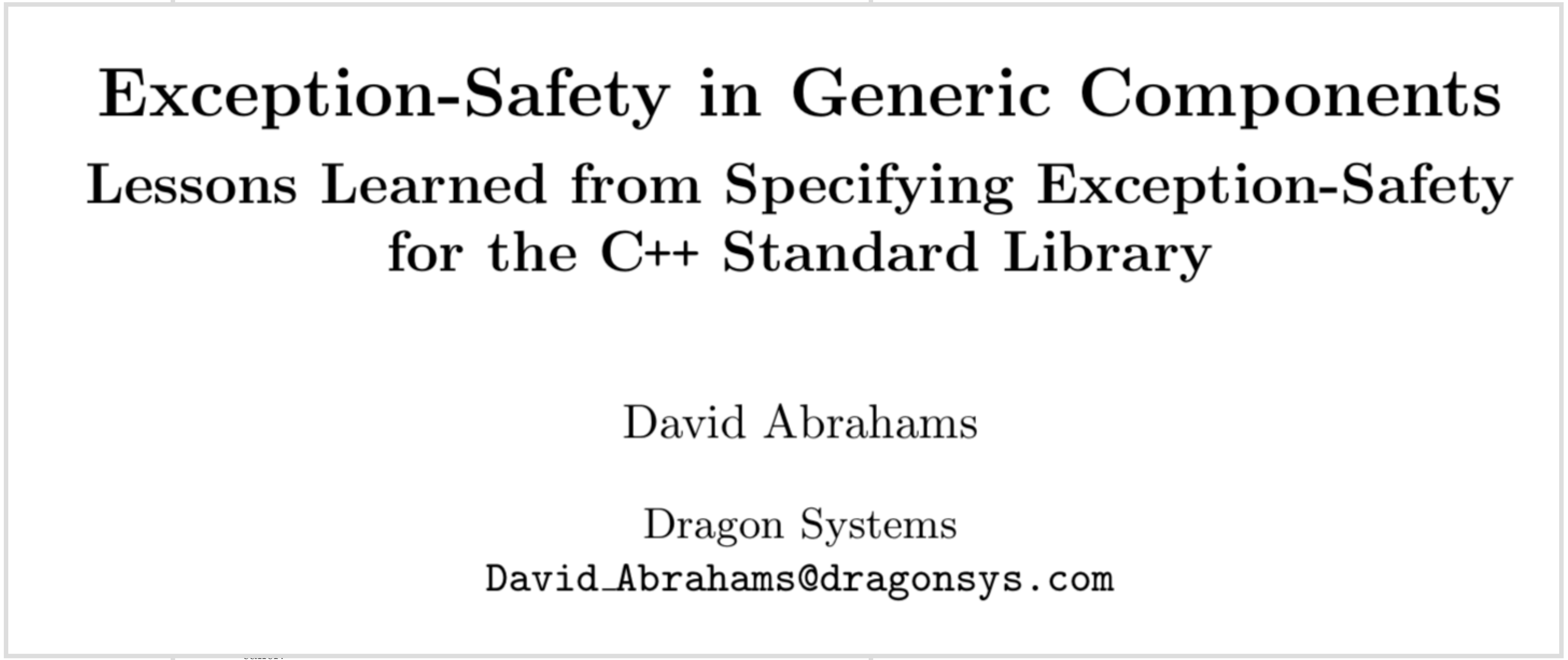
```
int* p{nullptr};
{
    int x{0};
    p = &x;
    ++p;
}
// p
```

- p is *invalid*, p may be assigned-to, or destructed
- *p is undefined behavior
- all other operations, including copy and comparisons, are implementation-defined and may trap

What is *valid*?

```
int* p{nullptr};
{
    int x{0};
    p = &x;
    ++p;
}
p = nullptr;
```

- p is valid but not dereferencable

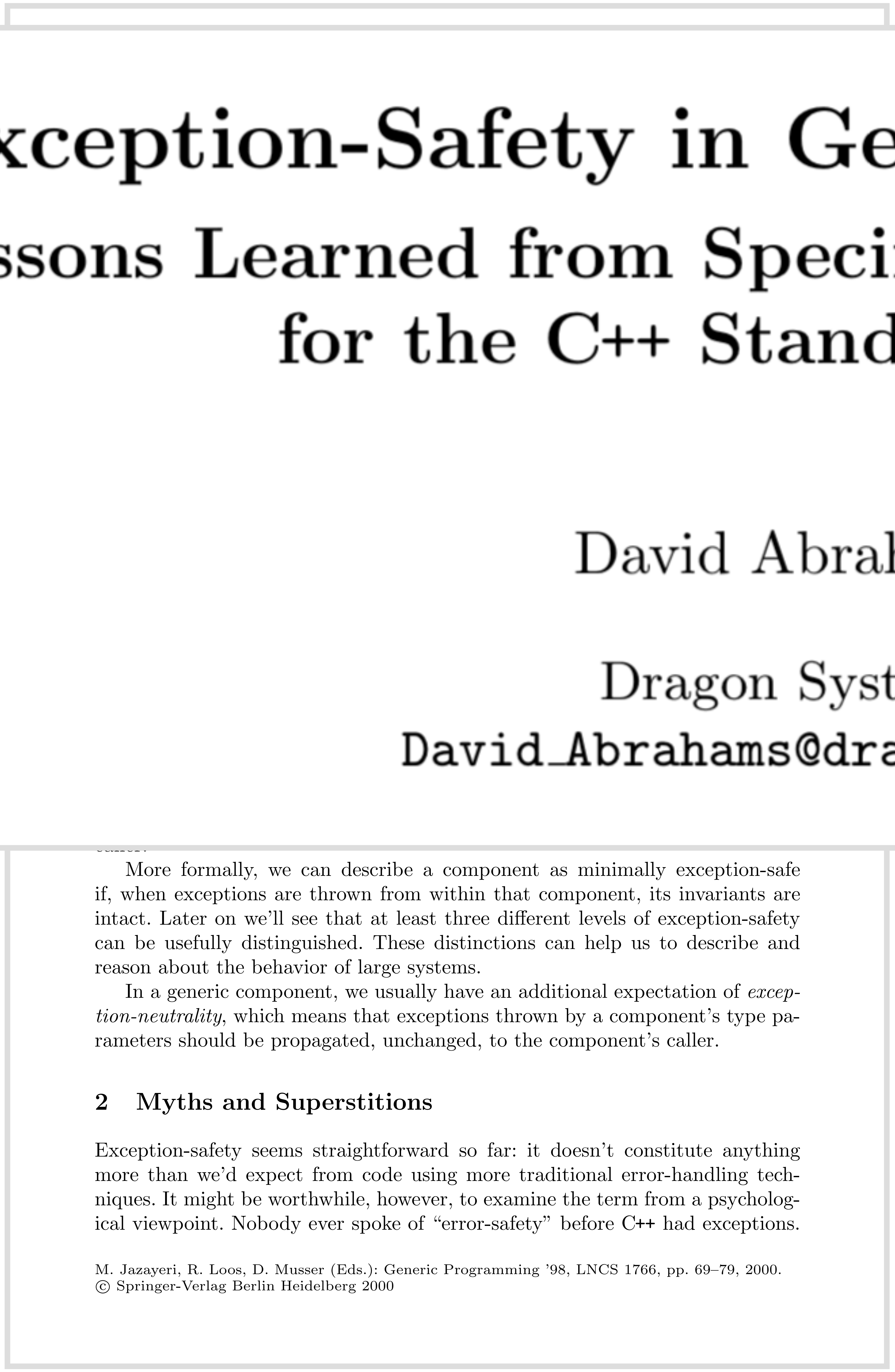


Exception-Safety in Generic Components

Lessons Learned from Specifying Exception-Safety for the C++ Standard Library

David Abrahams

Dragon Systems
David_Abrahams@dragonsys.com



More formally, we can describe a component as minimally exception-safe if, when exceptions are thrown from within that component, its invariants are intact. Later on we'll see that at least three different levels of exception-safety can be usefully distinguished. These distinctions can help us to describe and reason about the behavior of large systems.

In a generic component, we usually have an additional expectation of *exception-neutrality*, which means that exceptions thrown by a component's type parameters should be propagated, unchanged, to the component's caller.

2 Myths and Superstitions

Exception-safety seems straightforward so far: it doesn't constitute anything more than we'd expect from code using more traditional error-handling techniques. It might be worthwhile, however, to examine the term from a psychological viewpoint. Nobody ever spoke of "error-safety" before C++ had exceptions.

M. Jazayeri, R. Loos, D. Musser (Eds.): Generic Programming '98, LNCS 1766, pp. 69–79, 2000.
© Springer-Verlag Berlin Heidelberg 2000

1998

Abstract

"This paper represents the knowledge accumulated in response to a real-world need: that the C++ Standard Template Library exhibit useful and well-defined interactions with exceptions, the error-handling mechanism built-in to the core C++ language. It explores the meaning of exception-safety, reveals surprising myths about exceptions and genericity, describes valuable tools for reasoning about program correctness, and outlines an automated testing procedure for verifying exception-safety."

Myths

“It's almost as though exceptions are viewed as a mysterious attack on otherwise correct code, from which we must protect ourselves. Needless to say, this doesn't lead to a healthy relationship with error handling!”

Guarantees

- *Basic exception guarantee*
  - *Invariants* hold
  - The values of objects being modified are otherwise unspecified
- *Strong exception guarantee*
  - The state is restored to the state prior to the failing operation
  - *Invariants* hold by extension
- *No exception guarantee*

Fix mutating state when propagating an exception

- The exception guarantees are about what happens to mutable state
  - Operations that do not mutate state can propagate errors directly

Class Invariants & Error Propagation

- The guarantees say we need to consider:
  - Operations where invariants are temporarily broken

- Only consider member functions that modify the object

`vector` example

- Consider
  `operator=`
  `assign`
  `reserve`
  `shrink_to_fit`
  `clear`
  `insert`
  `erase`
  `push_back`
  `emplace_back`
  `pop_back`
  `resize`

- Ignore
  `constructors`
  `get_allocator`
  `at, operator[]`
  `front`
  `back`
  `data`
  `begin, end`
  `empty`
  `size`
  `max_size`
  `capacity`
  `comparison operators`
  `destructor`

vector example

- We also don't have to worry about non-*basis operation*

```
void resize(size_type sz, const T& c) {
    if (sz < size()) {
        erase(begin() + sz, end());
    } else {
        T t{c};
        reserve(sz);
        for (size_type f = 0, l = size() - sz; f != l; ++f) {
            push_back(t);
        }
    }
}
```

Fix invariants when propagating an exception

- Invariants should only be broken inside member functions with private access
  - Keep your *basis* small to narrow the impact of exceptions

Example

```
template <class T, class U>
class zip_vector {
    // invariant: _v0.size() == _v1.size()
    vector<T>   _v0;
    vector<U>   _v1;
public:
    void push_back(T&& x, U&& y) {
        _v0.push_back(move(x));
        _v1.push_back(move(y)); // what if this throws?
    }
};
```

Example

```
template <class T, class U>
class zip_vector {
    // invariant: _v0.size() == _v1.size()
    vector<T>   _v0;
    vector<U>   _v1;
public:
    void push_back(T&& x, U&& y) {
        _v0.push_back(move(x));
        try {
            _v1.push_back(move(y));
        } catch(...) {
            _v0.pop_back();
            throw;
        }
    }
};
```

Postconditions

- A *Postcondition* is an assertion that must be true just after an operation
  - Unless there is an error
    - Then invariants are satisfied but the state is otherwise unspecified

  - The basic guarantee falls out of all the documentation of your system

Safety and Exception Guarantees

- The basic and strong exception guarantees are statements about correctness
  - They are not transitive properties
  - "It is 'safe' in the sense that it is not allowed to crash, but its output may be unpredictable."
    - David Abrahams regarding the basic guarantee

The Other Way Around

- Top-down vs bottom-up design
- In Generic Programming, *lifting* is finding commonalities bottom-up across component
- The *other way around* is to find common *requirements* of callers and common models

Requirements

- Generalization of preconditions to also include
  - Operations & semantics
  - Dependent types
- In non-generic code, these requirements exist but are implied by the type
- Concepts allow us to specify type requirements
  - But are still missing value preconditions

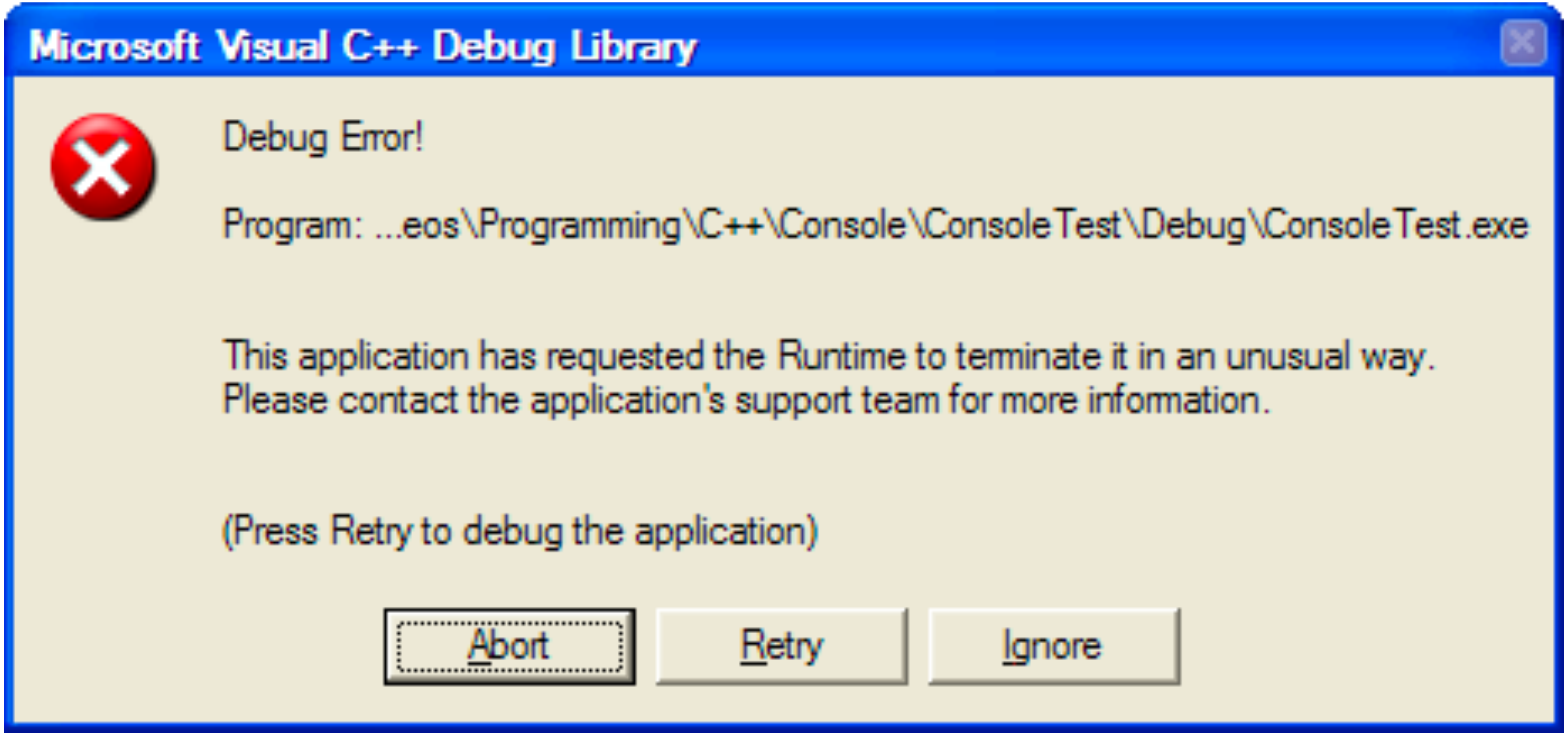
Guarantees

- Generalization of postconditions
  - May be conditionalized

Example

- sort **requires** that value_type(iterator) is movable
- basic_string<T> **guarantees** it is movable
- therefore an array<string> is sortable

Stopping Error Propagation


Microsoft Visual C++ Debug Library

Debug Error!

Program: ...eos\Programming\C++\Console\ConsoleTest\Debug\ConsoleTest.exe

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

(Press Retry to debug the application)

Stopping Error Propagation

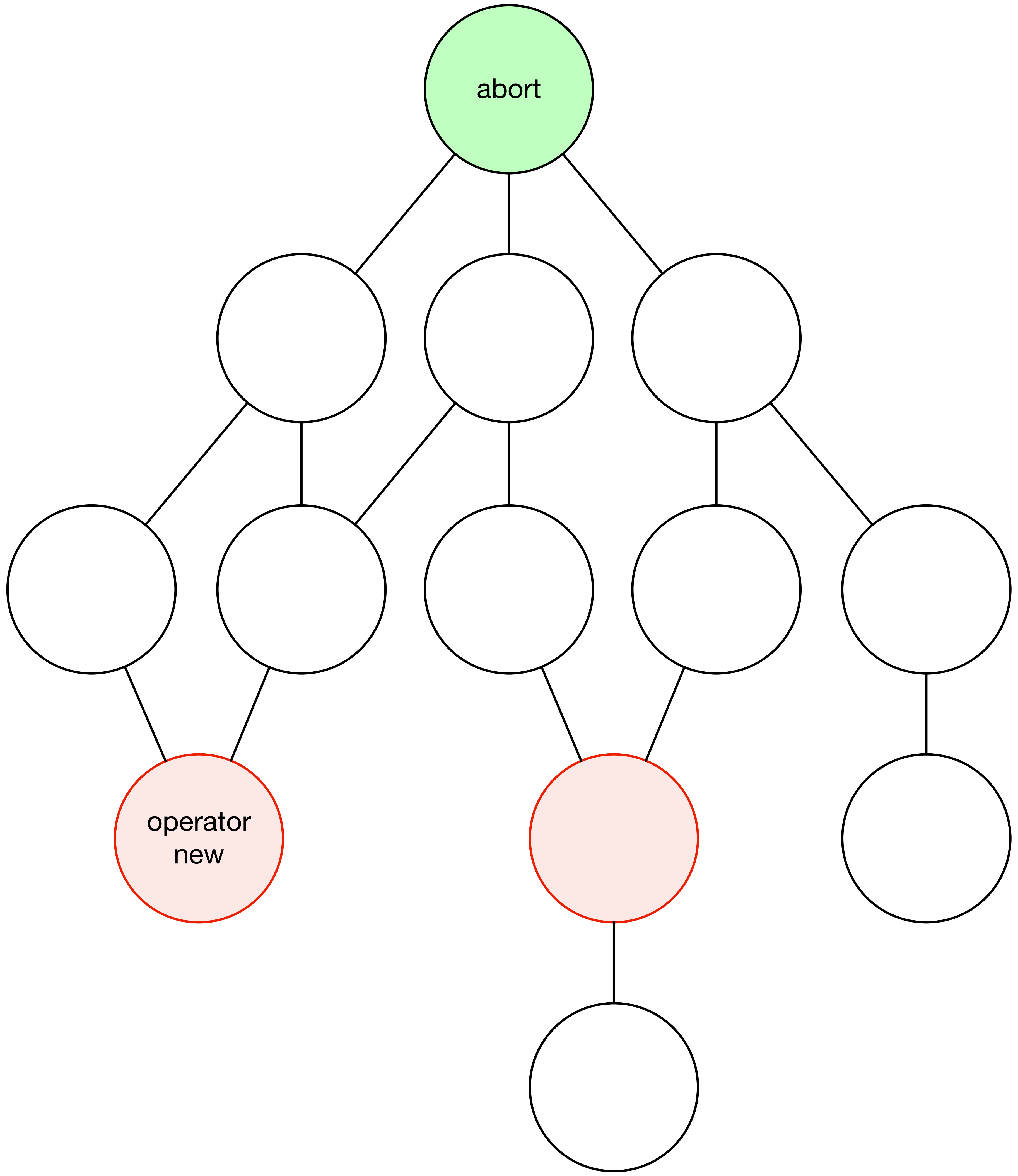
- The operation must satisfy its postconditions
  - Or terminate

Abort

Report & Terminate

- Program cannot satisfy postconditions
- Requirements
  - None
- Appropriate for failure during initialization

Abort is Likely to Happen Near Main

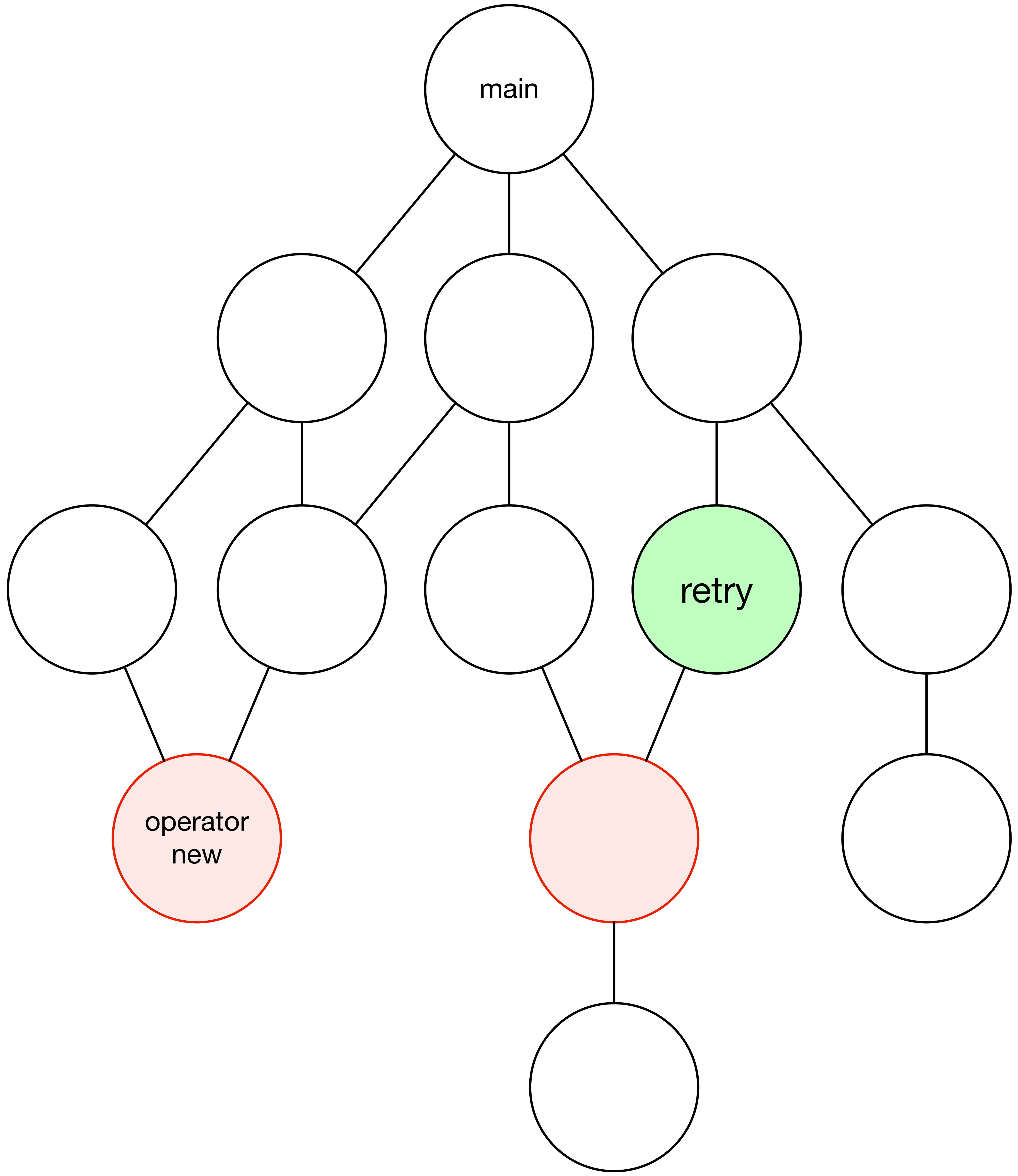


Retry

- Retry the operation using the same, or a different, approach
  - Successful retry will satisfy postconditions
  - Or, resume error propagation
- Requirements
  - Discard incomplete work

- Appropriate for some I/O, memory cache
- Best of done close to point of failure

Retry is Best Handled Near the point of Failure



Ignore

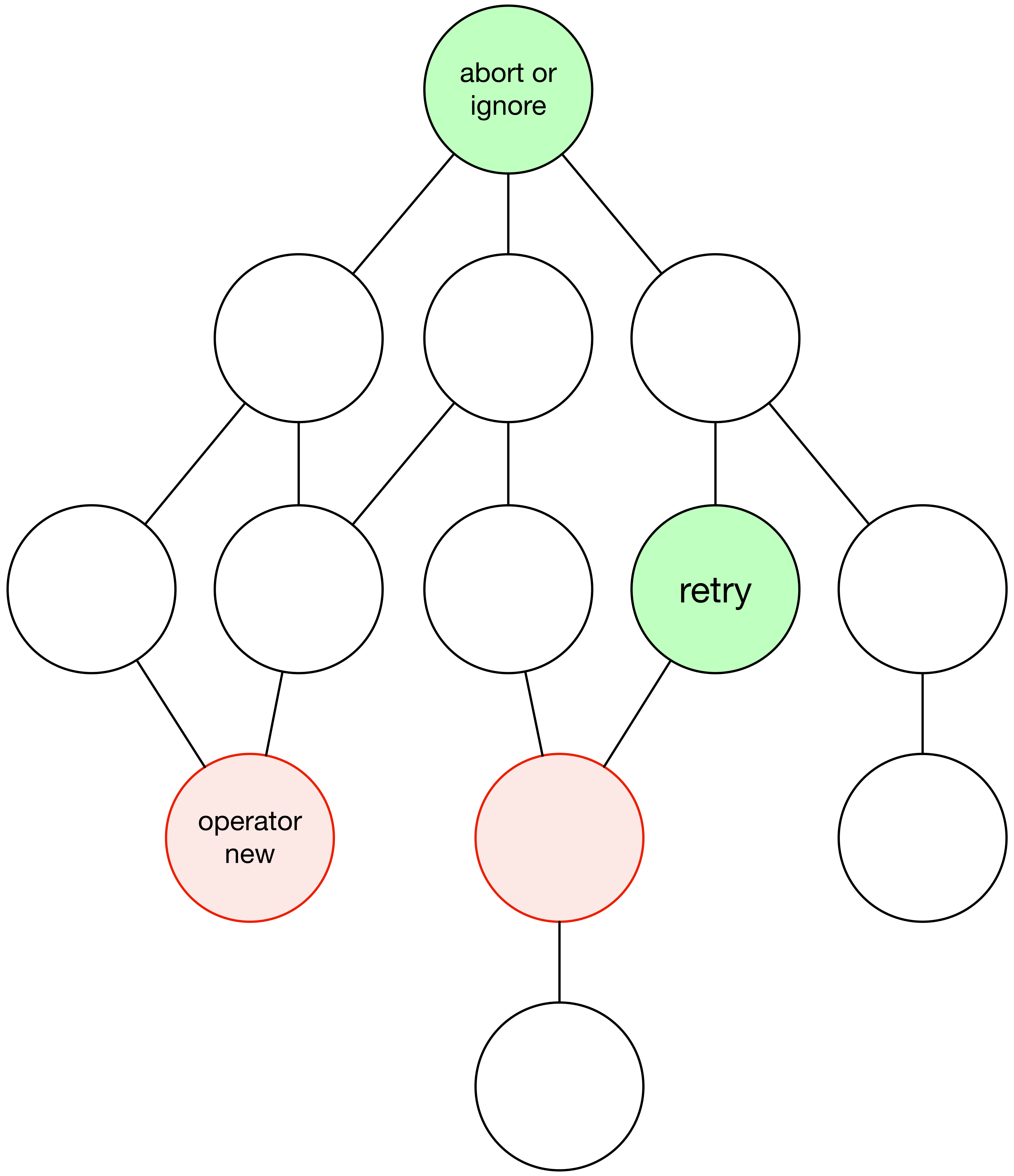
Report & Continue

- Requirements
  - Discard incomplete work

- Often implemented as *transactional* operations
  - Strong exception guarantee

- Appropriate for sequenced or optional operations

Retry is Best Handled Near thePpoint of Failure



Discard Incomplete Work

- “The options for recovery, in this case, are limited: either destruction or resetting the component to some known state before further use.”
  - Dave Abrahams regarding basic exception guarantee

Requirements of Exceptions

- What do we mean by “discard partial work”?
- An object which is being mutated when an exception is thrown must leave the object in a state
  - within the domain of destruction
  - within the domain of the left-hand side of assignment

- The object should also be in the domain of any operation which does not read the object
  - i.e. `clear()`

Requirements of Exceptions

- The exception guarantees are not requirements
- The implied requirement is “satisfies class invariants”
  - But that doesn’t actually tell you anything unless there is a requirement that some operations have no preconditions

Invariants?

- We'll call the class invariants in the absence of exceptions, the *desired invariants*
- The weaker invariant of only destructible and assignable-to is *invalid*

- We can, trivially, weaken class invariants to include this state
  - `invalid() || desired_invariants()`

- And add the precondition for all the other operations
  - `valid() && …`

Why invalid?

- Destructible and assignable-to are transitive
  - To destruct an object, destruct all the parts
  - To assign an object, assign all the parts
- If we define the class invariants and preconditions of its operations to include invalid
  - Without doing anything else in code to account for exceptions
  - A class using "`= default`" for destruction and assignment can be *exception correct*

Other Cases Simplified

- Ensuring destructible and assignable-to in cases where the default implementations are insufficient is often simpler than reasoning about the desired invariants
- Parts can be packaged to allow default definitions
  - i.e. `unique_ptr<T>` vs `T*`

Relationship to Move

- Any function operation of the form $x = f(x)$ can be expressed as a mutating operation $a(x)$.
- It can also be expressed as a consuming operation $x = g(move(x))$
- If $g()$ throws an exception, x is in a moved-from state
- The required postcondition after an error are identical to those of a moved-from object
  - See P2345

Class Invariants & Postconditions

- A *class invariant* is a postcondition
- A *postcondition* is an assertion that must be true just after an operation
  - Unless there is an error
    - Then the result may be invalid

Example

```
template <class T, class U>
class zip_vector {
    // invariant: invalid() || (_v0.size() == _v1.size())
    vector<T>   _v0;
    vector<U>   _v1;
public:
    // precondition: valid()
    void push_back(T&& x, U&& y) {
        _v0.push_back(move(x));
        _v1.push_back(move(y));
    }
};
```

Example

```
template <class T, class U>
class zip_vector {
    // invariant: _v0.size() == _v1.size()
    // may be invalid by exception
    vector<T>   _v0;
    vector<U>   _v1;
public:
   void push_back(T&& x, U&& y) {
        _v0.push_back(move(x));
        _v1.push_back(move(y));
    }
};
```

Correct Exception Handling has Zero Impact on Most Code

Adobe
Bē
Artwork by Dan Zucco

Disadvantage

- Stronger preconditions may result in unsafe operations

Addendum 1: What about shared state?

- Correct mutation requires exclusivity
- Everything within the program is part of the whole program
- You must understand the implicit structure as a whole
  - The scope of which is somewhere between the calling scope and the program as a whole
  - You cannot reason about it locally

- By catching close to `main()` you may be *under* the implicit structure
  - Or be able to identify the root(s)

Addendum 2: The Problems with C++ Exceptions

- Exceptions are the default
- Operator new throws and is replaceable
- Reliance on RTTI and inheritance
- No guarantee of `noexcept` move for standard components
- ABI leakage

About the artist

Dan Zucco

London-based 3D art and motion director Dan Zucco creates repeating 2D patterns and brings them to life as 3D animated loops. Inspired by architecture, music, modern art, and generative design, he often starts in Adobe Illustrator and builds his animations using Adobe After Effects and Cinema 4D. Zucco's objective for this piece was to create a geometric design that felt like it could have an infinite number of arrangements.

Made with

Ai **Adobe Illustrator** Ae **Adobe After Effects**

Adobe

Bē

Artwork by **Dan Zucco**