Woes of Scope Guards and Unique_Resource
5+ years in the making...

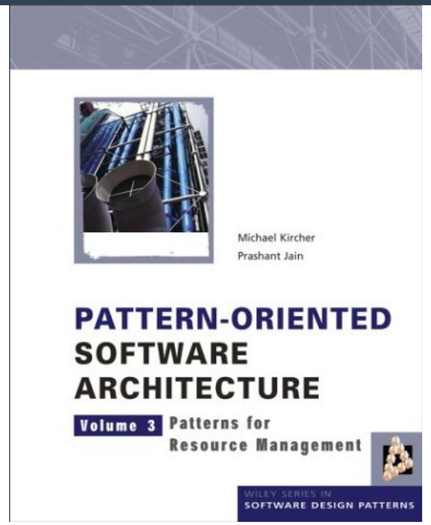
Prof. Peter Sommerlad
CPPCon 2018

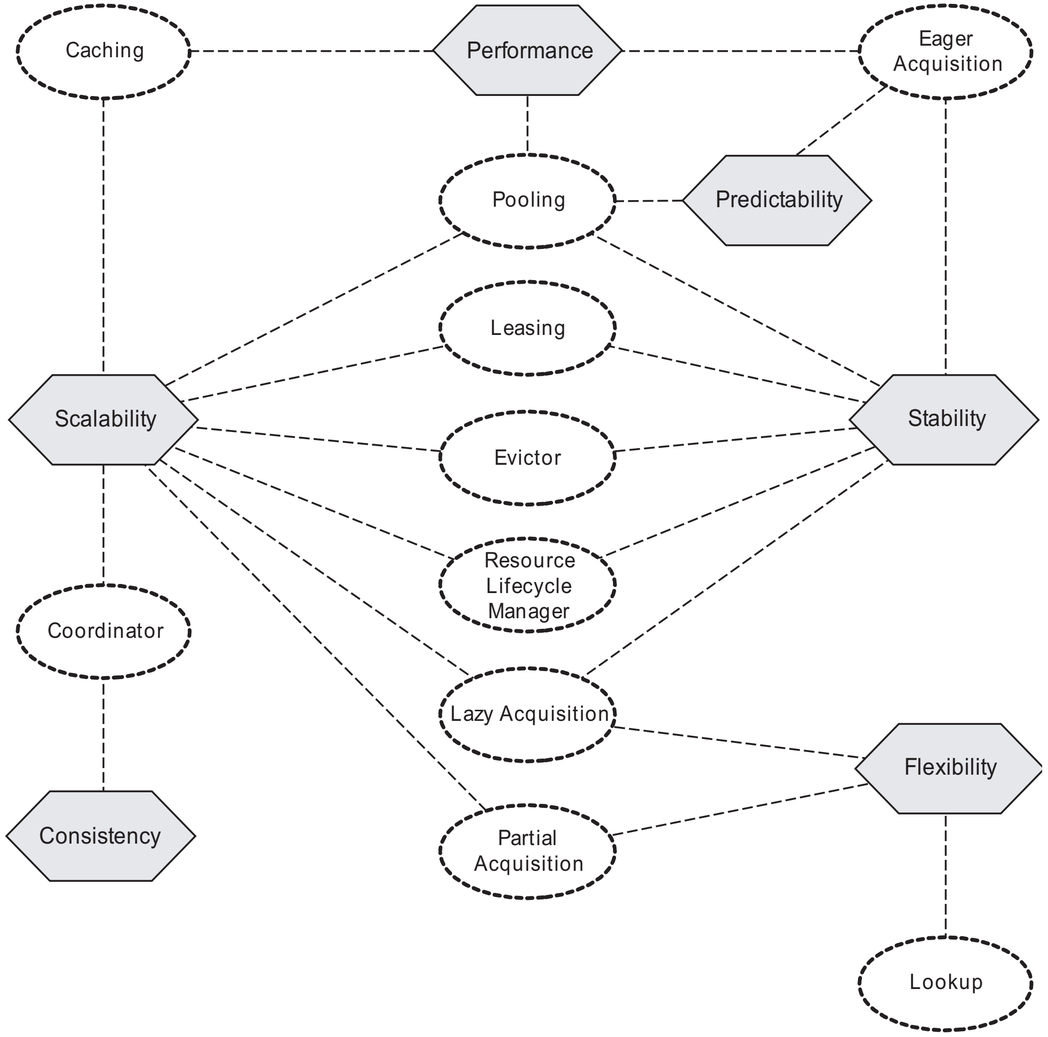
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

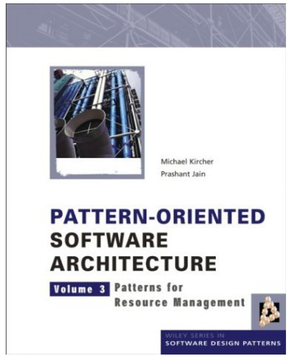
What is a resource?

- **Resources have a lifetime**
  - acquire, use, and release
- **Resources are lifetime managed**
  - often by the OS or runtime system
  - or by manager objects
- **Resources can be exclusive or shared**
- **Not releasing a resource can be an error**
  - a leak, causing problems (safety, security, DoS)
- **There is a pattern book on resource management**

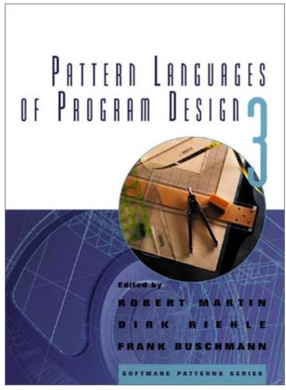




Resource Management is not easy



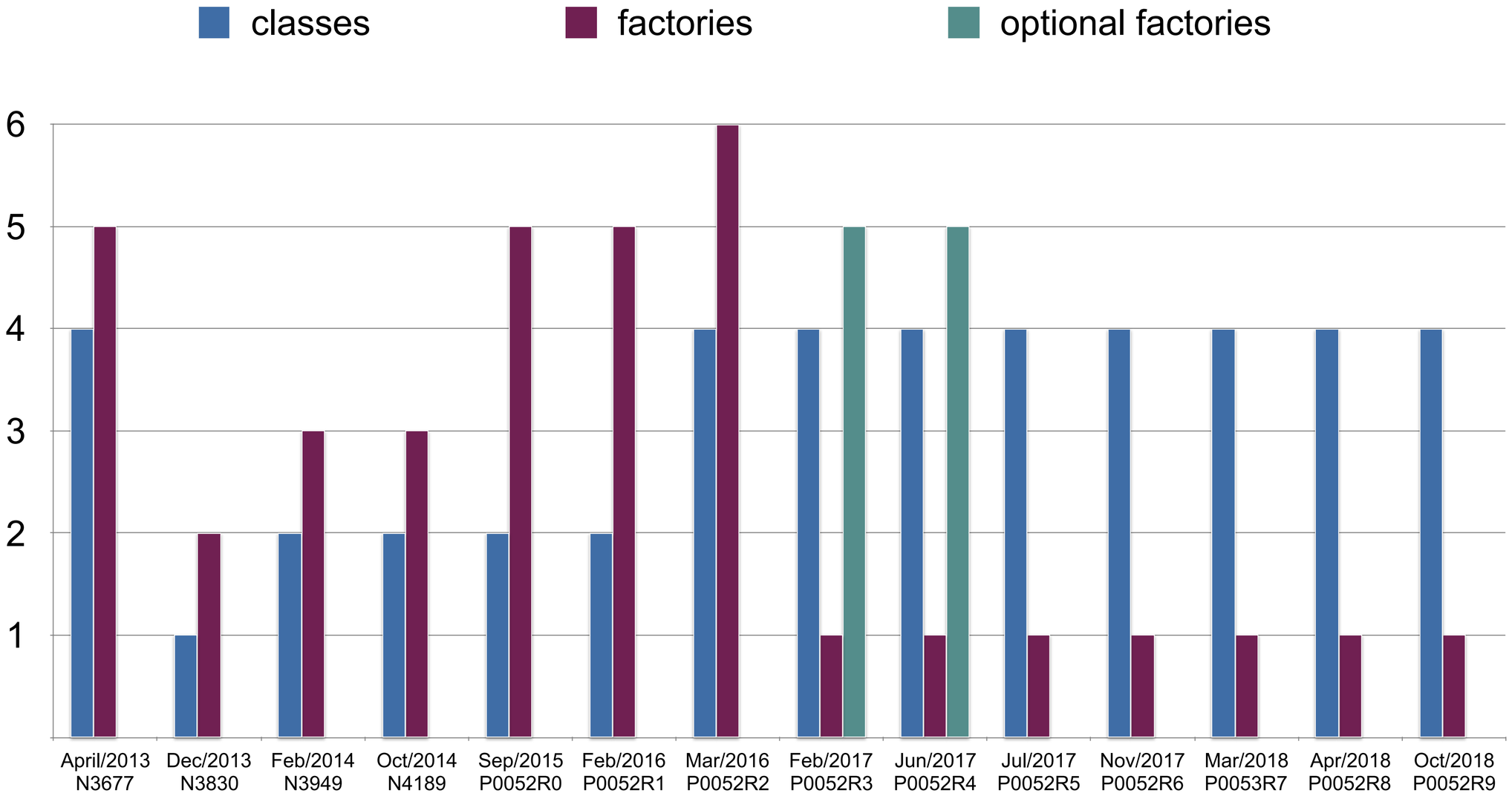
- **Pattern-oriented Software Architecture - Vol. 3: Patterns for Resource Management**
  - 10 patterns (2004), e.g. Pooling, Caching, Leasing, Lazy Acquisition
- **Missing feature from GoF Design Patterns (it contains Factory Method)**
  - Factory and **Disposal** Methods (Kevlin Henney, 2004)
  - Smalltalk and Java programmers often forget to release resources correctly
- **Manager Design Pattern (Peter Sommerlad, 1996)**
  - "The Manager design pattern encapsulates management of instances of a class into a separate manager object. This allows variation of management functionality independent of the class and for reuse of the manager for different object classes."



RAII - Resource Acquisition Is Initialization

- **Term coined by Bjarne Stroustrup**
- **One of the cornerstones of C++**
  - established by C++'s deterministic object lifetime model
  - and by ~destructors()
- **Idea: Use a resource in a scope and clean up when exiting from the scope**
  - acquire the resource in the constructor
  - use it through the class' interface
  - release the resource in the destructor
- **C++11 move mechanism made unique handles for resources workable**
  - `std::auto_ptr` just didn't get there safely

A timeline - just a simple(?), easy, innocent, generic RAII proposal...



Generic RAII is hard to get right, plus the standard is a moving target!

RAII classes in the standard

Using RAII is often "Sane and Safe" see my other talk

- **Anywhere, where destruction has a side effect...**
- **I/O handles: fstream**
- **Memory: unique_ptr, string, containers**
- **Lockables (aka Mutex): lock_guard, unique_lock, scoped_lock**

- **Who knows more?**
- **Missing for:**
  - other OS resource handles = original idea of unique_resource
  - own lifetime managed resources = makes implementation of generic unique_resource interesting

- **Many use Tracer classes (often encapsulated in with macros for easy turn off)**
  - for "printf debugging" or logging
  - can be useful in strange convoluted framework control flows
  - but if you need them in production code, something is wrong with your architecture
- **Or as an afterthought for "finally"**
  - which I consider a mistake borrowed from Java
  - **In C++ RAII RULES**

```
struct Tracer{
   explicit Tracer(std::ostream & out,std::string name="")
   :out{out},name{name}{
      out << "Tracer created: " << name << std::endl;
   }
   ~Tracer(){
      out << "Tracer destroyed: " << name << std::endl;
   }
   Tracer(Tracer const& other)
   :out{other.out},name{other.name+" copy"}{
      out << "Tracer copied: " << name << std::endl;
   }
   void show() const {
      out << "Tracer: " << name << std::endl;
   }
   std::ostream &out;
   std::string name;
};
```

unique_ptr as generic RAII? DON'T!

- **Often (mis-)proposed for resource handles that are pointer-sized**
  - it "works" but was not intended to.
  - what if your often not publicly announced/opaque HANDLE type changes (size)
  - will not work for non-trivial HANDLE types, where copying/moving might have side-effects
- **Deleter's pointer type alias must conform to concept NullablePointer**
  - can you guarantee that for an opaque handle type?
  - What about context-dependent invalid values?

```
struct Deleter
{
    // By defining the pointer type, we can delete a type other than T*.
    // In other words, we can declare unique_ptr<HANDLE, Deleter> instead of
    // unique_ptr<void, Deleter> which leaks the HANDLE abstraction.
    typedef HANDLE pointer;

    void operator()(HANDLE h)
    {
        if(h != INVALID_HANDLE_VALUE)
        {
            CloseHandle(h);
        }
    }
};

void OpenAndWriteFile()
{
    // Specify a deleter as a template argument.
    std::unique_ptr<HANDLE, Deleter> file(CreateFile(_T("test.txt"),
                                                     GENERIC_WRITE,
                                                     0,
                                                     NULL,
                                                     CREATE_ALWAYS,
                                                     FILE_ATTRIBUTE_NORMAL,
                                                     NULL));
    if(file.get() != INVALID_HANDLE_VALUE)
    {
        DWORD size;
        WriteFile(file.get(), "Hello World", 11, &size, NULL);
    }
}
```

8

Compare it against the documented error return value. That means that you should compare it against `INVALID_HANDLE`, 0, -1, non-zero, or `<=32` (I'm not kidding with the last one, see ShellExecute).

share improve this answer

answered Oct 11 '10 at 11:10

MSalters

131k ● 8 ● 114 ● 264

https://codesequoia.wordpress.com/2012/08/26/stdunique_ptr-for-windows-handles/

unique_ptr for FILE*

- **Most voted Stackoverflow anser:**
- **what is wrong with &::fclose ?**
  - not sanctioned by ISO C++
  - fclose(NULL) fatal, but fortunately not called by ~unique_ptr()
- **a conforming solution**
  - cppreference (later adjusted)
- **a better solution?**
  - my suggestion on SO
  - even better with separate FileCloser deleter class

Problem

Should be

```
unique_ptr<FILE, int(*)(FILE*)>(fopen("file.txt", "rt"), &fclose);
```

since http://en.cppreference.com/w/cpp/memory/unique_ptr

or, since you use C++11, you can use `decltype`

```
std::unique_ptr<FILE, decltype(&fclose)>
```

```
void close_file(std::FILE* fp) { std::fclose(fp); }
//...
  {
      std::unique_ptr<std::FILE, decltype(&close_file)> fp(std::fopen("demo.txt", "r"),
                                                           &close_file);
      if(fp) // fopen could have failed; in which case fp holds a null pointer
        std::cout << (char)std::fgetc(fp.get()) << '\n';
  } // fclose() called here, but only if FILE* is not a null pointer
    // (that is, if fopen succeeded)
```

```
unique_ptr<FILE, void(*)(FILE*)>(fopen("file.txt", "rt"),
   [](FILE *fp){ ::fclose(fp);});
```

```
static_assert(sizeof(std::unique_ptr<char const,decltype(&std::free)>)==sizeof(char*),"");
// compile error! too big
```

- **using std::free()'s address is not sanctioned by the standard**
- **and it adds overhead**
  - Therefore, define an empty function object to call free and be ready (note the const_cast required)

```
struct free_deleter{
  template <typename T>
  void operator()(T *p) const {
    std::free(const_cast<std::remove_const_t<T>*>(p));
  }
};
template <typename T>
using unique_C_ptr=std::unique_ptr<T,free_deleter>;

static_assert(sizeof(char *)==sizeof(unique_C_ptr<char>),"");
// compiles!
```

unique_resource, how can it be used for int handles...

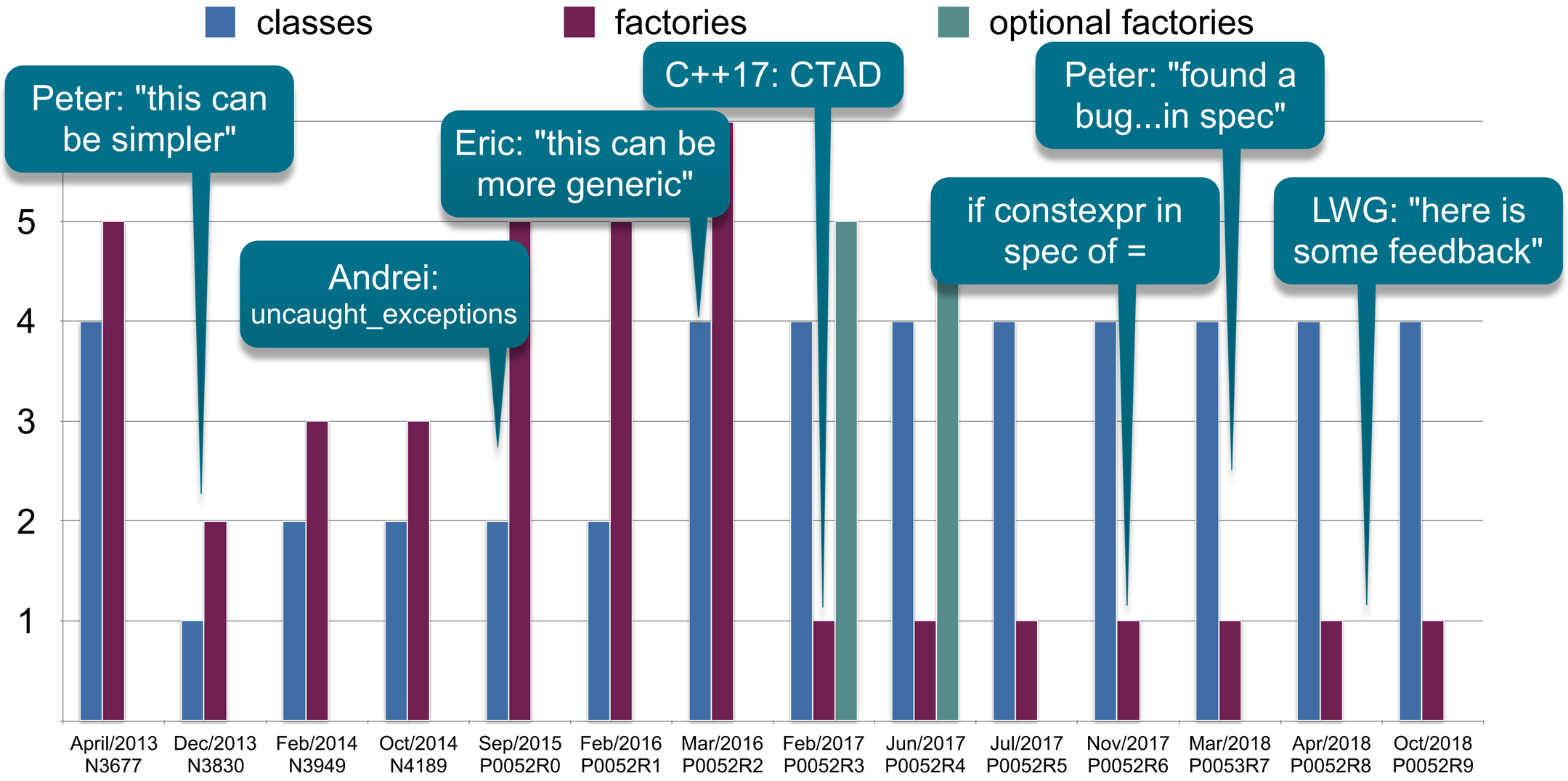

```
void demontrate_unique_resource_with_POSIX_IO() {
  const std::string filename = "./hello1.txt";
  auto close=[](auto fd){::close(fd);};
  {
    auto file = unique_resource(::open(filename.c_str(), O_CREAT | O_RDWR | O_TRUNC, 0666), close);

    ::write(file.get(), "Hello World!\n", 12u);
    ASSERT(file.get() =/= -1);
  }
  {
    std::ifstream input { filename };
    std::string line { };
    getline(input, line);
    ASSERT_EQUAL("Hello World!", line);
    getline(input, line);
    ASSERT(input.eof());
  }
  ::unlink(filename.c_str());
  {
    auto file = make_unique_resource_checked(::open("nonexistingfile.txt", O_RDONLY), -1, close);
    ASSERT_EQUAL(-1, file.get());
  }

}
```

- **scoped_function**
  - run function on scope exit
- **scoped_resource_unchecked**
  - call function with resource handle on scope exit
- **scoped_resource**
  - call function with resource handle on scope exit, unless handle is invalid
- **unique_resource**
  - "only for constexpr deleter functions"(?) = passing function pointer as template argument by macro

Pass function address as constructor argument
Factory functions for type deduction

- **https://wg21.link/N3677**

A timeline - just a simple(?), easy, innocent, generic RAII proposal...



Generic RAII is hard to get right, plus the standard is a moving target!

- **a single tuple-of-resources based attempt**
  - that was a very bad idea but I just proposed std::apply() and wanted to use it 🌈
- **implementation assumes well-behaved resources**
  - no problems with moving/copying them
  - all resources available on construction
- **single deleter function taking all resources**
  - no problems with moving it assumed
  - no problems with piecemeal moving
- **TOO SIMPLE...**

```
template<typename DELETER, typename ... R>
class scoped_resource {
   DELETER deleter; // deleter must be void(R... ) noexcept
   std::tuple<R...> resource;
   bool execute_on_destruction;
public:
   explicit
   scoped_resource(DELETER deleter, R... resource,
                   bool shouldrun=true) noexcept
      : deleter{std::move(deleter)}
      , resource{std::make_tuple(std::move(resource)... )}
      , execute_on_destruction{shouldrun}{}
   ~scoped_resource() {
      invoke(invoke_it::once);
   } // ... a lot omitted
   void invoke(invoke_it const strategy = invoke_it::once)
     noexcept {
      if (execute_on_destruction) {
         std::apply(deleter, resource);
      }
      execute_on_destruction = strategy==invoke_it::again;
   }
};

template<typename DELETER, typename ... R>
   auto make_scoped_resource(DELETER t, R ... r) {
   return scoped_resource<DELETER, R...>(std::move(t),
          std::move(r)... );
}
```

- **"No need to add a type for occasional RAII idioms"**
  - is that a good idea? 💡
  - does adding a type hurt? 😢
- **uses "interesting" macros**
  - macro expansion is incomplete syntax
    - needs `{lambda-body};`
  - IDEs and me hate those macros! 🤬
- **Introduced me to the concept of memorizing number of exceptions in flight** 😃
  - scope_exit, scope_fail, scope_success

```
template <typename FunctionType, bool executeOnException>
class ScopeGuardForNewException {
   FunctionType function_;
   UncaughtExceptionCounter ec_;
public:
   explicit ScopeGuardForNewException(const FunctionType& fn)
         : function_(fn) {
   }
   explicit ScopeGuardForNewException(FunctionType&& fn)
         : function_(std::move(fn)) {
   }
   ~ScopeGuardForNewException() noexcept(executeOnException) {
      if (executeOnException == ec_.isNewUncaughtException()) {
         function_();
} }
};
enum class ScopeGuardOnFail {};
template <typename FunctionType>
    ScopeGuardForNewException<
       typename std::decay<FunctionType>::type, true>
    operator+(detail::ScopeGuardOnFail, FunctionType&& fn) {
      return ScopeGuardForNewException<
        typename std::decay<FunctionType>::type, true>(
          std::forward<FunctionType>(fn));
}
#define SCOPE_FAIL \
   auto ANONYMOUS_VARIABLE(SCOPE_FAIL_STATE) \
   = ::detail::ScopeGuardOnFail() + [&]() noexcept
```

Sanity… What can go wrong?

- **If moving a resource is guaranteed to not throw, not much, you just move.**
  - otherwise, if moving really fails, you are left with a moved-from resource that you can no longer release, 🦄
- **BUT, if move might throw, you need to copy**
  - `std::move_if_noexcept(resource)`
  - Unless your type is not CopyConstructible 🛑
- **BUT, copying can fail, because it might need to acquire additional resources**
  - then you still have the original, that you can release

```
template< class T >
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible_v<T>
    && std::is_copy_constructible_v<T>,
    const T&,
    T&&
>::type move_if_noexcept(T& x) noexcept;

...

template<typename R, typename D>
class unique_resource
{
    static_assert((std::is_move_constructible_v<R>
                   && std::is_nothrow_move_constructible_v<R>)
                  || std::is_copy_constructible_v<R>,
              "resource must be nothrow_move_constructible or "
              "copy_constructible");
    static_assert((std::is_move_constructible_v<R>
                   && std::is_nothrow_move_constructible_v<D>)
                   || std::is_copy_constructible_v<D>,
              "deleter must be nothrow_move_constructible or "
              "copy_constructible");
```

- **For robust generic code, anything can be passed to your template**
  - compile-guard against it and SFINAE...
- **Do not leak, especially when copying fails**
  - you should test that moving always works
- **Eric Niebler provided infrastructure**
  - _box as holder object, gets a scope guard as ctor argument that is called on failure
  - basic_scope_guard provides a cleanup hook, if initialization is needed that might fail
- **conclusion:**
  you need to jump to many hoops for fail-safe generic code.

```
template<typename T>
class _box
{
    T value;
    _box(T const &t) noexcept(noexcept(T(t)))
      : value(t) {}
    _box(T &&t)
      noexcept(noexcept(T(std::move_if_noexcept(t))))
      : value(std::move_if_noexcept(t)) {}
public:
    template<typename TT, typename GG,
        typename =
          std::enable_if_t<std::is_constructible_v<T, TT>>>
    explicit _box(TT &&t, GG &&guard)
      noexcept(noexcept(_box((T &&) t)))
      : _box(std::forward<TT>(t))
    {
        guard.release(); // all went well
    }
...
    template<typename EFP, typename =
        std::enable_if_t<_ctor_from<EFP>::value>>
    explicit basic_scope_exit(EFP &&ef)
      noexcept(_noexcept_ctor_from<EFP>::value)
      : exit_function(std::forward<EFP>( ef),
            _make_failsafe(_noexcept_ctor_from<EFP>{}, &ef))
    {}
```

- **If construction can not throw, fine do it**
- **if if might throw, guarantee that if it fails the passed exit function get actually called**
- **Similarly on move construction**
  - if move constructing the exit function succeeds, _box-ctor-body runs and release()s "that"
  - otherwise "that" remains and its destruction calls its exit function
- **I lost you? that is expected. That is the reason, why something with such intricate complexity needs to be in the C++ standard.**
  DO NOT TRY AT HOME!

```
template<typename EFP,
         typename = std::enable_if_t<_ctor_from<EFP>::value>>
explicit basic_scope_exit(EFP &&ef)
  noexcept(_noexcept_ctor_from<EFP>::value)
  : exit_function(std::forward<EFP>( ef),
      _make_failsafe(_noexcept_ctor_from<EFP>{}, &ef))
{}
basic_scope_exit(basic_scope_exit &&that)
 noexcept(noexcept(detail::_box<EF>(that.exit_function.move(),
                   that)))
: Policy(that), exit_function(that.exit_function.move(), that)
{}
~basic_scope_exit() noexcept(noexcept(exit_function.get()()))
{
    if(this→should_execute())
        exit_function.get()();
}
...
static auto _make_failsafe(std::true_type, const void *)
{
    return detail::_empty_scope_exit{}; // NOP
}
template<typename Fn>
static auto _make_failsafe(std::false_type, Fn *fn)
{
    return basic_scope_exit<Fn &, Policy>(*fn);
}
```

Determined by policy (exit,fail,success)

- **You need to invent very nasty behaving classes that no one should write**
  - but you never know what you get
  - and the standard library needs to care about it
  - if it can still make it work (somehow)
- **Function objects that throw on copy or move**
- **resource objects that throw on move**
- **resource objects that can not be assigned**
  - so reset(newone) becomes unavailable
- **NEVER DO THAT IN PRODUCTION CODE!**

```
struct non_assignable_resource{
   non_assignable_resource()=default;
   non_assignable_resource(int){}
   void operator=(const non_assignable_resource &) = delete;
   non_assignable_resource& operator=(non_assignable_resource  &&)
     noexcept(false){ throw "buh";};
   non_assignable_resource(non_assignable_resource  &&) =default;
};
void testscopeExitWithNonAssignableResourceAndReset(){
   std::ostringstream out { };
   const auto &lambda = [&](auto &&) {out << "lambda done.\n";};
   {
      auto guard=unique_resource(non_assignable_resource{},
                 std::cref(lambda));
      //guard.reset(2);// compile error?
   }
   ASSERT_EQUAL("lambda done.\n",out.str());
}
```

Moving unique_resource - complexity explained through move-assignment

- **if moves can not throw, move**
  - sequence does not matter, because nothing can go wrong
- **if copying is needed, and that can throw!**
  - copy first, if that is OK, move the rest, if if can not throw
  - or copy both
- **Only if everything i… …ganize that assigned fro… …rce is no longer releas… …**
- **…wo resources at once is a nightmare**
  - more lead to insanity

This is the reason, why you should employ unique_resource when wrapping multiple resources in a single class

```
unique_resource &operator=(unique_resource &&that)
    noexcept(is_nothrow_delete_v &&
             std::is_nothrow_move_assignable_v<R…
             std::is_nothrow_move_assignabl…
{ // static_asserts omitted
  if(&that ≠ this){
   reset();
   if constexpr (is_n…                    …able_v<detail::_box<R>>)
      if conste…                          …e_assignable_v<detail::_box<D>>)
      {
                                          …move(that.resource);
                                          …d::move(that.deleter);
                                          …
                                          …leter = _as_const(that.deleter);
        resource = std::move(that.resource);
      }
    else if constexpr
      (is_nothrow_move_assignable_v<detail::_box<D>>) {
        resource = _as_const(that.resource);
        deleter = std::move(that.deleter);
     } else {
        resource = _as_const(that.resource);
        deleter = _as_const(that.deleter);
     }
     execute_on_destruction =
       std::exchange(that.execute_on_destruction, false);
  }
  return *this;
}
```

`unique_resource(unique_resource&& rhs) noexcept(`*see below*`)`

6 *Effects:* First, initialize `resource` as follows:

(6.1) — If `is_nothrow_move_constructible_v<R1>` is `true`, from `std::move(rhs.resource)`,

(6.2) — otherwise, from `rhs.resource`.

7 [*Note:* If initialization of `resource` throws an exception, `rhs` is left owning the resource and will free it in due time. — *end note*]

8 Then, initialize `deleter` as follows:

(8.1) — If `is_nothrow_move_constructible_v<D>` is `true`, from `std::move(rhs.deleter)`;

(8.2) — otherwise, from `rhs.deleter`.

9 If initialization of `deleter` throws an exception and if `is_nothrow_move_constructible_v<R1>` is `true`:

```
rhs.deleter(resource); rhs.release();
```

10 Finally, `execute_on_destruction` is initialized with `exchange(rhs.execute_on_destruction,false)`.

11 [*Note:* The explained mechanism ensures no leaking of resources. — *end note*]

`unique_resource(unique_resource&& rhs) noexcept(`*`see below`*`)`

6 *Effects:* First, initialize `resource` as follows:

(6.1) — If `is_nothrow_move_constructible_v<R1>` is `true`, from `std::move(rhs.resource)`,

(6.2) — otherwise, from `rhs.resource`.

7 [*Note:* If initialization of `resource` throws an exception, `rhs` is left owning the resource and will free it in due time. — *end note*]

8 Then, initialize `deleter` as follows:

(8.1) — If `is_nothrow_move_constructible_v<D>` is `true`, from `std::move(rhs.deleter)`;

(8.2) — otherwise, from `rhs.deleter`.

9 If initialization of `deleter` throws an exception and `is_nothrow_move_constructible_v<R1>` is `true` and `rhs.execute_on_destruction` is `true`:

```
rhs.deleter(resource); rhs.release();
```

10 Finally, `execute_on_destruction` is initialized with `exchange(rhs.execute_on_destruction,false)`.

11 [*Note:* The explained mechanism ensures no leaking and no double release of resources. — *end*

- **globals, throwing arbitrarily, dependency on intricate behavior, close coupling**
- **Who has not seen such abominations in "production" code?**

```
struct nasty{};

struct deleter_2nd_throwing_copy {
   deleter_2nd_throwing_copy()=default;
   deleter_2nd_throwing_copy(deleter_2nd_throwing_copy const&){
      if (copied %2) {
         throw nasty{};
      }
      ++copied;
   }
   void operator()(int const & t) const {
      ++deleted;
   }
   static inline int deleted{};
   static inline int copied{};
};
```

```
void test_sometimes_throwing_deleter_copy_ctor(){
   using uid=unique_resource<int,deleter_2nd_throwing_copy>;
   uid strange{1,deleter_2nd_throwing_copy{}};
   ASSERT_EQUAL(0,deleter_2nd_throwing_copy::deleted);

   strange.release();
   ASSERT_EQUAL(0,deleter_2nd_throwing_copy::deleted);

   try {
      uid x{ std::move(strange)};
      FAILM("should have thrown");
   } catch(nasty const &){
   }
   ASSERT_EQUAL(0,deleter_2nd_throwing_copy::deleted); // fails
   ASSERT_EQUAL(1,deleter_2nd_throwing_copy::copied);
}
```

More surprises with lambdas and scope guards (thanks to BSI)

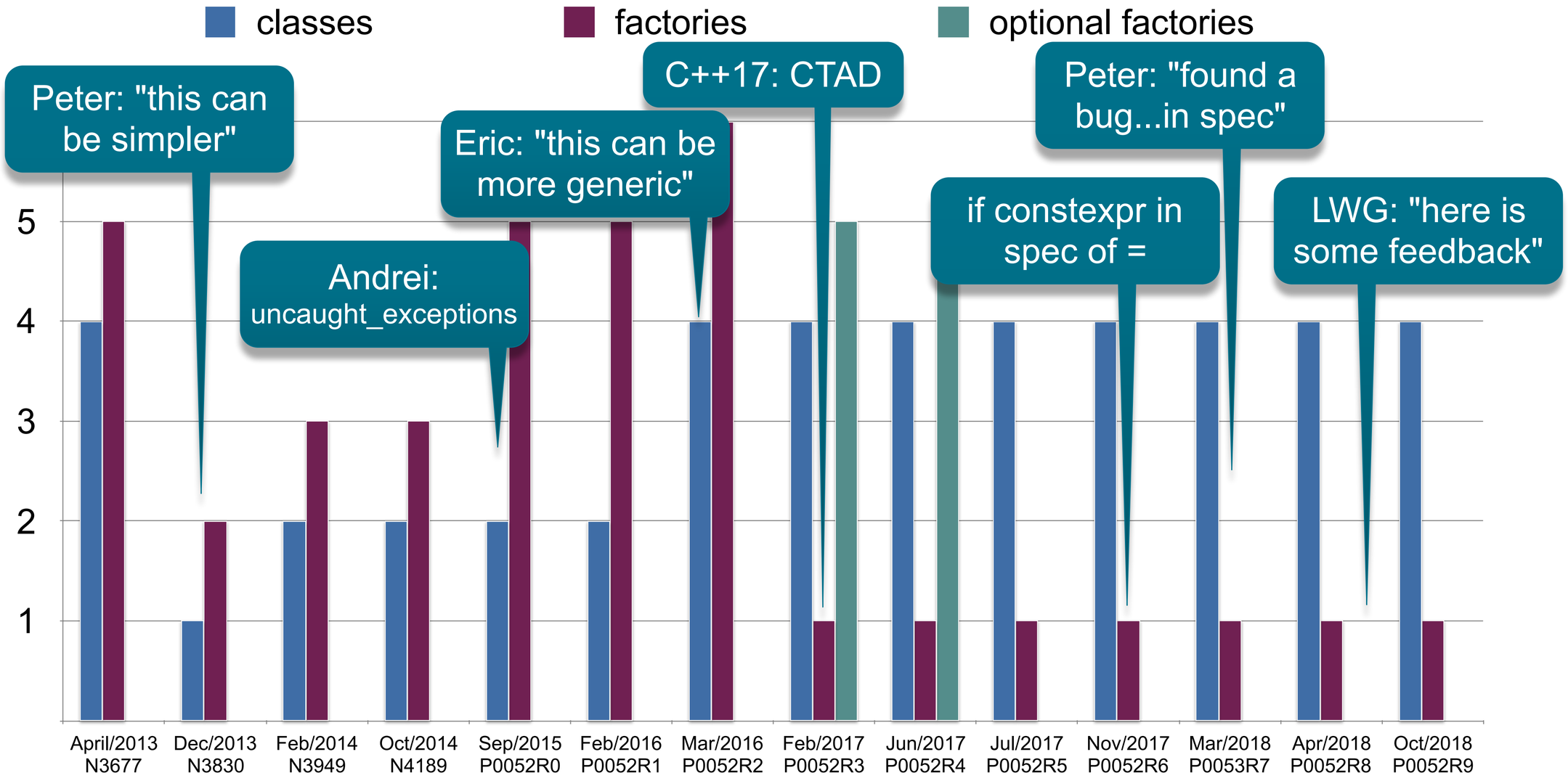
2 [*Note:* If the exit function object of a `scope_success` or `scope_exit` object refers to a local variable of the function where it is defined, e.g., as a lambda capturing the variable by reference, and that variable is used as a return operand in that function, that variable might have already been returned when the *`scope_guard`*'s destructor executes, calling the exit function. This can lead to surprising behavior. — *end note*]

- **accessing a returned variable in a scope guard lambda by reference, can be surprising**
- **you get the moved-from state**

```
std::string access_returned_from_string(size_t& len){
  std::string s{"a string"};
  scope_exit guard{[&]{ len = s.size();}};
  return (s); // pessimize copy elision
}
void DemonstrateSurprisingReturnedFromBehavior(){
  size_t len{0×ffffffff};
  auto s = access_returned_from_string(len);
  // exected: ASSERT_EQUAL(s.size(),len);
  // what really happens
  ASSERT_EQUAL(0,len);
}
```

The final feedback? Batavia August 2018 - what you get if you are not there...

THIS SLIDE'S CONTENT IS HIDDEN TO PROTECT THE REVIEWERS AND NOTE TAKERS



Generic RAII is hard to get right, plus the standard is a moving target!

Implementation and specification tasks left

- **SFINAE for empty base optimization with empty deleter objects (C++17), or:**
- **[[no_unique_address]] for deleter object**
  - should work with no-capture lambdas in C++20 --> sizeof(unique_resource<T,D>) == sizeof(T)
  - but would require restructuring current implementation (Eric's boxes for deleters)
- **[[nodiscard]] for scope guard and unique_resource constructors**
  - AFAIK not yet in the standard, but would be nice, I do not know what holds us back?
  - would avoid annoying bugs like:
    - `scope_exit{[]{ std::cout << "im am done" << std::endl;}};` // immediate
- **Incorporate the "final" feedback**
  - section names of library member specification changed in Rapperswil in June

- **According to Andrei: "No need to add a type for occasional RAII idioms"**
- **I do not follow Andrei's claim, better build nicely testable RAII abstractions around resources**
  - Most examples I found are hacks, e.g., in the core guidelines demonstrating gsl::finally()
  - Except for Andrei's transactional file handling examples:

```
using std::filesystem::path;
void copy_file_transact(path const & from, path const & to) {
   path t = to.native() + ".deleteme";
   auto guard= scope_fail{ [t]{remove(t);} };
   copy_file(from, t);
   rename(t, to);
}
```

Take aways

- **If you do not know RAII, learn about and use it consciously**
- **Understand your resources:**
  - what kind of resources
  - what lifecycles do your resources have
  - what amount of resources your programs need to deal with
- **Generic resource management is hard, even for a single resource, do not try two manually**
- **In your own classes**
  - Wrap each resource member acquired individually to achieve transaction semantics on acquisition failure of one
- **Do not naively use scope guards throughout your code base, build proper (RAII) abstractions**
- **Beware of "pseudo-dangling" references to moved-from variables in lambdas run at scope exit**

Tool Time Demo Cevelop tonight
Bring your laptop with Cevelop installed to get a C++ hat or chocolate

Questions?

peter.sommerlad@hsr.ch
@PeterSommerlad

Cevelop

Your C++ deserves it

Sponsors welcome!

Commercial licensing possible!

Download IDE at:
www.cevelop.com

By the way, thanks for the great piece of software! This is by far the best free IDE for C/C++ so far after trying basically all the free C/C++ IDE. motowizlee on github 27.04.2017