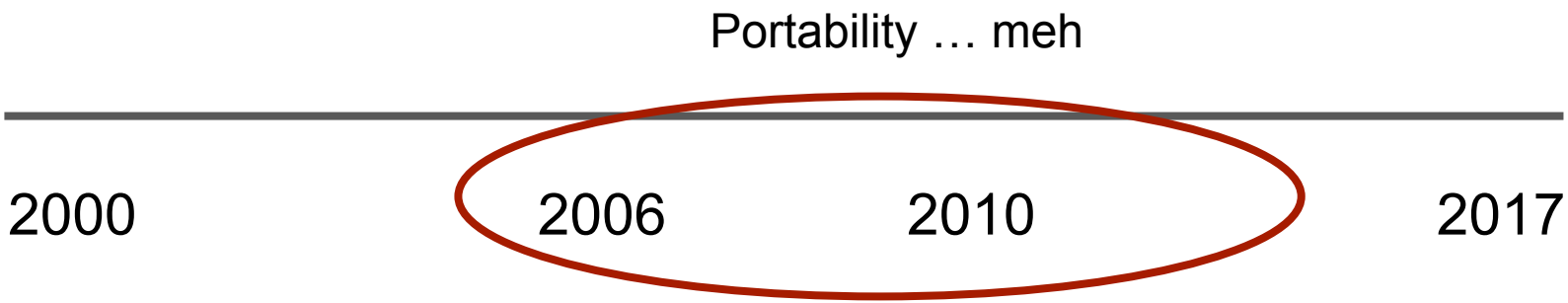
# C++ as a “Live at Head” Language

Titus Winters (titus@google.com)

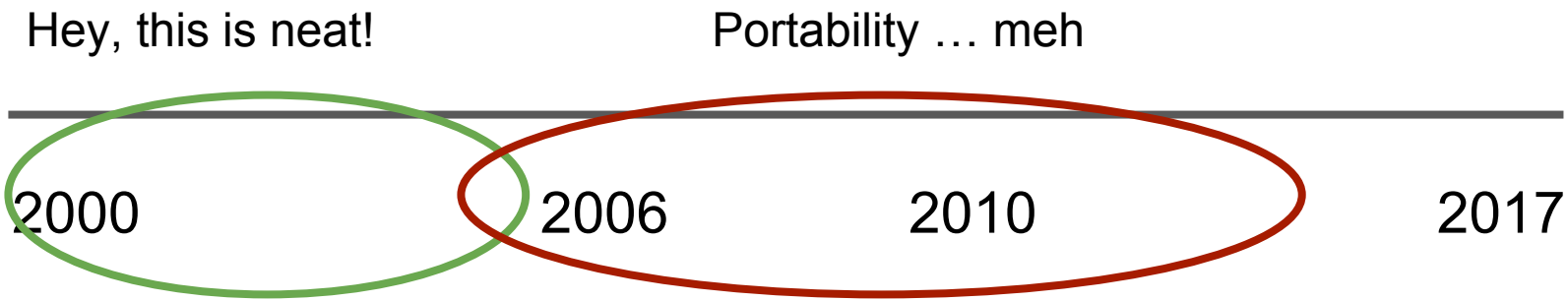
# History: Google C++



# History: Google C++

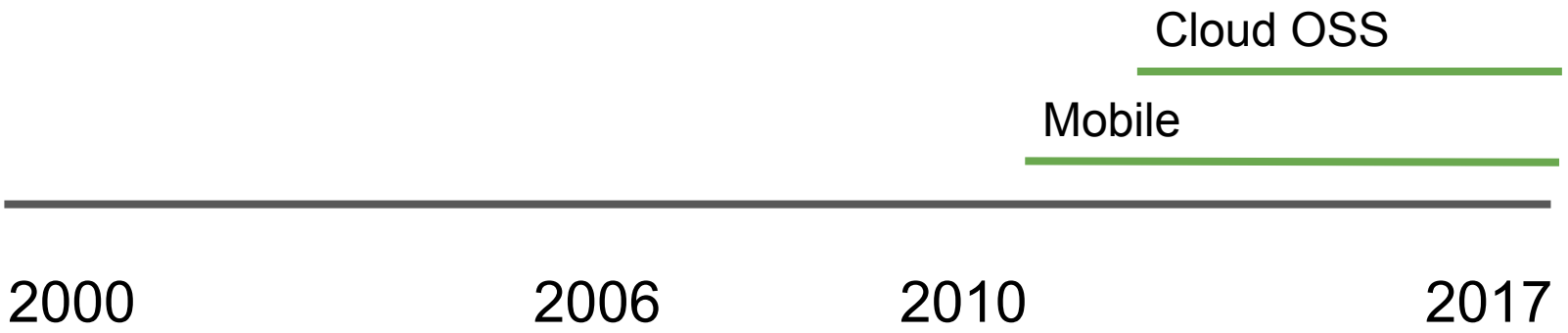
Hey, this is neat!

# History: Google C++

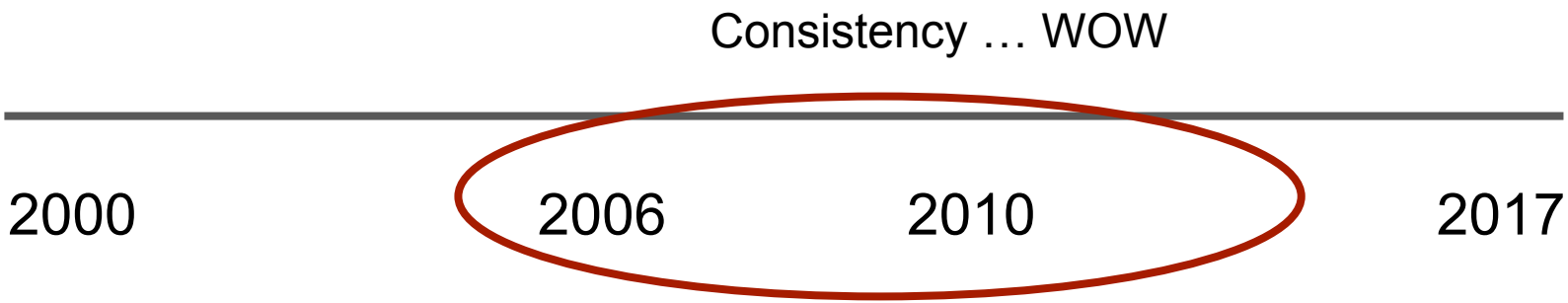


# History: Google C++

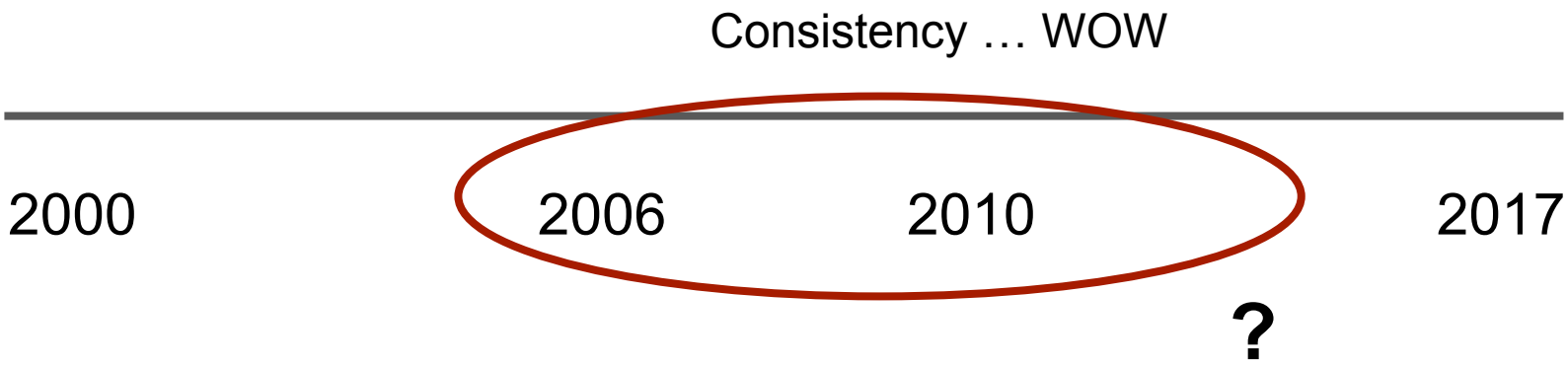
# History: Google C++



# History: Google C++



# History: Google C++



# Return to Open Sourcing C++ Libraries

# Do This In A Sustainable Way

# Software Engineering vs. Programming

Engineering is programming integrated over time.

# Required fields

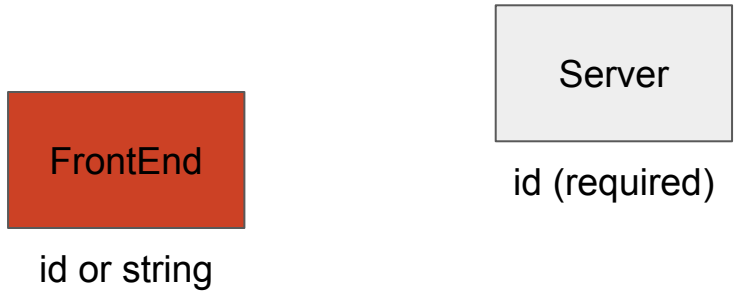
```
message Request {
   required int64 query_id = 17;
}
```

# Required fields

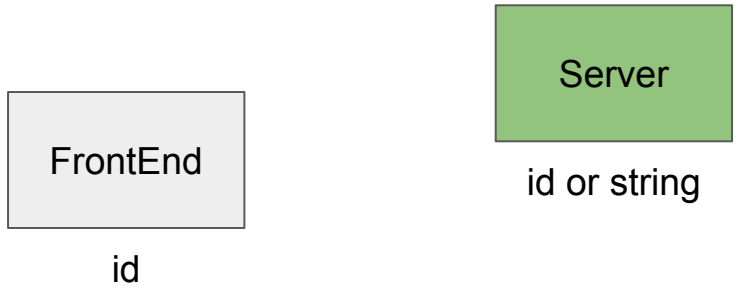
```
message Request {
   optional int64 query_id = 17;
   optional string query_string = 42;
}
```

# Required fields

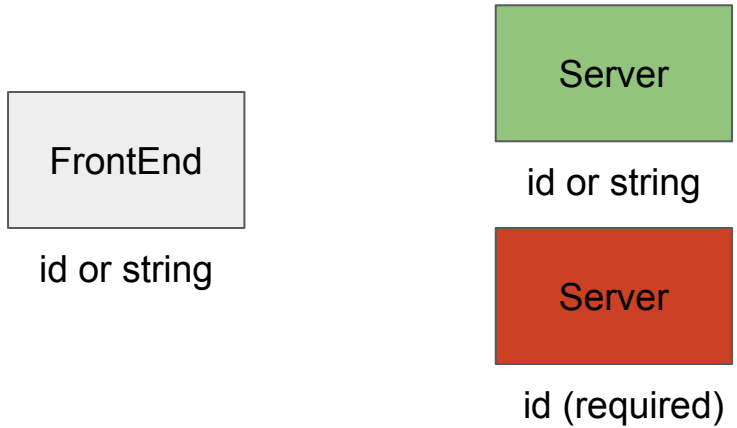
# Required fields



# Required fields



# Required fields



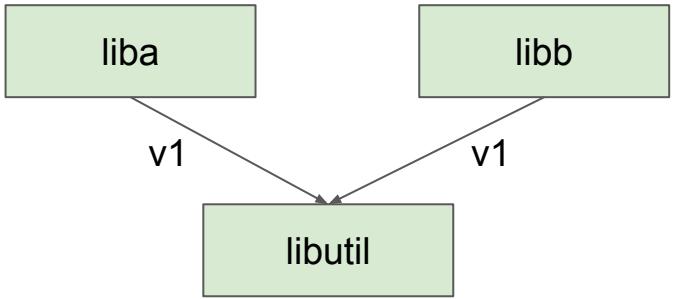
# Software Engineering is Resilience to Time

- Version Control Systems
- Continuous Integration
- Unittests
- Refactoring tools
- Design patterns
- Dependency management

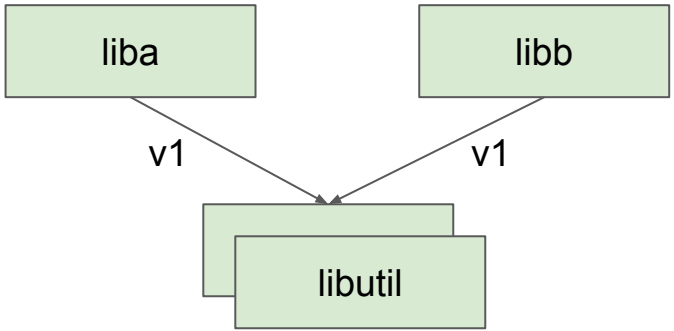
Software engineering is about resilience to change over time.

# Diamond Dependencies

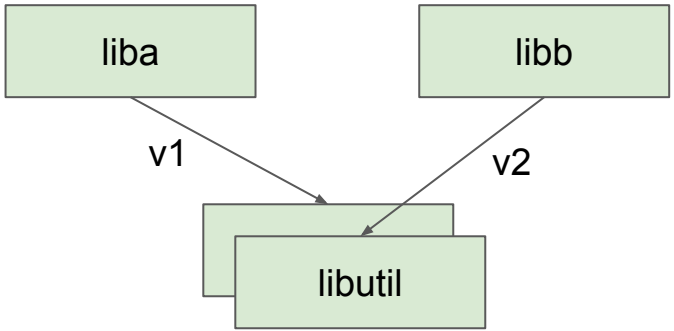
# Diamond Dependency



# Diamond Dependency

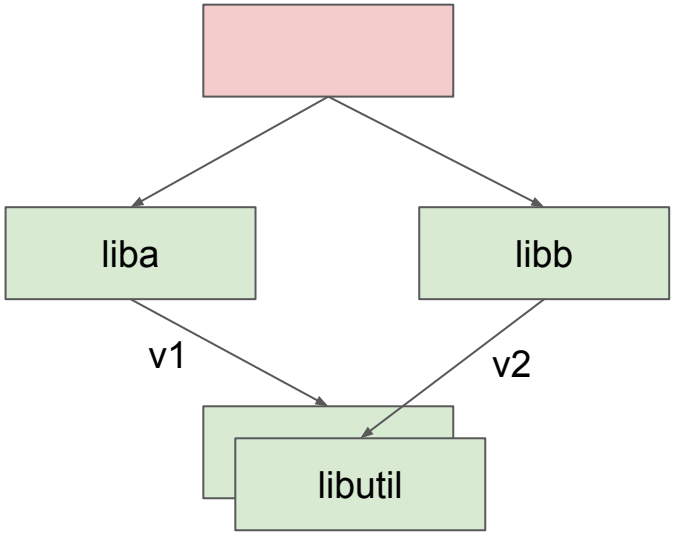


# Diamond Dependency




Version Skew

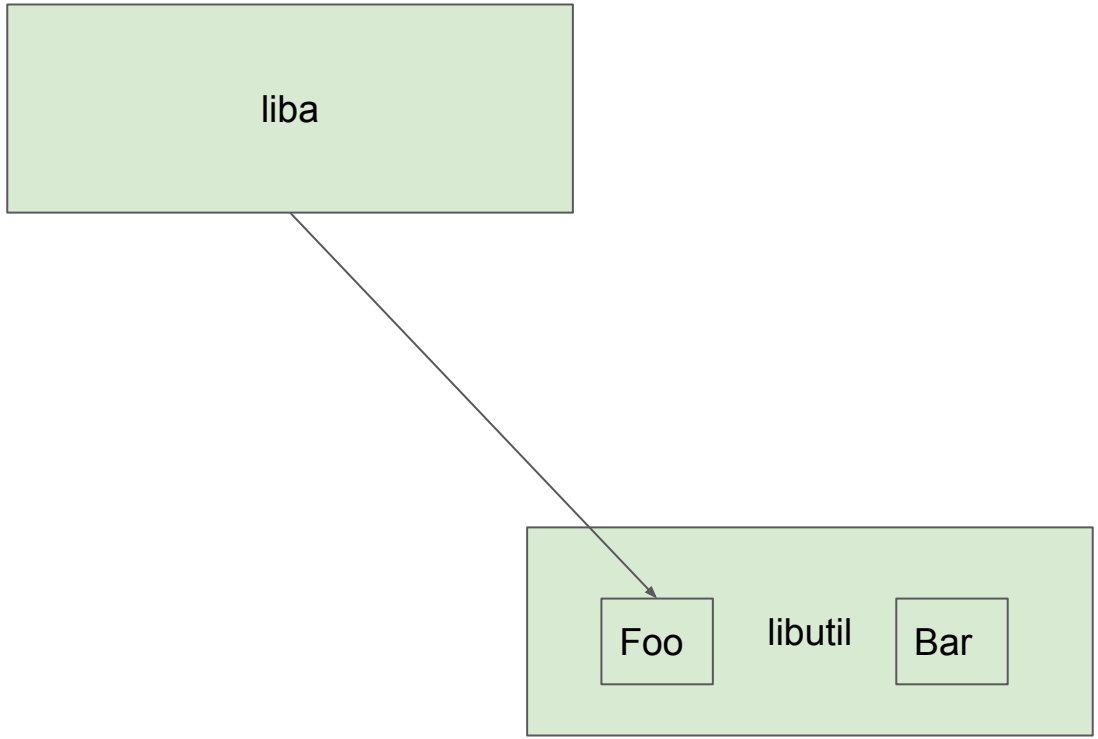
# Diamond Dependency

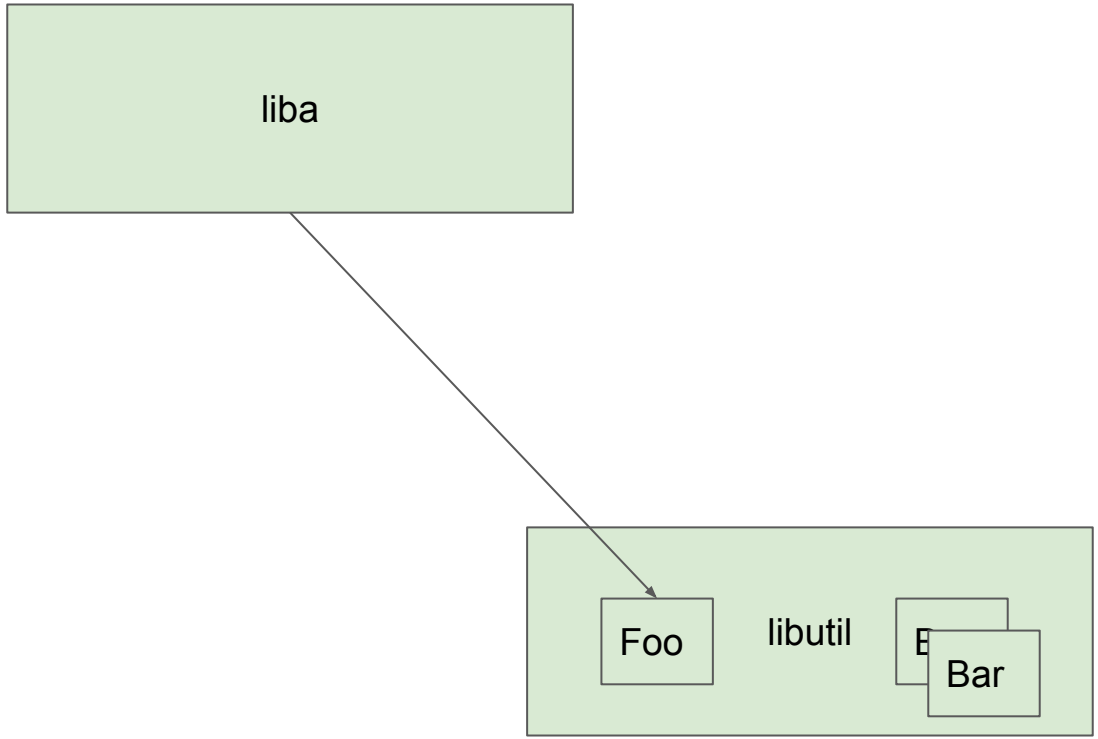


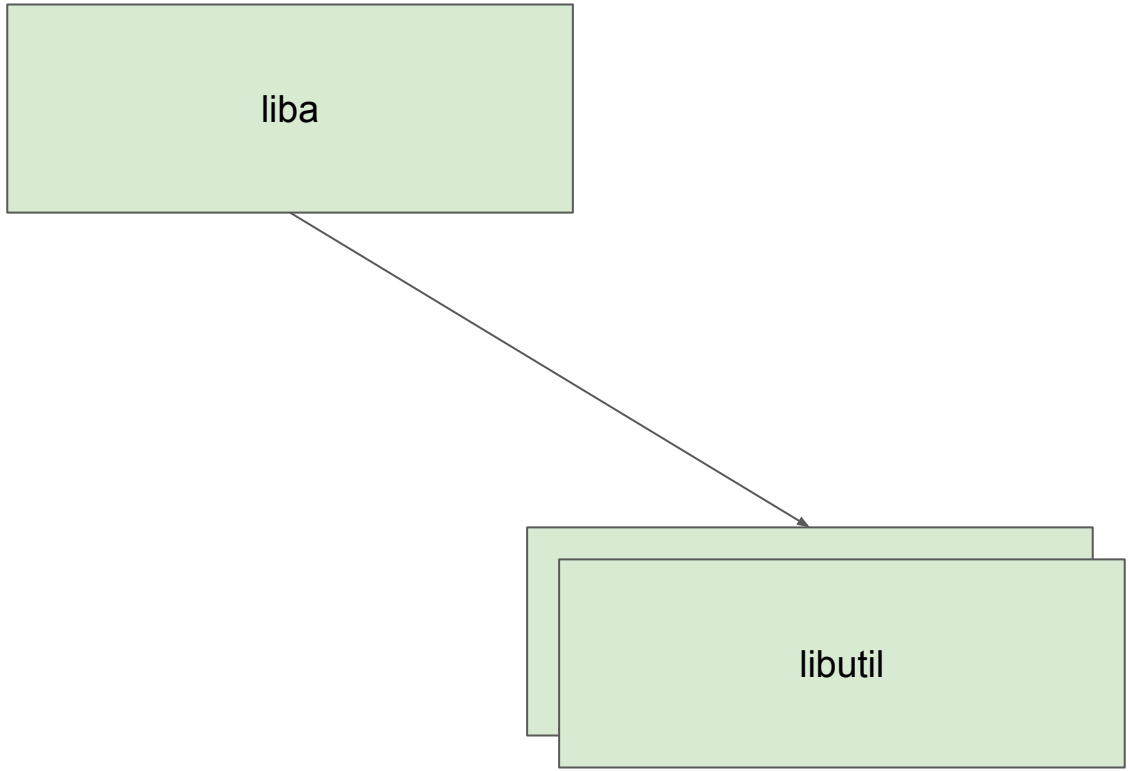
# Semantic Versioning (SemVer)

Release versions of the form *x.y.z* (1.3.17)

- Major number (API incompatibility)
- Minor number (Additional feature, but compatible)
- Patch number (bug fixes, etc)


liba
Foo
libutil
Bar

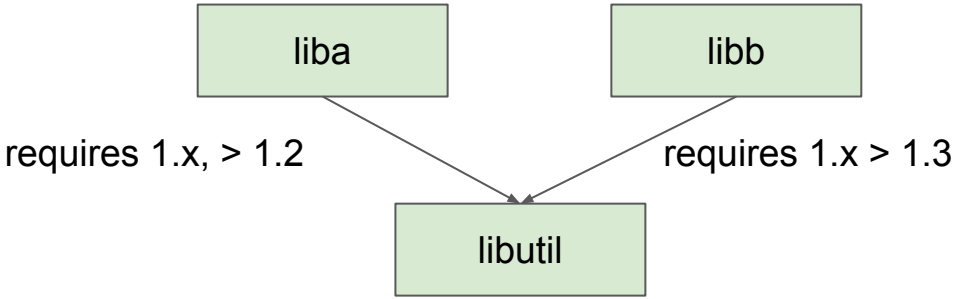

liba
Foo
libutil
Bar

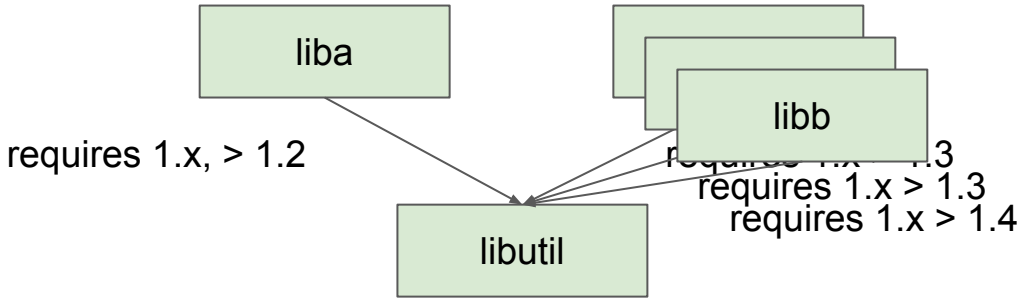

liba
libutil

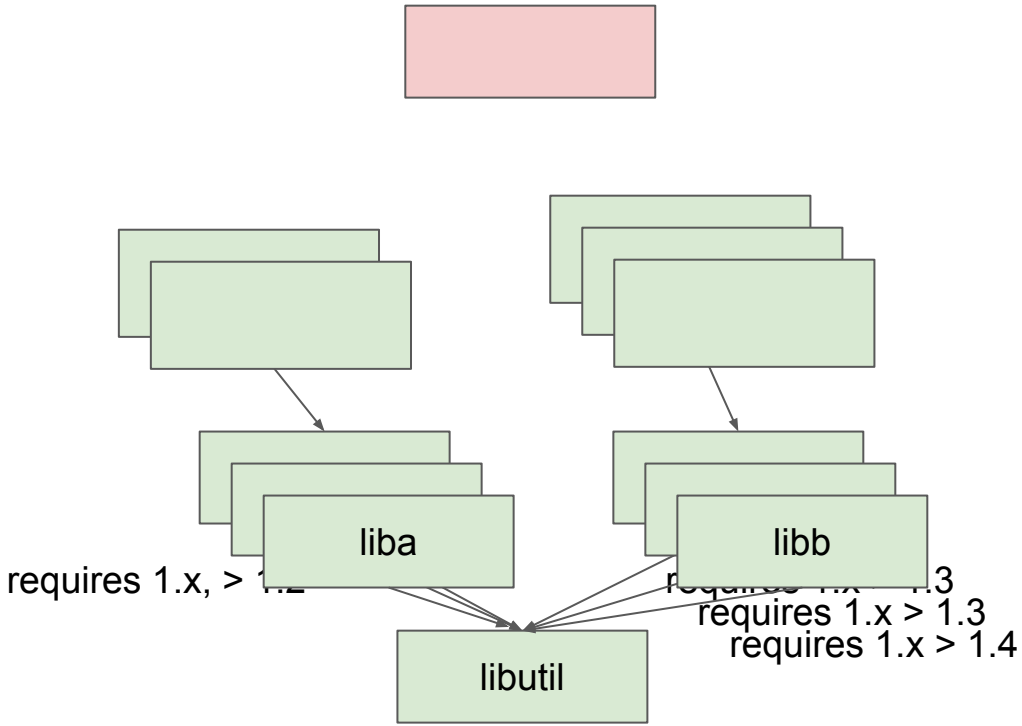
# Semantic Versioning (SemVer)

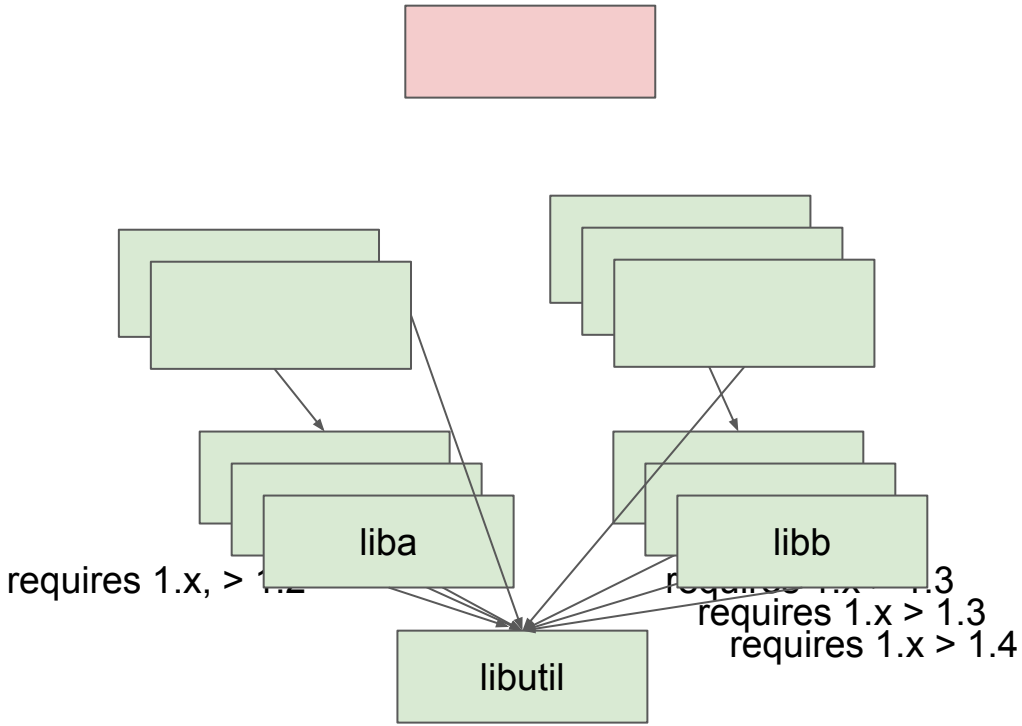
Release versions of the form *x.y.z* (1.3.17)

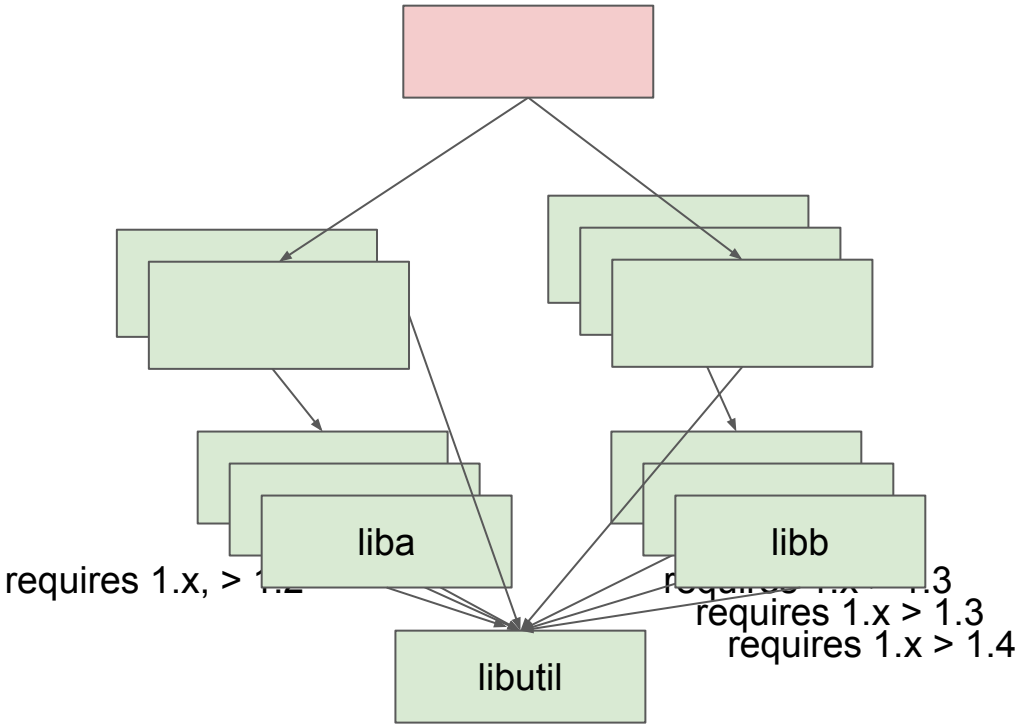
- Major number (API incompatibility)
- Minor number (Additional feature, but compatible)
- Patch number (bug fixes, etc)

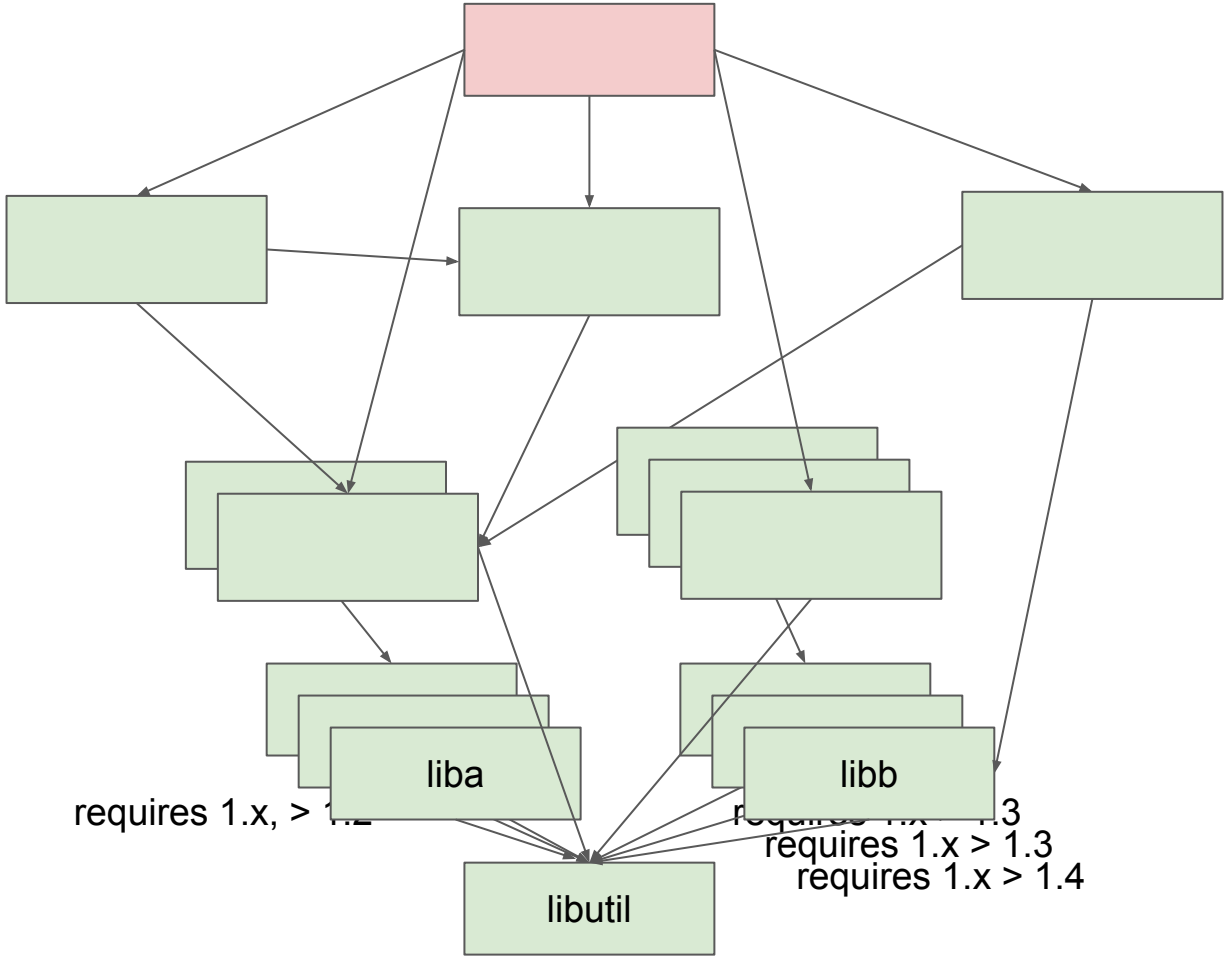

liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
libutil


liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
requires 1.x > 1.4
libutil


liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
requires 1.x > 1.3
requires 1.x > 1.4
libutil


liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
requires 1.x > 1.3
requires 1.x > 1.4
libutil


liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
requires 1.x > 1.3
requires 1.x > 1.4
libutil


liba
libb
requires 1.x, > 1.2
requires 1.x > 1.3
requires 1.x > 1.4
libutil

# SemVer

What’s the difference between a patch release and a major version?

# What Constitutes a “Breaking Change?”

What Constitutes a “Breaking Change?”

**Almost Everything.**

# Breaking Changes - Shades of Gray

Almost certainly fine: Adding whitespace or changing line numbers.

# Breaking Changes - Shades of Gray

Almost certainly fine: Adding whitespace or changing line numbers.

```
int Factorial(int n) {
  if (__LINE__ == 42) return 17;
  if (n <= 1) return 1;
  return n * Factorial(n - 1);
}
```

# Breaking Changes - Shades of Gray

Almost certainly fine: Adding whitespace or changing line numbers.

Certainly not fine: Removing an API.

# Breaking Changes

Almost certainly fine: Adding whitespace or changing line numbers.

???: Adding an overload.

Certainly not fine: Removing an API.

# Breaking Changes

???: Adding an overload.

```
auto* cb = NewCallback(&file::JoinPath,

                       basepath);
```

# Breaking Changes

Almost certainly fine: Adding whitespace or changing line numbers.

???: Adding an overload.

Certainly not fine: Removing an API.

# Breaking Changes

Almost certainly fine: Adding whitespace or changing line numbers.

???: Adding an overload.

???: Changing storage size or alignment of types.

???: Changing runtime efficiency.

Certainly not fine: Removing an API.

# Hyrum’s Law

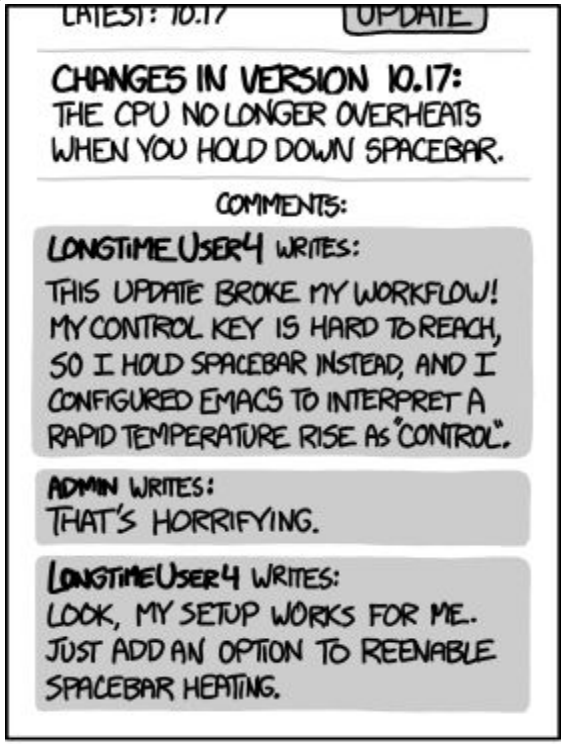
With a sufficient number of users of an API,

it does not matter what you promise in the contract,

all observable behaviors of your system

will be depended on by somebody.

# Breaking Changes



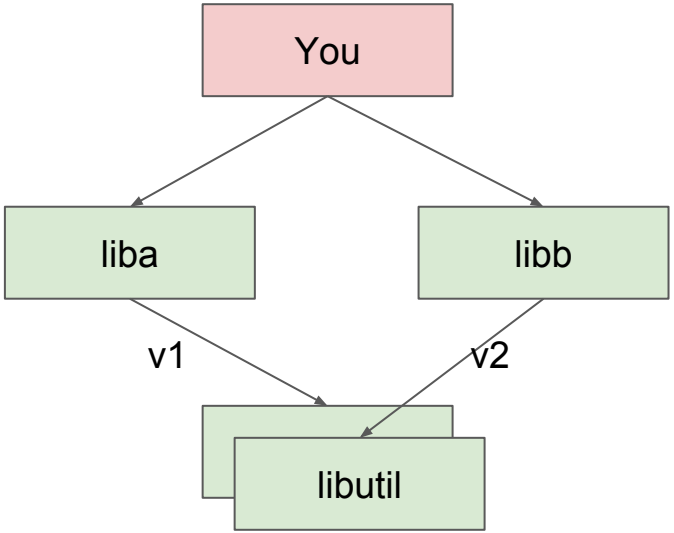

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

# SemVer - Recap

# Diamond Dependency

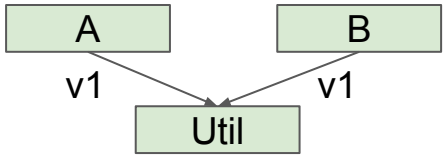
You aren’t an expert in:

- The code that has to change (liba)
- The code that did change (libutil)
- Whether it works

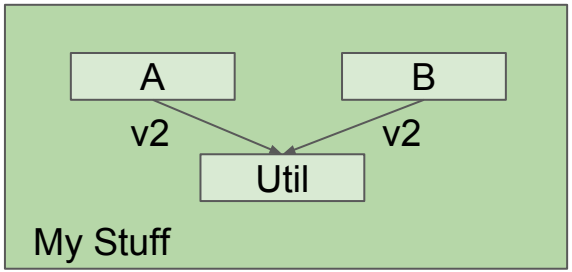


# Diamond Dependency Solutions

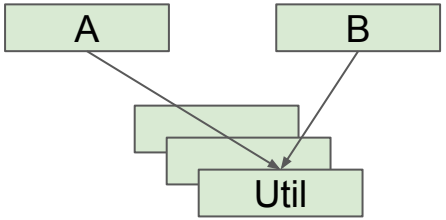
No breaking changes




Draw a bigger box

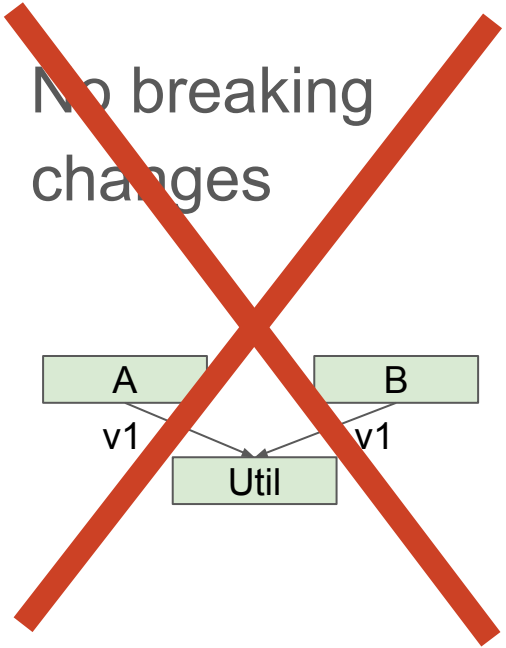



Easy-to-adopt changes



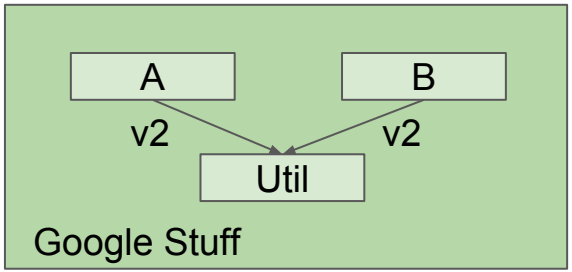
# Diamond Dependency Solutions

No breaking changes



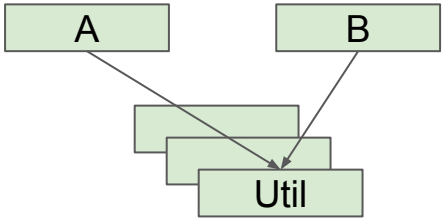

Fails over time

Draw a bigger box




Doesn’t scale

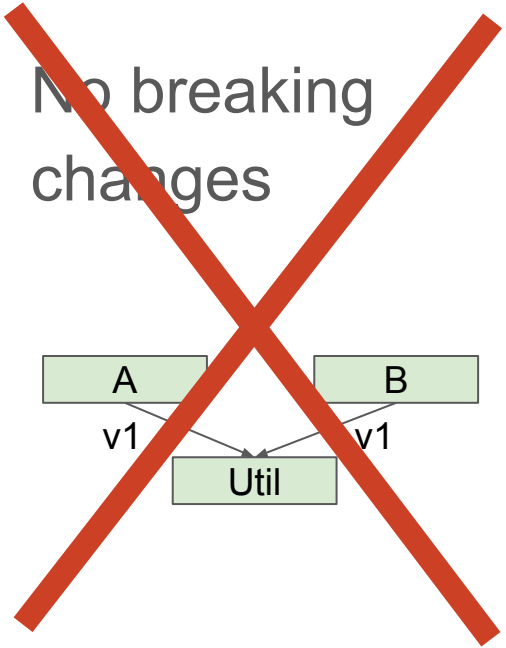
Only easy upgrades




Could work?

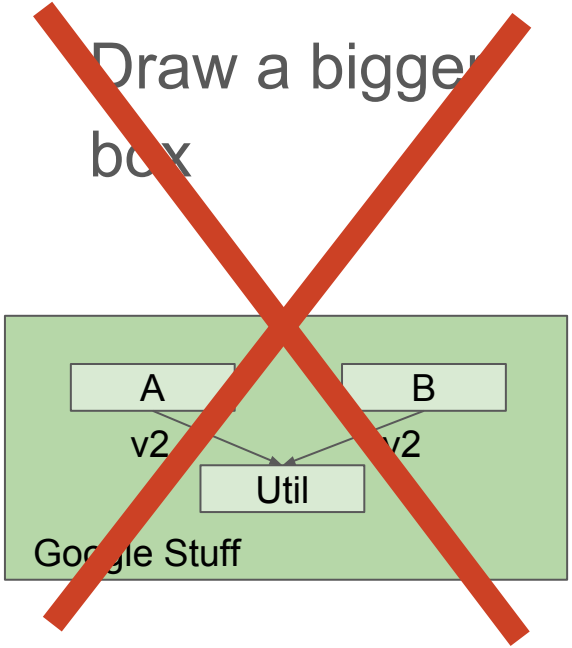
# Diamond Dependency Solutions

No breaking changes



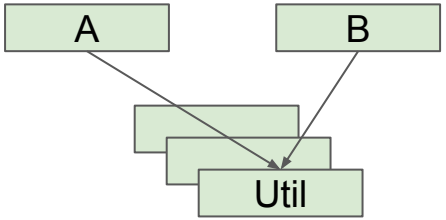

Fails over time

Draw a bigger box




Doesn't scale

Only easy upgrades




Could work?

# Easy Upgrades

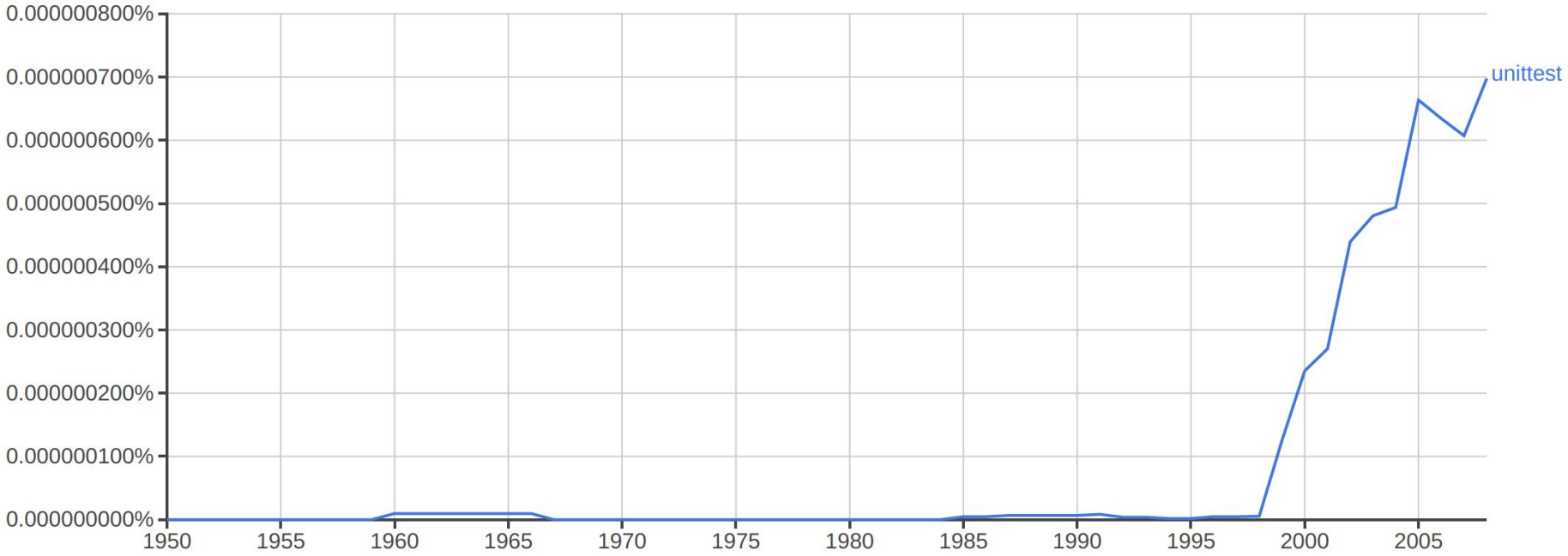
Diamond deps are hard:

- Know where/how to make the change
- Know what to change
- Know how to verify the change

(All in a project you don’t usually work in.)

# What’s happened recently?

Unittests are on the rise.



# Easy Upgrades

Diamond deps are hard:

- Know where/how to make the change
- Know what to change
- Know how to verify the change

(All in a project you don’t usually work in.)

# What’s happened recently?

Wright et al. “Large Scale Automated Refactoring Using ClangMR.” 2013. Proceedings of the 29th International Conference on Software Maintenance

CppCon2014: Hyrum Wright "Large-Scale Refactoring @ Google"

clang-tidy - an extensible platform for identifying and converting an old bad pattern into something better

# Easy Upgrades

Diamond deps are hard:

- Know where/how to make the change
- Know what to change
- Know how to verify the change

(All in a project you don’t usually work in.)

# Tools - Not A Silver Bullet

There are going to be cases where even a compiler-based tool gets tripped up.

There are also things where tools shouldn’t be necessary (updating internals, adding new APIs).

In the end, code that is using our libraries has to be at least minimally **well-behaved** before tools are going to suffice.

This is enough to solve diamond dependencies.

# This is enough to solve diamond dependencies.

- No API breaks without tools (easy upgrades)
- Users are well-behaved
- Unittests everywhere

This is enough to solve diamond dependencies.*

*for source distribution

This is enough to solve diamond dependencies.[1]

[1](for source distribution)[2]

[2] assuming there's only one "head" at a time

## Sad Reality

In C++, almost anything you do can break someone’s build.

# Compatibility Goals

Projects that intend to work over time should be clear what they promise, and what they require from you.

# Standard Compatibility (proposed)

We reserve the right to add things to namespace std.

- Do not add things to namespace std, except as directed

# Standard Compatibility (proposed)

We reserve the right to add things to namespace std.

- Do not add things to namespace std, except as directed (std::hash<MyT>)

# Standard Compatibility (proposed)

We reserve the right to add things to namespace std.

- Do not add things to namespace std, except as directed
- Do not forward declare things

# Compat: Forward declarations

```
namespace std {
    template<class Key, class Hash, class Cmp,
             class Alloc> class unordered_set;
}
```

# Standard Compatibility (proposed)

We reserve the right to add things to namespace std.

- Do not add things to namespace std, except as directed
- Do not forward declare things
- Assume the “call only” interface
  - Don’t take the address of functions
  - Don’t use template metaprogramming / introspection on type properties

# Compat: Call Only

```
void* (*a)(size_t, size_t, void*&, size_t&) =

    &std::align;
```

# Compat: Call Only

```
bool is_even(const int n) {
  std::vector<int> v;
  constexpr bool ebt =
    std::is_reference<decltype(v.emplace_back(1))>::value;
  if (ebt) return false;
  return (n & 1) == 0;
}
```

# Standard Compatibility (proposed)

We reserve the right to add things to namespace std.

- Do not add things to namespace std, except as directed
- Do not forward declare things
- Assume the “call only” interface
  - Don’t take the address of functions
  - Don’t use template metaprogramming / introspection on type properties

# Introducing Abseil

http://abseil.io

# Introducing Abseil

We’re releasing common libraries, with an **engineering** focus.

# Abseil

- 250MLoC+ already depends on this.
- 12K+ active developers already use this.
- Many external Google projects are moving toward this.
  - Protobuf, gRPC, TensorFlow
  - Chromium?
- First drop yesterday, (much) more to come

# Live At Head

# Things Abseil Needs

- We reserve the right to change implementation details.
- We reserve the right to add new APIs.
- We need the above to not require any action on behalf of any user.

# Abseil Compatibility

We reserve the right to add things to namespace absl

- Do not open namespace absl for any reason
- Do not rely on Argument Dependent Lookup (ADL)
- Do not forward declare things
- Do not make unqualified calls in the global namespace
- Assume the “call only” interface
- Do not rely on implementation details
- No `using namespace absl;`

# Abseil Compatibility

We reserve the right to change implementation details.

- ABI will not be stable
- Don’t depend on internals
  - Any namespace with “internal” in it
  - Any filename with “internal” in it.
  - #define private public - Not even once.
- Our #include graph may change - please IWYU

# What it takes to Live At Head

- Well behaved code
- Well maintained dependencies
  - Pin ill-maintained dependencies at some version
- Apply tools when necessary
- Run tests

# What is Abseil

- Zero config
- Utility code
- string routines
- Debugging / analysis facilities
- Guidance (Tip of the Week)
- C++11-compatible versions of standard types (pre-adopt)
- Standards-alternatives

General goal: Support 5 years back where possible.

# What is Abseil

5 year compat + zero config => Assume the standard, but work around if needed.

```
// ABSL_HAVE_THREAD_LOCAL
//
// Checks whether C++11's `thread_local` storage duration specifier is
// supported.
//
// Notes: Clang implements the `thread_local` keyword but Xcode did not support
// the implementation until Xcode 8.
#ifdef ABSL_HAVE_THREAD_LOCAL
#error ABSL_HAVE_THREAD_LOCAL cannot be directly set
#elif !defined(__apple_build_version__) || __apple_build_version__ >= 8000042
#define ABSL_HAVE_THREAD_LOCAL 1
#endif
```

# What is Abseil

String routines: Split, Join, stringify+concatenate.

```
// Split collections of string-ish things.
std::vector<std::string> v = absl::StrSplit("foo,bar,baz", ',');
EXPECT_THAT(v, ElementsAre("foo", "bar", "baz"));

std::vector<absl::string_view> v = absl::StrSplit("foo,bar,baz", ',');
std::list<absl::string_view> v = absl::StrSplit("foo,bar,baz", ',');
YourContainer<absl::string_view> v = absl::StrSplit("foo,bar,baz", ',');
```

# What is Abseil

String routines: Split, Join, stringify+concatenate.

```
// Joining collections of strings (or string_view, or char*, etc)
std::vector<std::string> v = {"foo", "bar", "baz"};
EXPECT_EQ("foo-bar-baz", absl::StrJoin(v, "-"));
```

# What is Abseil

String routines: Split, Join, stringify+concatenate.

```
// Variadic unformatted to-string+concatenate.
std::string s =
    absl::StrCat("Hello ", GetCppConName(), 2017);
```

# What is Abseil

## Debugging Facilities

- leakchecking - Build time ties into LeakSanitizer.
- stack traces - (If supported) get back the function pointers for your function stack
- AddressSanitizer / ThreadSanitizer support
- Static thread annotations

# What is Abseil

Pre-adopt C++17 types (in C++11)

- absl::string_view
- absl::optional
- absl::any
- (soon) absl::variant

# Wait … Types are Expensive!

Consider:

```
absl::optional<Foo> MaybeFoo();
```

# Wait … Types are Expensive!

Consider:

```
absl::optional<Foo> MaybeFoo();

// Must fix calls / assignment.
absl::optional<Foo> f = MaybeFoo();
```

# Wait … Types are Expensive!

Consider:

```
absl::optional<Foo> MaybeFoo();

// Must fix calls / assignment.
absl::optional<Foo> f = MaybeFoo();

// Must fix passing of those.
AcceptsOptionalFoo(f);
```

# Wait … Types are Expensive!

```
// Checks whether C++17 std::optional is available.
#ifdef __has_include
#if __has_include(<optional>) && __cplusplus >= 201703L
#define ABSL_HAVE_STD_OPTIONAL 1
#endif
#endif

#ifdef ABSL_HAVE_STD_OPTIONAL
#include <optional>
namespace absl {
using std::optional;
}
```

When using a new standard,
pre-adopted types melt away.

# Wait … Types are Expensive!

Consider:

```
absl::optional<Foo> MaybeFoo();

// Must fix calls / assignment.
absl::optional<Foo> f = MaybeFoo();

// Must fix passing of those.
AcceptsOptionalFoo(f);
```

# Wait … Types are Expensive!

## Consider ADL issues:

```
absl::string_view name = GetCppConName();

std::cout << StrCat(name, 2017) << std::endl;
```

# Abseil Compatibility

We reserve the right to add things to namespace absl

- Do not open namespace absl for any reason
- Do not rely on ADL
- Do not forward declare things
- Do not make unqualified calls in the global namespace
- Assume the “call only” interface
- Do not rely on implementation details

# What is Abseil - Guidance

Google’s C++ “Tip of the Week”

- 130+ essays
- “Effective C++”
- Longer-form explanations for Style Guide
- Largely compatible with Core Guidelines

# What is Abseil

## Standards alternatives?

aka - How to alienate my friends on the committee

# Standards Design Priorities

1. You do not pay for what you do not use.

# Standards Design Priorities

1. You do not pay for what you do not use.
2.

# Standards Design

For any problem space that the standard is solving, if there is runtime overhead for some design/feature on a reasonable platform/workload, we'll find an option to avoid it.

# Example: `std::chrono`

High Frequency Trading

- CPU costs on time ops add up
- Extreme precision (nanos)

Embedded Microcontroller

- 16-bit 1-second ticks

# Example: `std::chrono`

High Frequency Trading

- CPU costs on time ops add up
- Extreme precision (nanos)

Embedded Microcontroller

- 16-bit 1-second ticks

Compromise: class templates

```
std::chrono::duration<Rep, Period>
std::chrono::time_point<Clock, Duration>
```

# Standard alternatives

absl::Time / absl::Duration

- All operations are defined
- Saturating arithmetic (InfiniteFuture / InfiniteDuration)
- `absl::Now()` is usually far more optimized than your standard library equivalents

# Standard alternatives

absl/synchronization -

- absl::Mutex
  - Reader/Writer locks
  - Deadlock detection
  - Harder to misuse API

# std::mutex

worker.cc:

```
void Finish() {
  lock_->lock();
  shared_state_ += 1;
  lock_->unlock();
  cv_->notify_one();
}
```

waiter.cc:

```
void Wait() {
  lock_->lock();
  cv_->wait(*shared_lock_, []() {
    return shared_state_ == 1;
  });
  lock_->unlock();
}
```

# absl::Mutex

worker.cc:

```
void Finish() {
  lock_->Lock();
  shared_state_ += 1;
  lock_->Unlock();
}
```

waiter.cc:

```
void Wait() {
  lock_->Lock();
  lock_->Await(Condition([this]() {
      return shared_state_ == 1;
  }));
  lock_->Unlock();
}
```

# Standard alternatives

-DABSL_ALLOCATOR_NOTHROW

- Does allocation throw on your platform?
- If not, have some optimization.

# absl vs. std

We are **not** competing.

- These aren’t “better” designs, these are designs resulting from different priorities and legacies.
- Decide which set of priorities works for you
- The standard is still the right thing for interoperability.

# C++ - Live at Head

Challenges:

- No standard build flags/mode/etc
- One Definition Rule (ODR)

# C++ - Live at Head

Challenges:

- No standard build flags/mode/etc
- ODR
- Challenging / brittle language

# C++ - Live at Head

Challenges:

- No standard build flags/mode/etc
- ODR
- Challenging / brittle language
- No standard build system

# C++ - Live at Head

Opportunities:

- No standard package manager
- Good/consistent unittest systems

# C++ - Live at Head

Opportunities:

- No standard package manager
- Good/consistent unittest systems
- Compiler-based refactoring tools

# C++ - Live at Head

## Call to Action

# Call to Action

Consider engineering vs. programming

# Call to Action

Understand your dependencies

# Call to Action

Write well-behaved code

# Call to Action

Use up-to-date versions

# Call to Action

Apply tools

# Call to Action

Write tests

# Abseil - Next up

- Other Google OSS projects will build on top of Abseil
- Upcoming features:
  - Command line flags / logging
  - `absl::variant`
  - Hash containers
  - More debugging / runtime features
  - Random numbers

# Recommended Talks

- Matt Kulukundis - hashing (Wednesday)
- Jon Cohen - Type moving (Wednesday)
- Gennadiy Rozenthal - ABI stuff (Thursday)

Also, “Hands on With Abseil” (Today)



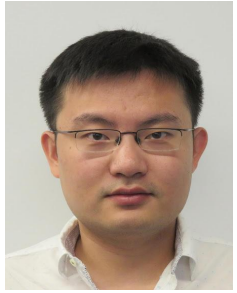




















# Abseil - Live at Head

# Questions?