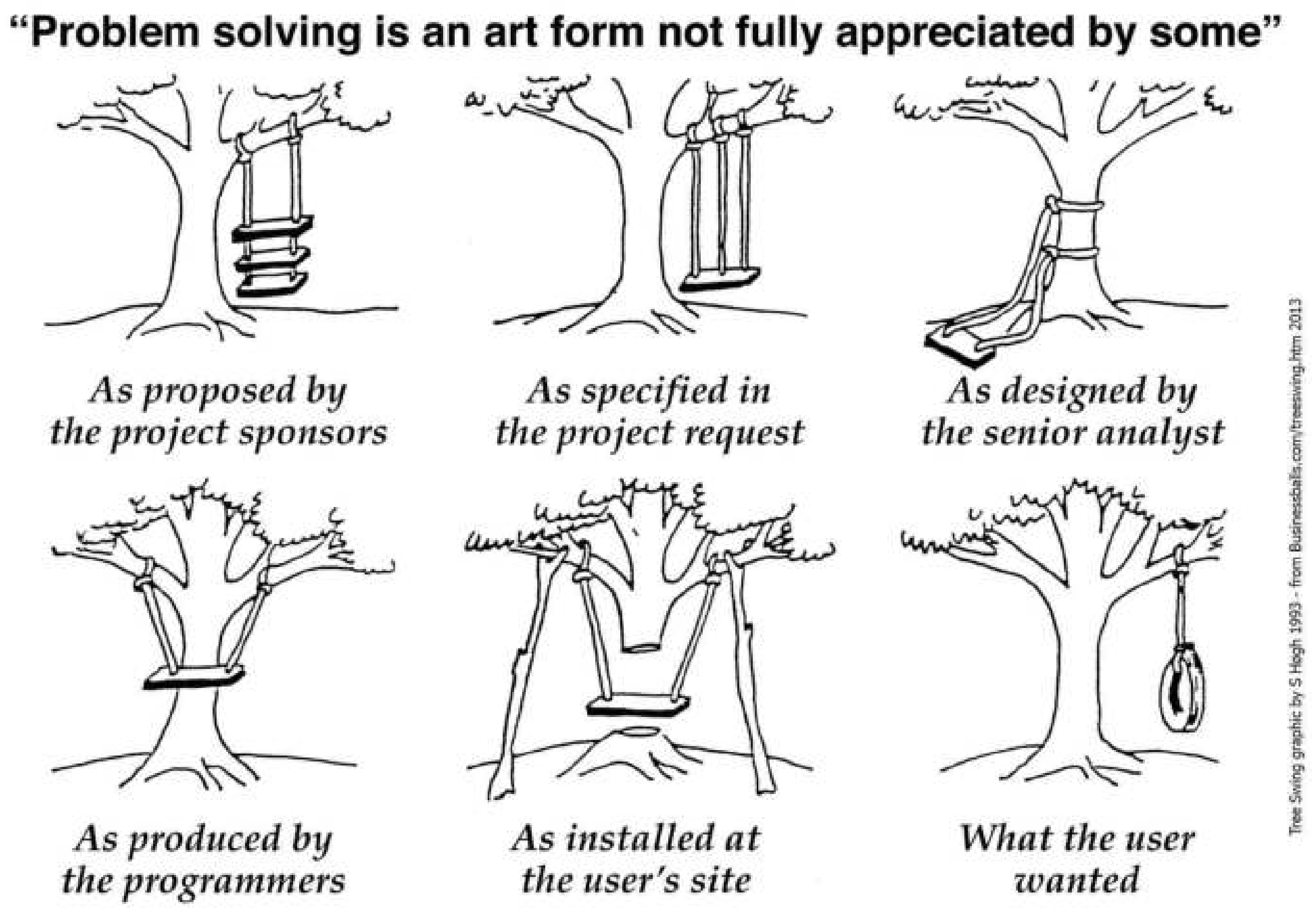
Software Engineering Is About Tradeoffs

Mateusz Pusz

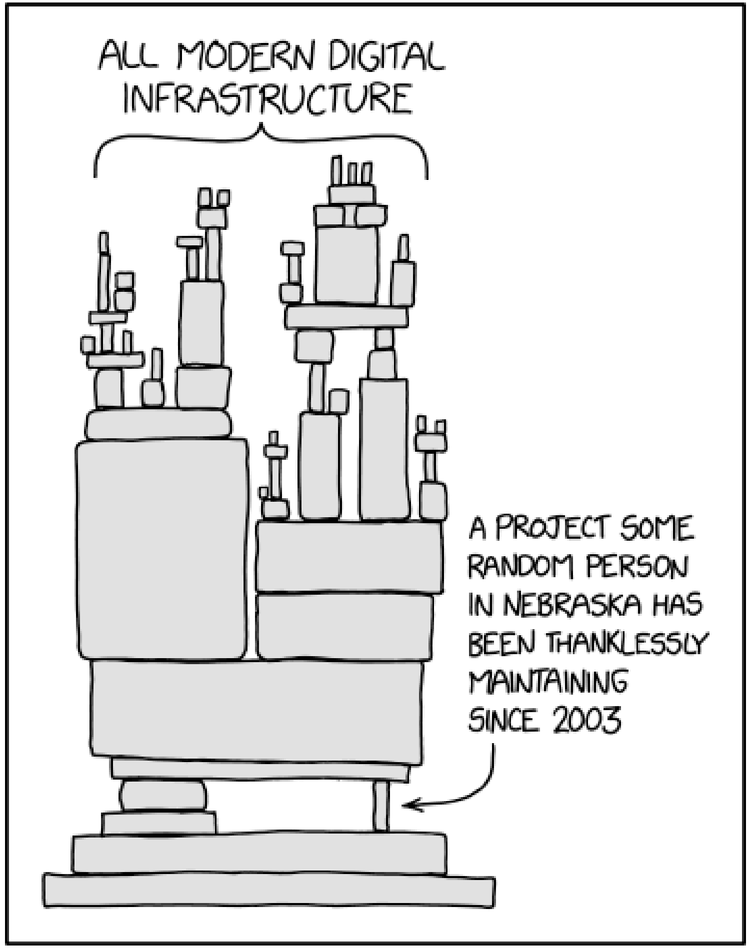
2021

THE ART OF THE SOFTWARE DESIGN

Problem Solving vs. Point Of View



Being Able To See The Whole Picture



The cost of finding the best solution

Let's make some design tradeoffs

The rest of the talk puts us in the product architect's shoes...

DIFFERENT VARIANTS OF EXCEPTION SAFETY GUARANTEES

Some Library

```
void establish_connections(int first, int last);

int main()
{
  try {
    establish_connections(1, 20);
    // ...
  }
  catch(const std::exception& ex) {
    std::cerr << "Unhandled exception: " << ex.what() << "\n";
  }
}
```

Some Library

```
struct connection {
  explicit connection(int port) : port_(port)
  {
    if(shit_happened())
      throw std::runtime_error("Can't connect!");
  }

  int port() const { return port_; }
private:
  int port_;
};
```

Some Library

```
struct connection {
  explicit connection(int port) : port_(port)
  {
    if(shit_happened())
      throw std::runtime_error("Can't connect!");
  }

  int port() const { return port_; }
private:
  int port_;
};
```

```
std::vector<connection> connections;
```

Some Library

```
void establish_connections(int first, int last)
{
  int retry_num = 10;
  while(connections.empty() || connections.back().port() != last) {
    try {


    }
    catch(const std::runtime_error& ex) {









    }
  }
}
```

Some Library

```
void establish_connections(int first, int last)
{
  int retry_num = 10;
  while(connections.empty() || connections.back().port() != last) {
    try {
      for(; first<=last; ++first)
        add_connection(first);
    }
    catch(const std::runtime_error& ex) {









    }
  }
}
```

Some Library

```
void establish_connections(int first, int last)
{
  int retry_num = 10;
  while(connections.empty() || connections.back().port() != last) {
    try {
      for(; first<=last; ++first)
        add_connection(first);
    }
    catch(const std::runtime_error& ex) {
      if(retry_num-- > 0) {
        std::cout << "Exception caught: " << ex.what() << "\n";
        std::cout << "Retrying...\n";
      }
      else {
        std::cout << "FAILED\n";
        throw;
      }
    }
  }
}
```

Some Library

```
void add_connection(int port)
{
  connections.emplace_back(port);
}
```

How often do you analyze exception safety of your code?

How often do you analyze exception safety of your code?

1. When was the last time you reviewed the *exception safety of your algorithm*?

How often do you analyze exception safety of your code?

1. When was the last time you reviewed the *exception safety of your algorithm*?
2. How often do you analyse *exception safety of your custom class assignment operator*?

How often do you analyze exception safety of your code?

1. When was the last time you reviewed the *exception safety of your algorithm*?
2. How often do you analyse *exception safety of your custom class assignment operator*?
3. Which *exception safety the latest library utility* written by you provides?

Exception Safety Guarantees

Exception Safety Guarantees

1. Nothrow (or nofail) exception guarantee
2. Strong exception guarantee
3. Basic exception guarantee
4. No exception guarantee

Exception Safety Guarantees

1. Nothrow (or nofail) exception guarantee
2. Strong exception guarantee
3. Basic exception guarantee
4. No exception guarantee

Only in case of "No exception guarantee" if the function throws an exception, the program may not be in a valid state: resource leaks, memory corruption, or other invariant-destroying errors may have occurred.

The importance of the Exception Safety Guarantees

Even if theoretically known to developers, Exception Safety Guarantees are **often ignored in the production code**!

Imagine the C++ Standard Library with only Basic Exception Safety Guarantee

Imagine the C++ Standard Library with only Basic Exception Safety Guarantee

```
void add_connection(int port)
{
  if(!connections.empty()) {
    // check if last emplace operation failed
    auto last_it = std::prev(connections.end());
    if(connections.dirty(last_it)) {
      // cleanup the container
      connections.erase(last_it);
    }

    // insert a new entry
    connections.emplace_back(port);
  }
}
```

What about `variant`?

std::variant<Types...>::valueless_by_exception

```
constexpr bool valueless_by_exception() const noexcept;
```
(since C++17)

Returns `false` if and only if the variant holds a value.

Notes

A variant may become valueless in the following situations:

- (guaranteed) an exception is thrown during the move initialization of the contained value from the temporary in copy assignment
- (guaranteed) an exception is thrown during the move initialization of the contained value during move assignment
- (optionally) an exception is thrown when initializing the contained value during a type-changing assignment
- (optionally) an exception is thrown when initializing the contained value during a type-changing `emplace`

Since variant is never permitted to allocate dynamic memory, previous value cannot be retained in these situations.

This applies even to variants of non-class types:

```
struct S {
    operator int() { throw 42; }
};
std::variant<float, int> v{12.f}; // OK
v.emplace<1>(S()); // v may be valueless
```

Example

```
struct basic_socket {
  // ...
};

struct fast_socket {
  fast_socket() = default;
  fast_socket(const fast_socket&)
  { throw std::runtime_error("Ouch!"); }
  // ...
};

using socket = std::variant<basic_socket, fast_socket>;
```

Example

```
struct basic_socket {
  // ...
};

struct fast_socket {
  fast_socket() = default;
  fast_socket(const fast_socket&)
  { throw std::runtime_error("Ouch!"); }
  // ...
};

using socket = std::variant<basic_socket, fast_socket>;
```

```
void process(socket& s)
{
  establish_connections(1, 20);
  std::cout << "Setting socket to a fast one\n";
  s = fast_socket{};
}
```

What about `variant`?

```
int main()
{
  try {
    socket s;
    try {
      process(s);
    }
    catch(const std::exception&) {
      if(s.valueless_by_exception()) {
        std::cout << "Resetting socket to a basic one\n";
        s = basic_socket{};
      }
    }
    // continue using socket
    // ...
  }
  catch(const std::exception& ex) {
    std::cerr << "Unhandled exception: " << ex.what() << "\n";
  }
}
```

"Never-Empty" Guarantee of Boost.Variant

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**
2. Destroy the content of the left-hand side

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**
2. Destroy the content of the left-hand side
3. Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**
2. Destroy the content of the left-hand side
3. Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side
4. In the event of failure, copy **backup** to the left-hand side storage

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**
2. Destroy the content of the left-hand side
3. Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side
4. In the event of failure, copy **backup** to the left-hand side storage
5. In the event of success, deallocate the data pointed to by **backup**

"Never-Empty" Guarantee of Boost.Variant

IN CASE OF A POTENTIALLY THROWING COPY-CONSTRUCTION

1. Copy-construct the content of the left-hand side to the heap as a **backup**
2. Destroy the content of the left-hand side
3. Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side
4. In the event of failure, copy **backup** to the left-hand side storage
5. In the event of success, deallocate the data pointed to by **backup**

Provides a strong exception safety with the potential cost of a dynamic memory allocation. **`boost::variant`** does not provide allocator support.

Poll: Choose the best solution

Poll: Choose the best solution

SOLUTION #1: `std::variant`

- Basic Exception Safety only
  - invalid state possible
  - manual cleanup may be needed
- Never allocates memory
  - fast
  - can be easily used in an embedded or constrained environment

Poll: Choose the best solution

SOLUTION #1: `std::variant`

- Basic Exception Safety only
  - invalid state possible
  - manual cleanup may be needed
- Never allocates memory
  - fast
  - can be easily used in an embedded or constrained environment

SOLUTION #2: `boost::variant`

- Strong Exception Safety
  - no invalid states
  - no need for a manual cleanup
- Dynamic memory allocations
  - slow
  - may fail
  - no allocator support
  - no-go for some projects (i.e. embedded, safety-critical, hard real-time, ...)

QUANTITY CREATION HELPERS

`mp-units`

Quantity

> Property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed by means of a number and a reference. A reference can be a measurement unit, …
>
> *-- ISO 80000*

Quantity

Property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed by means of a number and a reference. A reference can be a measurement unit, …

-- ISO 80000

EXAMPLE

- 123 kilometers of length
- 70 kilometers per hour of speed

units::quantity class template

```
template<Dimension D, UnitOf<D> U, Representation Rep = double>
class units::quantity;
```

`units::quantity` class template

```
template<Dimension D, UnitOf<D> U, Representation Rep = double>
class units::quantity;
```

```
quantity<si::dim_length, si::kilometre> d(123);
quantity<si::dim_speed, si::kilometre_per_hour, int> v(70);
```

`units::quantity` class template

```
template<Dimension D, UnitOf<D> U, Representation Rep = double>
class units::quantity;
```

```
quantity<si::dim_length, si::kilometre> d(123);
quantity<si::dim_speed, si::kilometre_per_hour, int> v(70);
```

Above quantity type is verbose to type so some helpers should be provided to improve users experience.

Solution #1: Dimension-Specific Aliases

- Helper aliases provided for *quantities of each dimension*

```
namespace si {

template<Unit U, Representation Rep = double>
using length = quantity<dim_length, U, Rep>;

template<Unit U, Representation Rep = double>
using speed = quantity<dim_speed, U, Rep>;

}
```

Solution #1: Dimension-Specific Aliases: Example

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

Solution #1: Dimension-Specific Aliases

PROS

- Cheap to standardize (only one per dimension)
- Fast to compile (alias template)
- Works both for literals and regular variables

Solution #1: Dimension-Specific Aliases

PROS

- Cheap to standardize (only one per dimension)
- Fast to compile (alias template)
- Works both for literals and regular variables

CONS

- Still quite verbose to type
  - i.e. namespace has to be repeated for a unit
- CTAD for alias templates does not work
  - cannot deduce a representation type from the initializer

Solution #2: Unit-Specific Aliases

- Provided for *quantities of each unit in the library*

```
namespace units::aliases::isq::si::inline length {

template<Representation Rep = double>
using km = units::isq::si::length<units::isq::si::kilometre, Rep>;

}

namespace units::aliases::isq::si::inline speed {

template<Representation Rep = double>
using km_per_h = units::isq::si::speed<units::isq::si::kilometre_per_hour, Rep>;

}
```

Solution #2: Unit-Specific Aliases: Example

```
using namespace units::aliases::isq;
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

Solution #2: Unit-Specific Aliases: Example

```
using namespace units::aliases::isq;
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

- With C++20 CTAD for alias templates

```
using namespace units::aliases::isq;
si::length::km d(123.);
si::speed::km_per_h v(70);
```

Solution #2: Unit-Specific Aliases: Example

```
using namespace units::aliases::isq;
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

- With C++20 CTAD for alias templates

```
using namespace units::aliases::isq;
si::length::km d(123.);
si::speed::km_per_h v(70);
```

- Possibility to be more terse if desired

```
using namespace units::aliases::isq::si;
auto d = km(123.);
auto v = km_per_h(70);
```

Solution #2: Unit-Specific Aliases

PROS

- Fast to compile (alias template)
- Works both for literals and regular variables
- User can chose to either use a long or terse version

Solution #2: Unit-Specific Aliases

PROS

- Fast to compile (alias template)
- Works both for literals and regular variables
- User can chose to either use a long or terse version

CONS

- Expensive to standardize (every unit of every dimension)

Solution #3: Quantity References

- Provided for *named units only*

```
namespace length_references {

inline constexpr auto km = reference<dim_length, kilometre>{};

}  // namespace length_references

namespace time_references {

inline constexpr auto h = reference<dim_time, hour>{};

}  // namespace time_references

namespace references {

using namespace length_references;
using namespace time_references;

}  // namespace references
```

Solution #3: Quantity References: Example

```cpp
using namespace units::isq::si::references;
auto d = 123. * km;        // si::length<si::kilometre, double>
auto v = 70 * (km / h);    // si::speed<si::kilometre_per_hour, int>
```

Solution #3: Quantity References: Example

```cpp
using namespace units::isq::si::references;
auto d = 123. * km;        // si::length<si::kilometre, double>
auto v = 70 * (km / h);    // si::speed<si::kilometre_per_hour, int>
```

- It is also possible to easily define custom quantity references from existing ones

```cpp
inline constexpr auto km_per_h = km / h;
auto v = 70 * km_per_h;    // si::speed<si::kilometre_per_hour, int>
```

Solution #3: Quantity References

PROS

- Medium effort to standardize as provided only for named units
- Works both for literals and regular variables
- Easy to compose custom references for unnamed derived units

Solution #3: Quantity References

PROS

- Medium effort to standardize as provided only for named units
- Works both for literals and regular variables
- Easy to compose custom references for unnamed derived units

CONS

- Slower to compile (class template instantiation)
- Sometimes awkward to type
  - `20 * (m / s) / (10 * (m / (s * s)))`
- Objects with short names often shadow user's local variables (i.e. **`m`**, **`t`**, **`N`**, ...)
- Sometimes hard to understand

```
constexpr Speed auto avg_speed(double d, double t)
{
  return d * m / (t * s);
}
```

Solution #4: User Defined Literals

- *Integral and a floating-point version* provided for *quantity of each unit*

```cpp
inline namespace literals {

constexpr auto operator"" q_km(unsigned long long l) {
  return length<kilometre, std::int64_t>(l);
}
constexpr auto operator"" q_km(long double l) {
  return length<kilometre, long double>(l);
}

constexpr auto operator"" q_km_per_h(unsigned long long l) {
  return speed<kilometre_per_hour, std::int64_t>(l);
}
constexpr auto operator"" q_km_per_h(long double l) {
  return speed<kilometre_per_hour, long double>(l);
}

}  // namespace literals
```

Solution #4: User Defined Literals: Example

```
using namespace units::isq::si::literals;
auto d = 123.q_km;      // si::length<si::kilometre, long double>
auto v = 70q_km_per_h; // si::speed<si::kilometre_per_hour, std::int64_t>
```

Solution #4: User Defined Literals: Example

PROS

- Compatible with **`std::chrono::duration`**
- Terse and easy to understand

Solution #4: User Defined Literals: Example

PROS

- Compatible with `std::chrono::duration`
- Terse and easy to understand

CONS

- The slowest to compile
- The most expensive to standardize (2 instances per every unit)
- Works only for literals (not for common variables)
- No control over the representation type (only `std::int64_t` or `long double`)
- Cannot be disambiguated with the namespace name
  - i.e. collisions between `cm` in SI and CGS

Standardization takes time

- We could provide all of the options for standardization...
- ... but most probably it would not be accepted
- Time needed to
  - discuss in working groups
  - prepare the ISO specification
  - implement in various implementations of the C++ Standard Library
  - teach and learn by the C++ Community

Comparison

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

SOLUTION #2: UNIT ALIASES

```
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

SOLUTION #2: UNIT ALIASES

```
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

```
auto d = km(123.);
auto v = km_per_h(70);
```

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

SOLUTION #2: UNIT ALIASES

```
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

```
auto d = km(123.);
auto v = km_per_h(70);
```

SOLUTION #3: QUANTITY REFERENCES

```
auto d = 123. * km;
auto v = 70 * (km / h);
```

Comparison

SOLUTION #1: DIMENSION ALIASES

```
si::length<si::kilometre> d(123);
si::speed<si::kilometre_per_hour, int> v(70);
```

SOLUTION #2: UNIT ALIASES

```
si::length::km<> d(123);
si::speed::km_per_h<int> v(70);
```

```
auto d = km(123.);
auto v = km_per_h(70);
```

SOLUTION #3: QUANTITY REFERENCES

```
auto d = 123. * km;
auto v = 70 * (km / h);
```

SOLUTION #4: UDLS

```
auto d = 123.q_km;
auto v = 70q_km_per_h;
```

Poll: Choose the best solution

FEATURE	#1 DIMENSION ALIASES	#2 UNIT ALIASES	#3 REFERENCES	#4 UDLS
Literals and variables support	Yes	Yes	Yes	Literals only
Preserves user provided representation type	No	Yes	Yes	No
Explicit control over the representation type	Yes	Yes	No	No
Readability	Medium	Good	Medium	Good
Possibility to resolve ambiguity	Yes	Yes	Yes	No
Hard to resolve shadowing issues	No	No	Yes	No
Controlled verbosity	No	Yes	No	No
Easy composition for derived units	No	No	Yes	No
Implementation and standardization effort	Lowest	High	Medium	Highest
Compile-time performance	Fastest	Fast	Medium	Slowest

THE DOWNCASTING FACILITY

`mp-units`

The Downcasting Facility In Action

The Downcasting Facility In Action

```cpp
using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

The Downcasting Facility In Action

```
using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

The Downcasting Facility In Action

```cpp
using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```cpp
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

```
70 km/h
```

The Downcasting Facility In Action

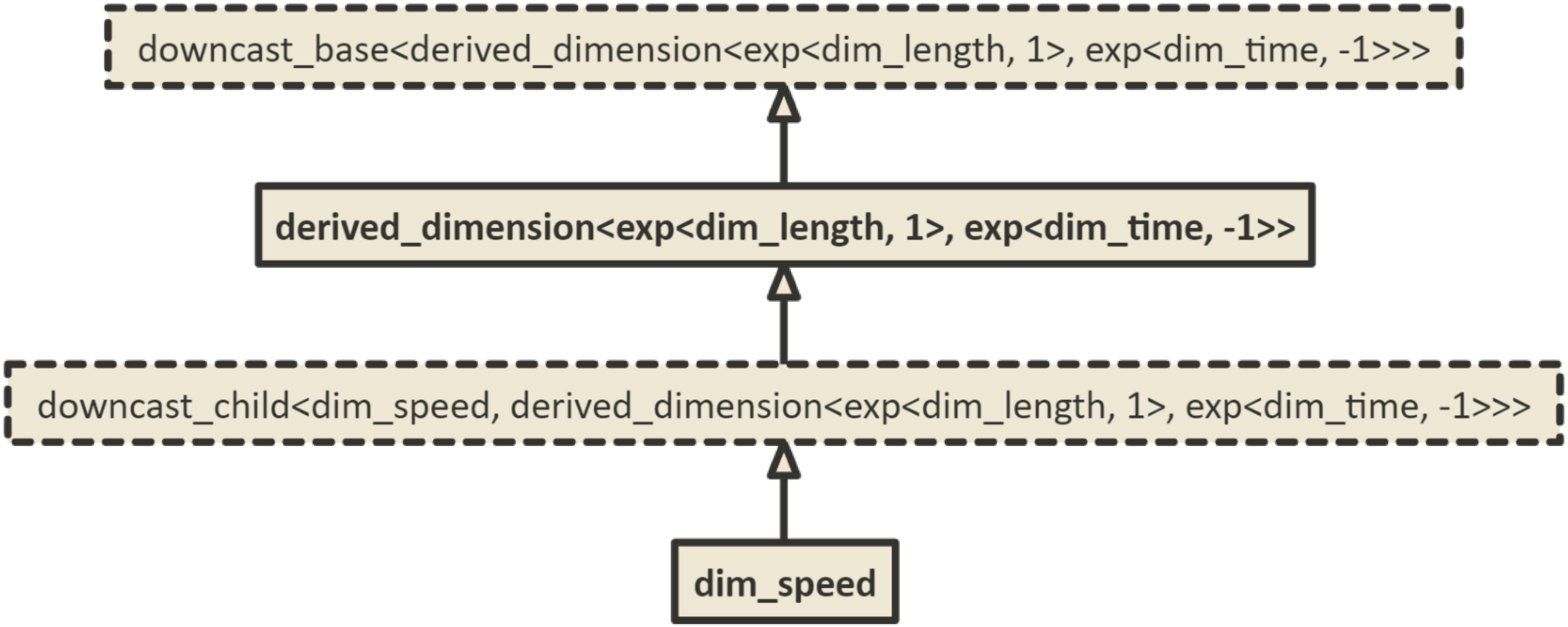
```
using namespace units::isq;

constexpr Speed auto avg_speed(Length auto d, Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

```
using namespace units::isq::si::references;
auto s = avg_speed(140 * km, 2 * h);
```

```
(gdb) ptype s
type = class units::quantity<units::isq::si::dim_speed, units::isq::si::kilometre_per_hour, int>
[with D = units::isq::si::dim_speed, U = units::isq::si::kilometre_per_hour, Rep = int] {
...
```

Downcasting facility (Version 2.0): Overview



One Definition Rule (ODR) - Translation Unit

Only one definition of any variable, function, class type, enumeration type, concept or template is allowed in any one translation unit.

One Definition Rule (ODR) - Program

- One and only one definition of every non-inline function or variable that is odr-used is required to appear in the entire program
- For an inline function or inline variable, a definition is required in every translation unit where it is odr-used
- One and only one definition of a class is required to appear in any translation unit where the class is used in a way that requires it to be complete

One Definition Rule (ODR) - Program

- One and only one definition of every non-inline function or variable that is odr-used is required to appear in the entire program
- For an inline function or inline variable, a definition is required in every translation unit where it is odr-used
- One and only one definition of a class is required to appear in any translation unit where the class is used in a way that requires it to be complete

but ...

One Definition Rule (ODR) - Program

There can be **more than one definition in a program**, as long as

- each definition appears in a *different translation unit*
- each definition *consists of the same sequence of tokens*
- *name lookup* from within each definition *finds the same entities*
- overloaded operators, including conversion, allocation, and deallocation functions refer to the same function
- the language *linkage is the same*

If all these requirements are satisfied, the program behaves as if there is only one definition in the entire program. Otherwise, the behavior is undefined.

Problem: One Definition Rule Violation

avg_speed.h

```
constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

Problem: One Definition Rule Violation

avg_speed.h

```
constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

file_1.cpp

```
#include "avg_speed.h"
#include <units/isq/si/speed.h>

void foo()
{
  auto s = avg_speed(140 * km, 2 * h);
  // ...
}
```

```
70 km/h
```

Problem: One Definition Rule Violation

avg_speed.h

```
constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

file_1.cpp

```
#include "avg_speed.h"
#include <units/isq/si/speed.h>

void foo()
{
  auto s = avg_speed(140 * km, 2 * h);
  // ...
}
```

```
70 km/h
```

file_2.cpp

```
#include "avg_speed.h"


void boo()
{
  auto s = avg_speed(140 * km, 2 * h);
  // ...
}
```

```
70 [1/36 × 10¹] m/s
```

But this is how some customization points behave...

ab.h

```
struct A { int value; };
struct B { int value; A* a; };

std::ostream& operator<<(std::ostream& os, const B& b)
{ return os << "[" << b.value << ", " << b.a->value << "]"; }
```

But this is how some customization points behave...

ab.h

```
struct A { int value; };
struct B { int value; A* a; };

std::ostream& operator<<(std::ostream& os, const B& b)
{ return os << "[" << b.value << ", " << b.a->value << "]"; }
```

file_1.cpp

```
#include "ab.h"

void swap(B& lhs, B& rhs) noexcept
{ std::ranges::swap(lhs.value, rhs.value); }

void foo()
{
  A a1{1}, a2{2};
  B b1{1, &a1}, b2{2, &a2};
  std::ranges::swap(b1, b2);
  std::cout << "b1: " << b1 << ", b2: " << b2 << "\n";
}
```

```
b1: [2, 1], b2: [1, 2]
```

But this is how some customization points behave...

ab.h

```
struct A { int value; };
struct B { int value; A* a; };

std::ostream& operator<<(std::ostream& os, const B& b)
{ return os << "[" << b.value << ", " << b.a->value << "]"; }
```

file_1.cpp

```
#include "ab.h"

void swap(B& lhs, B& rhs) noexcept
{ std::ranges::swap(lhs.value, rhs.value); }

void foo()
{
  A a1{1}, a2{2};
  B b1{1, &a1}, b2{2, &a2};
  std::ranges::swap(b1, b2);
  std::cout << "b1: " << b1 << ", b2: " << b2 << "\n";
}
```

```
b1: [2, 1], b2: [1, 2]
```

file_2.cpp

```
#include "ab.h"



void boo()
{
  A a1{1}, a2{2};
  B b1{1, &a1}, b2{2, &a2};
  std::ranges::swap(b1, b2);
  std::cout << "b1: " << b1 << ", b2: " << b2 << "\n";
}
```

```
b1: [2, 2], b2: [1, 1]
```

However most will not compile without the definition

ab.h

```
struct A { int value; };
struct B { int value; A* a; };

// std::ostream& operator<<(std::ostream& os, const B& b)
// {
//   return os << "[" << b.value << ", "
//             << b.a->value << "]";
// }
```

file_1.cpp

```
#include "ab.h"

void foo()
{
  A a1{1}, a2{2};
  B b1{1, &a1}, b2{2, &a2};
  std::cout << "b1: " << b1 << ", b2: " << b2 << "\n";
}
```

```
<source>: In function 'void foo()':
<source>:24:23: error: no match for 'operator<<' (operand types are 'std::basic_ostream<char>' and 'B')
   24 |   std::cout << "b1: " << b1 << ", b2: " << b2 << "\n";
      |   ~~~~~~~~~~~~~~~~~~~ ^~ ~~
      |             |            |
      |             |            B
      |            std::basic_ostream<char>
```

Solution #1: Keep it as it is

- We get **speed** *when such definition is included by the user*
  - an `unknown_dimension<exp<length, 1>, exp<time, -1>>` otherwise
- *Document the fact* that **the same physical system definition** (the same set of header files) **has to be included in all translation units**
- *C++20 modules should help*
  - i.e. `units.isq.si` module includes definitions of all the SI quantities

Solution #2: Compile-time error when resulting dimension or unit is undefined

- **No support** for an `unknown_dimension<exp<length, 1>, exp<time, -1>>` at all
- **Everything will have to be predefined**
  - even if it is just some partial result of some arithmetic calculation that in the end will result in a known dimension/unit

Solution #2: Compile-time error when resulting dimension or unit is undefined

- No support for an `unknown_dimension<exp<length, 1>, exp<time, -1>>` at all
- Everything will have to be predefined
  - even if it is just some partial result of some arithmetic calculation that in the end will result in a known dimension/unit

```
Speed auto velocity = km_per_h(160);
Speed auto sink_rate = m_per_s(0.7);
auto temp = pow<2>(velocity) + pow<2>(sink_rate);  // will not compile
speed::km_per_h s1 = sqrt(temp);
speed::km_per_h s2 = sqrt(pow<2>(velocity) + pow<2>(sink_rate)); // will not compile
```

Solution #2: Compile-time error when resulting dimension or unit is undefined

- No support for an `unknown_dimension<exp<length, 1>, exp<time, -1>>` at all
- Everything will have to be predefined
  - even if it is just some partial result of some arithmetic calculation that in the end will result in a known dimension/unit

```
Speed auto velocity = km_per_h(160);
Speed auto sink_rate = m_per_s(0.7);
auto temp = pow<2>(velocity) + pow<2>(sink_rate);  // will not compile
speed::km_per_h s1 = sqrt(temp);
speed::km_per_h s2 = sqrt(pow<2>(velocity) + pow<2>(sink_rate)); // will not compile
```

May constrain the library too much.

Solution #3: Get rid of the Downcasting Facility

- The user should be responsible for providing a specific type

```
auto s1 = 120 * km / (2 * h);
si::speed<si::kilometre_per_hour> s2 = s1;
std::cout << s2 << "\n";
```

- **`s1`** always results with an **`unknown_dimension<exp<length, 1>, exp<time, -1>>`**
  - the result printed in terms of base units
- **`s2`** has a type explicitly provided by the user and implicitly converts from **`s1`**
  - output printed as expected

Solution #3: Get rid of the Downcasting Facility

- The user should be responsible for providing a specific type

```
auto s1 = 120 * km / (2 * h);
si::speed<si::kilometre_per_hour> s2 = s1;
std::cout << s2 << "\n";
```

- **`s1`** always results with an **`unknown_dimension<exp<length, 1>, exp<time, -1>>`**
  - the result printed in terms of base units
- **`s2`** has a type explicitly provided by the user and implicitly converts from **`s1`**
  - output printed as expected

Removing the Downcasting Facility simplifies the standardization effort as well.

Solution #3: Get rid of the Downcasting Facility

- How to provide a specific type for the following?

```
constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

Solution #3: Get rid of the Downcasting Facility

- How to provide a specific type for the following?

```
constexpr units::isq::Speed auto avg_speed(units::isq::Length auto d, units::isq::Time auto t)
{
  const auto s = d / t;
  std::cout << s << "\n";
  return s;
}
```

This is not only about printing the output on the console. Types are affected as well which results with:

- long compilation errors
- poor debugging experience

Poll: Choose the best solution

1. Keep it as it is
2. Compile-time error when resulting dimension or unit is undefined
3. Get rid of the Downcasting Facility

DESIGNING IS HARD

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints
- Often there is no golden bullet
  - a need to choose from several suboptimal solutions

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints
- Often there is no golden bullet
  - a need to choose from several suboptimal solutions
- C++ Standardization takes time

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints
- Often there is no golden bullet
  - a need to choose from several suboptimal solutions
- C++ Standardization takes time
- Early adopters and feedback are always welcomed

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints
- Often there is no golden bullet
  - a need to choose from several suboptimal solutions
- C++ Standardization takes time
- Early adopters and feedback are always welcomed
- Naming is hard... ;-)

Designing is Hard

- Different customers have various
  - expectations
  - experience
  - constraints
- Often there is no golden bullet
  - a need to choose from several suboptimal solutions
- C++ Standardization takes time
- Early adopters and feedback are always welcomed
- Naming is hard... ;-)

Please help: https://mpusz.github.io/units!

CAUTION
Programming
is addictive
(and too much fun)