accu 2023

BUILDING INTERFACES THAT ARE HARD TO USE INCORRECTLY

ANDREAS WEIS

Building Interfaces That Are Hard to Use Incorrectly

Andreas Weis

Woven by Toyota

ACCU 2023

About me - Andreas Weis (he/him)

- ComicSansMS
- Co-organizer of the Munich C++ User Group
- Currently working as a Runtime Engineer for Woven by Toyota

Motivation


97
Collective Wisdom
from the Experts
97 Things Every
Programmer
Should Know
O'REILLY*
Edited by Kevlin Henney

Motivation

Item #55 by Scott Meyers:
Make Interfaces Easy to Use Correctly and Hard to Use Incorrectly

Motivation

Good interfaces are:

- **Easy to use correctly.** People using a well-designed interface almost always use the interface correctly, because that's the path of least resistance. In a GUI, they almost always click on the right icon, button, or menu entry, because it's the obvious and easy thing to do. In an API, they almost always pass the correct parameters with the correct values, because that's what's most natural. With interfaces that are easy to use correctly, things just work.

Motivation

Kate Gregory - Simplicity: Not Just for Beginners

Motivation

Good interfaces are:

- **Easy to use correctly.** People using a well-designed interface almost always use the interface correctly, because that's the path of least resistance. In a GUI, they almost always click on the right icon, button, or menu entry, because it's the obvious and easy thing to do. In an API, they almost always pass the correct parameters with the correct values, because that's what's most natural. With interfaces that are easy to use correctly, things just work.
- **Hard to use incorrectly.** Good interfaces anticipate mistakes people might make and make them difficult — ideally impossible — to commit. A GUI might disable or remove commands that make no sense in the current context, for example, or an API might eliminate argument-ordering problems by allowing parameters to be passed in any order.

Overview

- Categorization of precondition violations
- Leveraging the type system to prevent precondition violations
- Techniques for restricting access to data
- Techniques for restricting control flow

What is an incorrect use?

```
std::vector<int> v;
v.resize(10);

int i = v[99];
```

Function Preconditions

- The contract provided of a function can be expressed through *preconditions* and *postconditions*.
- When calling the functions within the defined preconditions, the contract guarantees that the function will establish the postcondtions.
- When calling the function outside the preconditions, anything can happen.

$\Rightarrow$ Violation of preconditions is always an "incorrect use".

What is an incorrect use?

```
std::vector<int> v;
v.resize(10);

int i = v.at(99);
```

- Is this incorrect use?

What is an incorrect use?

```
float f = std::sqrtf(-1.f);
```

- Is this incorrect use?

Wide Contracts - Defensive Programming

- Widening a contract reduces the number of preconditions
- Thus a function with a wide contract has seemingly fewer opportunities for incorrect use
- However, the postconditions under the widened contract are usually significantly more complex
- Which again reintroduces new opportunities for incorrect use in other places
- Many preconditions are not detectable

$\Rightarrow$ Widening contracts does not prevent "incorrect use".

What is an incorrect use?

```
void compute(std::span<std::byte> buffer) {
  // zero out buffer
  std::memset(buffer.data(), buffer.size(), 0);
}
```

Precondition Violations

- Invalid argument

What is an incorrect use?

```
std::fstream fin;
char buffer[256] = {};
fin.read(buffer, 256);
```

What is an incorrect use?

```
std::mutex mtx;
mtx.lock();
mtx.lock();
```

What is an incorrect use?

```
std::vector<int> v = getNumbers();
auto it_begin = std::ranges::begin(v);
v.push_back(42);
int first_element = *it_begin();
```

What is an incorrect use?

```
void consumeObject(MyClass&& object);

MyClass my_object = createObject();
my_object.doStuff();
consumeObject(std::move(my_object));
my_object.doMoreStuff();
```

Precondition Violations

- Invalid argument
- Invalid context

How can we prevent those violations?

Type System to the rescue!

- Type system allows us to restrict the set of values assigned to an object
- Type system allows us to restrict the set of operations that can be applied to an object

Opaque Types

```
void glTexImage2D(
  GLenum target,
  GLint level,
  GLint internalformat,
  GLsizei width,
  GLsizei height,
  GLint border,
  GLenum format,
  GLenum type,
  const void * data);

glTexImage2D(1, 2, 3, 4, 5, 6, 7, 8, &data);
```

Enumerations

- `GLenum` is just a typedef for an `unsigned int`.
- Enumerator values are defined as preprocessor constants.
- Some values even overlap, defining two unrelated enumerators to the same integer value.
- Compiler will never catch an invalid enumerator argument. This is a runtime error at best.

Scoped Enumerators

```
enum class Shape {
  Circle = 1, Square = 2, Triangle = 3
};
enum class Color {
  Red = 1, Green = 2, Blue = 4
};
void setShapeColor(Shape, Color);

setShapeColor(Color::Red,
              Shape::Circle);      // does not compile!
Shape s1 = Color::Blue             // does not compile!
Shape s2 = static_cast<Shape>(3); // requires cast
```

Scoped Enumerators as Opaque Typedefs

```
enum class Handle : std::uint32_t { Invalid = 0 };
Handle createResource();

Handle h1 = createResource();
Handle h2 = Handle::Invalid;
Handle h3{ 42 };
Handle h4 = 42;          // does not compile
enum class OtherHandle : std::uint32_t {};
OtherHandle h5 = h1;     // does not compile
```

Opaque Typedefs

```
template<typename Tag_T>
struct OpaqueHandle { std::uint32_t i; };

struct FileHandleTag {};
using FileHandle = OpaqueHandle<FileHandleTag>;
struct ProcessHandleTag {};
using ProcessHandle = OpaqueHandle<ProcessHandleTag>;

FileHandle openFile(std::filesystem::path);

FileHandle h1 = openFile("important.dat");
ProcessHandle h2 = h1;     // does not compile
```

Opaque Typedefs

Types defined this way don't support common operations out-of-the-box:

- Compare for equality (`operator==`)
- Store in a `map` (`operator<`)
- Store in an `unordered_map` (`std::hash`)
- Printed to the console (`operator<<(std::ostream)`, `std::formatter`)
- Arithmetic

Libraries like Björn Fahller's `strong_type` provide customizable opaque types.
Arithmetic is supported by units libraries like Mateusz Pusz's mp-units.

Structuring arguments

```
void* memset(void* buffer, int fill_char,
             size_t buffer_size);

memset(buffer, 10, 20);
```

Compare with:

```
void* my_memset(std::span<std::byte> span, int fill_char);

my_memset(std::span(buffer, 20), 10);
```

Structuring arguments

```
struct MemsetArgs {
  void* buffer;
  size_t size;
  int fill_char;
};
void* my_memset(MemsetArgs const& args);

my_memset({ .buffer = b, .size = 20, .fill_char = 10 });
```

Structuring arguments

```
Rectangle r1{10, 20, 30, 40};

Rectangle r2{Point{ .x = 10, .y = 20 },
             Point{ .x = 30, .y = 40 }};

Rectangle r3{Point{ .x = 10, .y = 20 },
             Extents{ .width = 30, .height = 40 }};
```

Structured Data Access

```
using Shape = std::variant<Rectangle, Circle, Triangle>;

Shape s = getShape();
if (std::holds_alternative<Circle>(s)) {
  float r = std::get<Circle>(s).getRadius();
}
```

Structured Data Access

```
float getArea(Rectangle const& r);
float getArea(Circle const& c);
float getArea(Triangle const& t);

Shape my_shape = getShape();
float const area = std::visit(
  [](auto const& s) { return getArea(s); },
  my_shape);
```

Structured Data Access

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

Shape my_shape = getShape();
std::visit(overloaded{
  [](Rectangle const& r) { /* ... */ }
  [](Circle const& c) { /* ... */ }
  [](Triangle const& t) { /* ... */ }
});
```

Data Types with special properties

Consider these alternatives:

```
// \pre Widget must not be nullptr
void processWidget(Widget* w);

void processWidget(Widget& w);

void processWidget(gsl::not_null<Widget*> w);
```

Data Types with special properties

Consider these alternatives:

```
// \pre str must be a null-terminated string
void processString(char const* str);

void processString(std::string_view str);

void processString(gsl::czstring str);
```

Limited Friendship

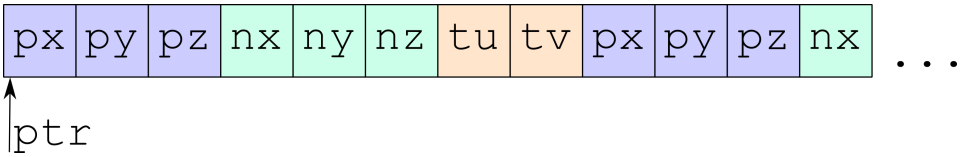
```
class Database {
  Internals m_privateState;
public:
  Data retrieveData(Handle h);
private:
  Data retrievePrivilegedData(Handle h);
};

class Client;
class PrivilegedClient;
```

Limited Friendship

```
struct AccessToken {
private:
    AccessToken() = default;
    friend class PrivilegedClient;
};

class Database {
  Internals m_privateState;
public:
  Data retrieveData(Handle h);
  Data retrievePrivilegedData(Handle h, AccesToken t);
};
```

Declarative Interface

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;
```




```
glBufferData(GL_ARRAY_BUFFER, size, ptr, ...);
glVertexAttribPointer(index_pos, 3, GL_FLOAT,
                      stride, offset_pos);
// ...
```

Declarative Interface

```
struct Vertex {
  Vec3 position;
  Vec3 normal;
  Vec2 textureCoordinate;
};
std::vector<Vertex> vertex_buffer;
```

- Geometry data starts its life as structured data in C++.
- Data gets memcopied into a single buffer that is transferred to the GPU.
- Location of the individual vertex components inside this buffer are configured through a series of `glVertexAttribPointer` calls.

Declarative Interface

```
using MyFormat = VertexFormat<
  Component::Position3f,
  Component::Normal3f,
  Component::TexCoord2f
>;

shader.bind<MyFormat, Component::Position3f>("Position");
shader.bind<MyFormat, Component::Normal3f>("Normal");
shader.bind<MyFormat, Component::TexCoord2f>("TexCoord");
```

Declarative Interface

- Specify the *what* instead of the *how*
- Restrict the parameter space to a set of predefined sensible values
- Leave just enough room for customization for what is sensible for the domain
- Deduce the tricky configuration parameters from the declarative description

Higher-level interface is potentially more restrictive than the lower-level one. Consider leaving an escape hatch for exotic use cases.

Modeling State Through Types

```
class Widget {
public:
  // Constructs an uninitialized widget.
  Widget();
  //! Initializes the widget.
  //! \pre Widget must not have been initialized.
  void init();
  //! \pre Widget must have been initialized.
  void doStuff();
};
```

Two-Phase Initialization

```
class Widget {
private:
  Widget();
  void init();
public:
  void doStuff();
};

Widget createWidget() {
  Widget w;
  w.init();
  return w;
}
```

Two-Phase Initialization

```
class UninitializedWidget {
public:
  UninitializedWidget();
};
class Widget {
private:
  Widget();
public:
  void doStuff();
};

Widget initializeWidget(UninitializedWidget&& w);
```

Error Handling Paradigms

Consider these alternatives:

```
// Returns 0 on success, error code on failure.
Result_T retrieveFrob(Frob* out_frob);

// Sets errno in case of failure
Frob retrieveFrob();

// Throws in case of failure
Frob retrieveFrob();
```

C++23 std::expected

```
std::expected<Frob, Error_T> retrieveFrob() {
  if (!frobState.isGood()) {
    return std::unexpected(Error_T::BadFrob);
  }
  return frobState;
}

std::expected<Frob, Error_T> f = retrieveFrob();
if (!f) {
  // handle error
} else {
  f->doStuff();
}
```

C++23 std::expected

```
std::expected<Frob, Error_T> retrieveFrob() {
  if (!frobState.isGood()) {
    return std::unexpected(Error_T::BadFrob);
  }
  return frobState;
}

f.value_or(defaultFrob).doStuff();
```

.

Execution Tokens

- Restrict the circumstances under which a function may be executed.
- Only allow the execution of a function in specific contexts, e.g. within a transaction, when a lock is held, when a resource is available, etc.
- Scope in which the relevant context is active is reflected by the program structure.

Execution Tokens

```
class ExecutionToken {
private:
  ExecutionToken();
};
class Provider {
public:
  ExecutionToken establishContext();
};

class Widget {
public:
  void process(ExecutionToken&& token);
};
```

Execution Tokens

- Models one-off events: Establishing the context allows us to execute one operation
- C++ move semantics allow us to retain a consumed execution token (e.g. in a member variable). Again, external tools can help prevent this.

Scoped Execution Contexts

```
vkBeginCommandBuffer(cmd_buffer, &args);
  vkCmdCopyImage(cmd_buffer, ...);
  vkCmdBeginRenderPass(cmd_buffer, ...);
    vkCmdBindVertexBuffers(cmd_buffer, ...);
    vkCmdDraw(cmd_buffer, ...);
  vkCmdEndRenderPass(cmd_buffer, ...);
vkEndCommandBuffer(cmd_buffer);
```

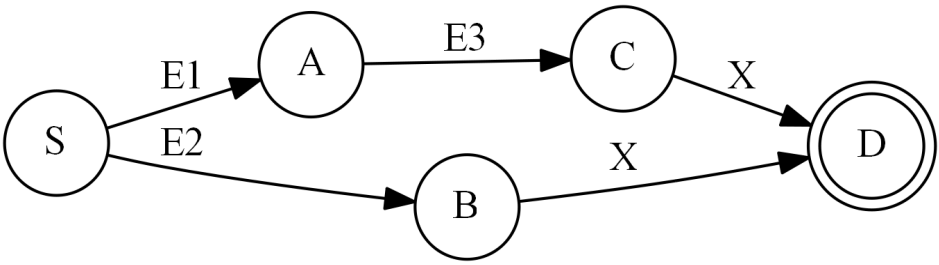
Scoped Execution Contexts

```
class CommandRecorder {
  void cmdCopyImage(...);
  RenderPassRecorder beginRenderPass(...);
};

class RenderPassRecorder {
  void cmdBindVertexBuffers(...);
  void cmdDraw(...);
};

class CommandBuffer {
  CommandRecorder begin();
};
```

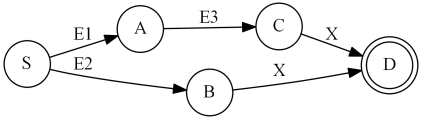
Scoped Execution Contexts

```
CommandRecorder beginCommandBuffer(CommandBuffer&&);
CommandBuffer endCommandBuffer(CommandRecorder&&);
RenderPassRecorder beginRenderPass(CommandRecorder&&);
CommandRecorder endRenderPass(RenderPassRecorder&&);

CommandBuffer cmd_buffer = ...;
auto cmd_recorder = beginCommandBuffer(std::move(cmd_buffer),
cmd_recorder.cmdCopyImage(...);
cmd_buffer = endCommandBuffer(std::move(cmd_recorder));
```

Complex State Machines



Complex State Machines




```
namespace State {
  class S;
  class A;
  // ...
}
namespace Events {
  class E1;
  // ...
}
```

Complex State Machines

```
State::S initial_state;
State::A state_a =
  transition(std::move(initial_state), Events::E1{});
State::B state_c =
  transition(std::move(state_a), Events::E3{});
```

Complex State Machines

- Gives full control over the possible state transitions and available interfaces for each state.
- Leaves behind a dead state object with each transition.
- No uniform storage for states - type erased storage again loses all of the compile-time control.

The GoF State Pattern

```
struct State {
  virtual ~State() = default;
  virtual void doStuff() = 0;
};
struct StateA : public State { /* ... */ };
struct StateB : public State { /* ... */ };

class Context {
  std::unique_ptr<State> m_state;
public:
  void doStuff() { m_state->doStuff(); }
  void setState(StateDesc new_state);
};
```

Conclusion

- Harden interfaces by making it impossible to violate preconditions without widening the contract.
- Two fundamental classes of preconditions: Restrictions on arguments and restrictions on context.
- A variety of techniques exist to move precondition checks to compile-time.
- Difficult trade-offs between correctness and implementation effort and usability.

Thanks for your attention.

 ComicSansMS