A Case-study in Rewriting a Legacy GUI Library for Real-time Audio Software in Modern C++ (Reprise)

ROTH MICHAELS

Roth Michaels
Principal Software Engineer
Native Instruments

NATIVE INSTRUMENTS

iZOTOPE

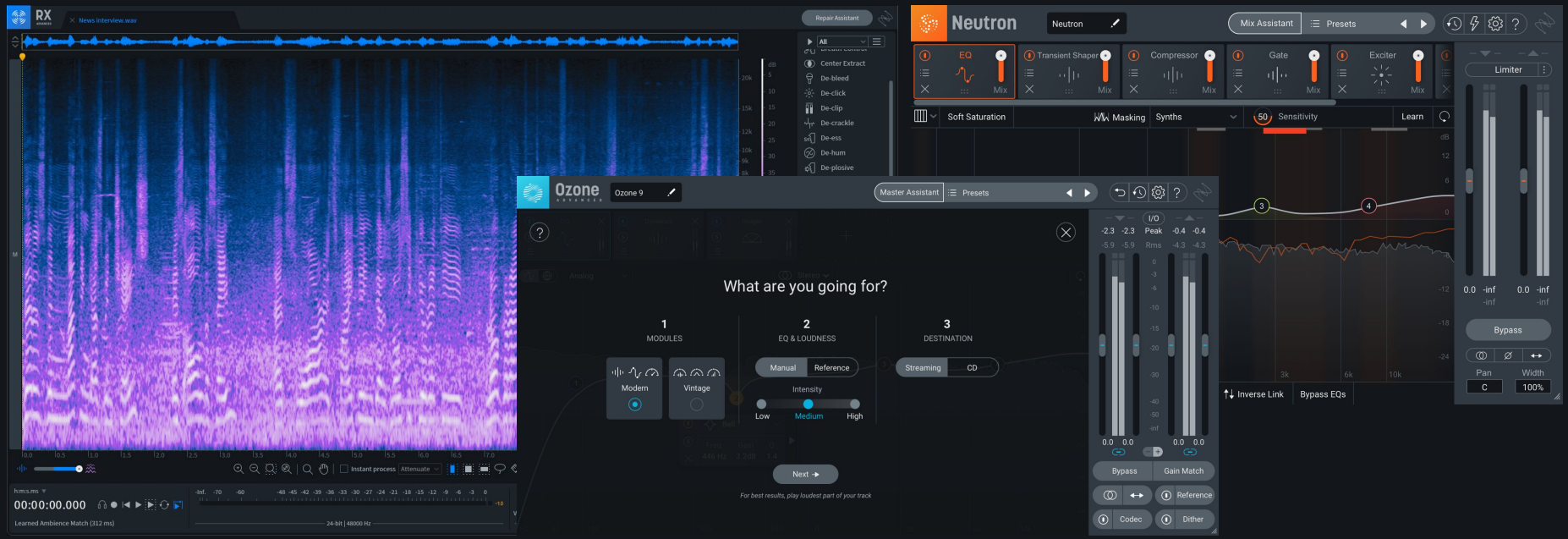
Plugin Alliance

BRAINWORX


Inst 3
Manuell
Vergleichen
Widerrufen
Wiederholen
Ansicht:
Editor
7 SKIES, Magnificence - THE DRILL
FILTER
CUTOFF
RESO
ENV AMT
THE DRILL
FLT ENV
ATTACK
DECAY
SUSTAIN
RELEASE
AMP
LAYER 1
LAYER 2
AMP ENV
Inst 4
gr808
808 Center
Mode
[mono]
808 Sides
Panning
[attack: fast]
[releaseeeeee]
Reverb: [Cloud Grain]
Size
Amount:
gr808
kick
Rev Lo-Cut:
Rev Hi-Cut:
Presets: 1 2 3
Clay and Kelsy
Bouncy FX: [off]
Speed ModW:
Kontakt
Inst 2
MS20 Drums
Cutoff
Saturation
Reverb
Delay
Delay Time
Peter Flint:
MS20 Drums
Kontakt

iZotope

real-time audio plug-ins | music, film, television, and radio



You can do it!

A Case-study in Rewriting a Legacy GUI Library for Real-time Audio Software in Modern C++

ROTH MICHAELS

@dustinmorrison6315 3 years ago

A lot of this could be also be solved with c++20 concepts.

 3  Reply

A Case-study in Rewriting a Legacy Gui Library for Real-time Audio Software in Modern C++ (Reprise)

ROTH MICHAELS

What do I mean by “property system”?

Function template overloading

- Synthesized member storage
- Synthesized getters/setters
- Change notifications
- Serialization / Deserialization

Objective-C

```
@interface Button : NSView
@property (copy, readwrite) NSString* text;
@property (copy, readwrite) NSColor* bgColor;
@end

@implementation Button
@synthesize text;
@synthesize bgColor;
@implementation
```

Swift

```
class Button : NSView {
  var text: String = "" {
    didSet {
      SetNeedsLayout();
      SetNeedsDisplay();
    }
  }
  var bgColor: UIColor = UIColor.blackColor() {
    didSet { SetNeedsDisplay(); }
  }
}
```

Where we are going...

Where we are going...

```
GLASS_PROPERTIES(ButtonProperties,
  (UIProperty, Text, ColorPropType, ""),
  (UIProperty, BGColor, ColorPropType, kBlack),
  (UIProperty, BorderColor, ColorPropType, kBlack),
  (UIProperty, BorderWidth, UIntPropertyType, 4u)
)

class Button
  : public Glass::View
  , public HasProperties<Button,
ButtonProperties::List> {
  // ...
```

Where we are going...

```
using namespace ButtonProperties;

auto b = make_shared<Button>();

b->SetProperty<Text>("Say Hello");
b->SetProperty<BGColor>(kBlue);
b->SetProperty<BorderColor>(kGold);
```

Where we are going...

```
using namespace ButtonProperties;

Button::Draw() {
  FillBox(GetBounds(), GetProperty<BGColor>());
  DrawBox(GetBounds(), GetProperty<BorderWidth(),
                       GetProperty<BorderColor>());
  DrawText(GetBounds(), kCentered,
GetProperty<Text>());
}
```


Life as a plug-in
Many challenges running in another application's process

Challenges for real-time audio plug-ins

Multiple instantiations

- Multiple instances of the same plug-in are expected within a single host process
- No global state

Many ways to get on screen

- Host creates an OS window (e.g. NSWindow)
  - Host provides view (e.g. NSView)
  - Host requests view (e.g. NSView)
- We create an OS window

Performance

- One of many running UIs
- Metering wants high framerate
- Users need responsive control of audio

Threading model

- Hosts can call us from any thread
- Each host may do this this differently

Canvas
(ca. 2002)

JUCE
(ca. 2004)

Canvas
(ca. 2002)

JUCE
(ca. 2004)

Canvas
(ca. 2002)

Qt
(ca. 2014 - ??)

Let’s do a rewrite!

XML Layout Files

```
<?xml version="1.0" standalone="yes" ?>
<iZCanvasLayout-1>
    <NewChild Name="HorizDivider" Type="Glass::Pane">
        <Property Name="BackgroundColor" Type="Optional: Glass::Color"
Value="313a42" />
        <Property Name="CornerRadius" Type="Float4Dim" Value="0" />
        <Property Name="flex-direction" Type="Enum: FlexDirection" Value="Row"
/>
        <Property Name="flex-position" Type="Enum: FlexPosition"
Value="Relative" />
        <Property Name="height" Type="Flex Float" Value="1px" />
        <Property Name="width" Type="Flex Float" Value="100%" />
    </NewChild>
    <NewChild Name="TopRow" Type="Glass::Pane">
    <!-- ... -->
```

JSON StyleSheets

```
{
      "Classes": [
            {
                  "ClassName": "FilterControlCurve",
                  "Properties": [
                        {
                              "PropertyName": "Point Selected Border",
                              "PropertyType": "Image",
                              "Value":
                             "png/ControlCurve_Point_Selected_Border.png"
                        }
            }
    ]
}
```

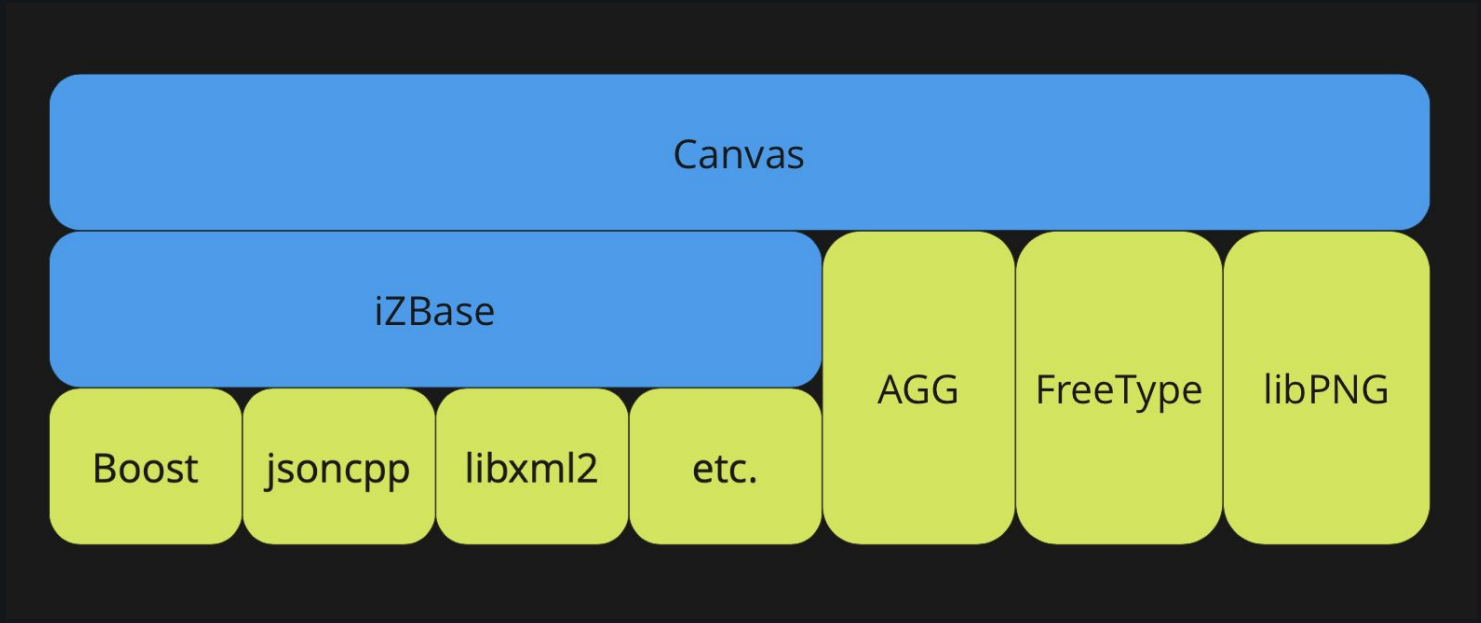

Neoverb
Neoverb
Reverb Assistant
CPPCon
Mod
Rate
2.07 Hz
Depth
51 %
-34.1 dB
Pre-Delay
Smooth
-37.2 dB
Dry/Wet
Level
Live
[NONE]
255
000000.00
Arrow
0000FF
FFFFFF
0.000000, 12.0000
10.000000
false
invalid
566068
Pre EQ
Reverb EQ
100 %
Freq 340 Hz
Gain 0.0 dB
Q 0.7

Which string?

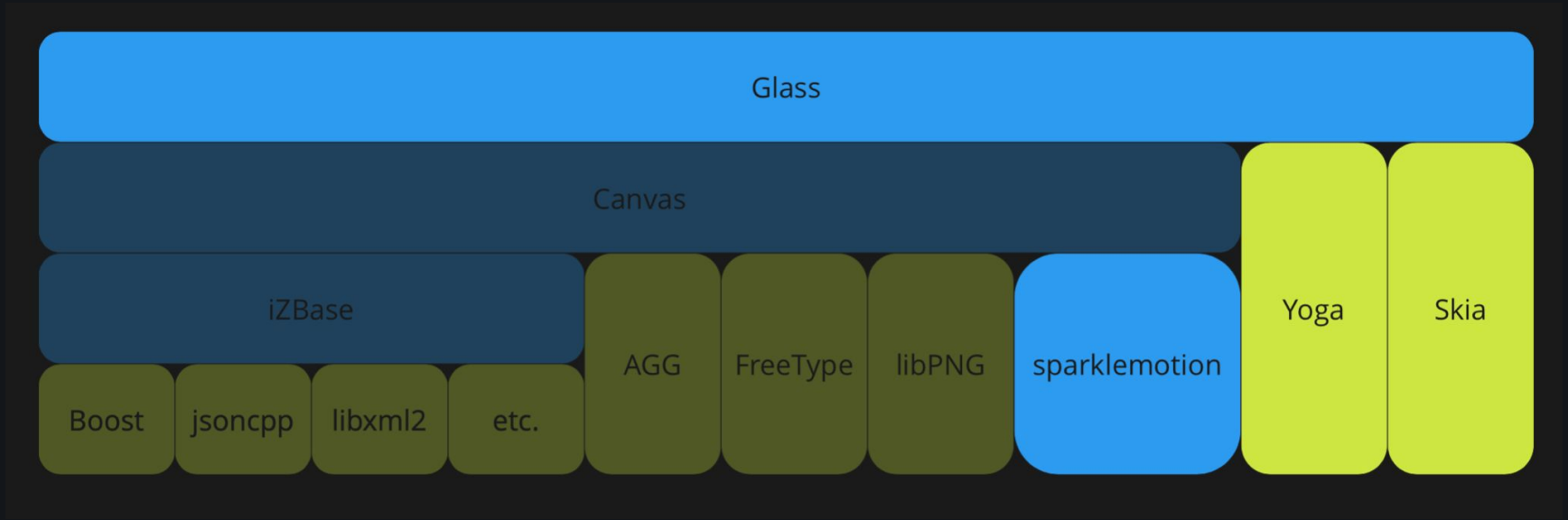
- `String`
- `QString`
- `juce::string`
- `std::string`

Glass: It's what windows are made out of !

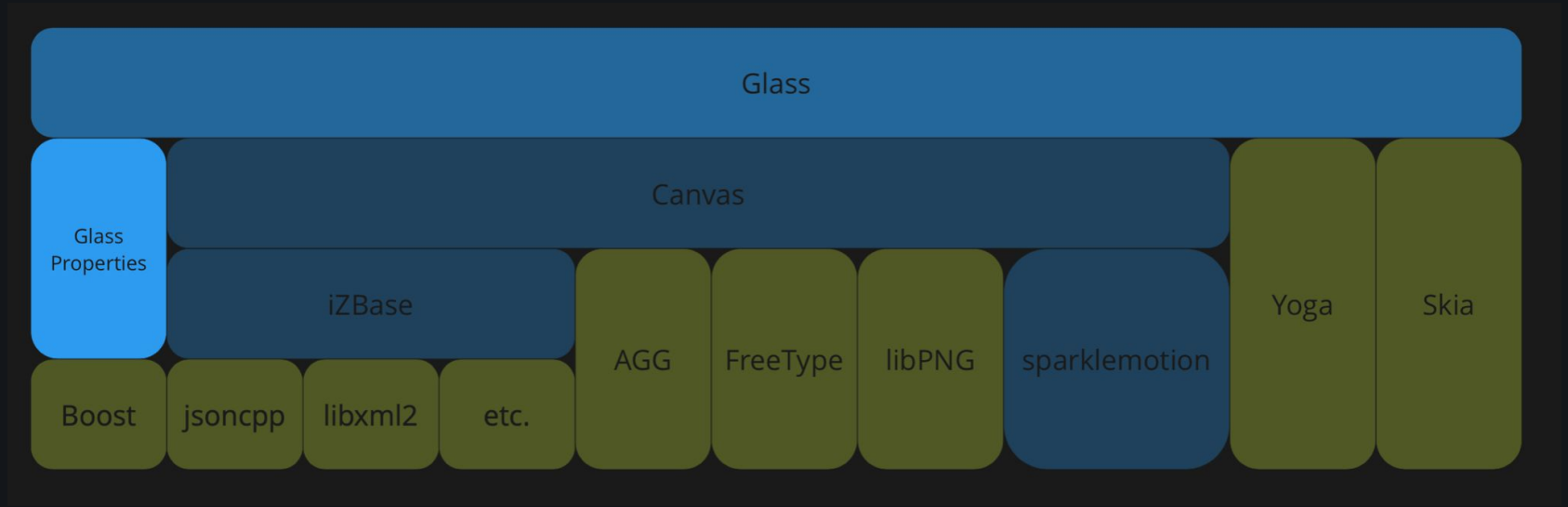
Porcelain rewrite



Porcelain rewrite



Porcelain rewrite



You are finished if you wait to finish

You are finished if you wait to finish

Insight

You are finished if you wait to finish

Insight

Vocal Doubler

You are finished if you wait to finish

Insight

Vocal Doubler

Nectar

You are finished if you wait to finish

Insight

Vocal Doubler

Nectar

Neutron

You are finished if you wait to finish


Insight


Vocal Doubler


Nectar


Neutron


Ozone

You are finished if you wait to finish


Insight


Vocal Doubler


Nectar


Neutron


Ozone


Tonal Balance Control

You are finished if you wait to finish

Insight

Vocal Doubler

Nectar

Neutron

Ozone

Tonal Balance Control

Dialogue Match

You are finished if you wait to finish

Insight

Vocal Doubler

Nectar

Neutron

Ozone

Tonal Balance Control

Dialogue Match

RX

You are finished if you wait to finish


Insight


Vocal Doubler


Nectar


Neutron


Ozone


Tonal Balance Control


Dialogue Match


RX


Neoverb

What’s wrong with Canvas properties?

Imperative and not type-safe...

```
// View.h
namespace Canvas {
class View : public PropertyHolder, public Trackable {
… }
}
// Button.h
class Button : public Canvas::View {
public:
  static const char* const kText;
  static const char* const kBGColor;
  static const char* const kBorderColor;
  static const char* const kBorderWidth;
  // ...
};
```

```
// View.h
namespace Canvas {
class View : public PropertyHolder, public Trackable {
… }
}
// Button.h
class Button : public Canvas::View {
public:
  static const char* const kText;
  static const char* const kBGColor;
  static const char* const kBorderColor;
  static const char* const kBorderWidth;
  // ...
};
```

```
// View.h
namespace Canvas {
class View : public PropertyHolder, public Trackable {
… }
}
// Button.h
class Button : public Canvas::View {
public:
  static const char* const kText;
  static const char* const kBGColor;
  static const char* const kBorderColor;
  static const char* const kBorderWidth;
  // ...
};
```

```
class PropertyHolder {
public:
  bool CreateProperty(string_view name,
                      string_view typeName,
                      any value);

  template <typename T>
  optional<T> GetProperty(string_view name);

  template <typename T>
  bool SetProperty(string_view name, T value);
private:
  unordered_map<string, any> m_properties;
};
```

```
template <typename T>
optional <T> GetProperty(string_view name) {
   const auto& it = std::find_if(
     m_propertyValues.cbegin(), m_propertyValues.cend(),
     [&](const auto& e) {
              return std::string_view{e.first} == name;
   });
   if (it == m_propertyValues.cend()) { return {}; }

   auto* value = boost::any_cast<T>(&it->second.value);
   if (!value) { return {}; }

   return *value;
}
```

```
template <typename T>
optional <T> GetProperty(string_view name) {
   const auto& it = std::find_if(
     m_propertyValues.cbegin(),
m_propertyValues.cend(),
     [&](const auto& e) {
              return std::string_view{e.first} == name;
   });
   if (it == m_propertyValues.cend()) { return {}; }

   auto* value =
boost::any_cast<T>(&it->second.value);
   if (!value) { return {}; }

   return *value;
}
```

```
template <typename T>
optional <T> GetProperty(string_view name) {
   const auto& it = std::find_if(
     m_propertyValues.cbegin(),
m_propertyValues.cend(),
     [&](const auto& e) {
              return std::string_view{e.first} == name;
   });
   if (it == m_propertyValues.cend()) { return {}; }

   auto* value =
boost::any_cast<T>(&it->second.value);
   if (!value) { return {}; }

   return *value;
}
```

```
template <typename T>
optional <T> GetProperty(string_view name) {
   const auto& it = std::find_if(
     m_propertyValues.cbegin(),
m_propertyValues.cend(),
     [&](const auto& e) {
              return std::string_view{e.first} == name;
   });
   if (it == m_propertyValues.cend()) { return {}; }

   auto* value =
boost::any_cast<T>(&it->second.value);
   if (!value) { return {}; }

   return *value;
}
```

```
template <typename T>
optional <T> GetProperty(string_view name) {
   const auto& it = std::find_if(
     m_propertyValues.cbegin(),
m_propertyValues.cend(),
     [&](const auto& e) {
              return std::string_view{e.first} == name;
   });
   if (it == m_propertyValues.cend()) { return {}; }

   auto* value =
boost::any_cast<T>(&it->second.value);
   if (!value) { return {}; }

   return *value;
}
```

```
class PropertyHolder {
public:
  // ...

  Signal<> GetPropSignal(string_view name);

  // ...
};
```

```
class PropertyHolder {
public:
  // ...

  template <typename T>
  optional<T> GetSerializedValue(string_view name);

  template <typename T>
  bool SetSerializedValue(string_view name, T value);

  // huge API surface continued ...
};
```

```
// Button.h
class Button : public Canvas::View {
  static const char* const kText;
  static const char* const kBGColor;
  static const char* const kBorderColor;
  static const char* const kBorderWidth;
  // ...
};
```

```
// Button.cpp
const char* const Button::kText = "Text";
const char* const Button::kBGColor = "BG Color"
const char* const Button::kBorderColor = "Border Color";
const char* const Button::kBorderWidth = "Border Width";

Button::Button() {
  CreateProperty<string>(kText, kStringProp, "");
  CreateProperty<Color>(kBGColor, kColorProp, kBlack);
  CreateProperty<Color>(kBorderColor, kColorProp, kBlue);
  CreateProperty<unsigned>(kBorderWidth, kUIntProp, 4);
// ...
```

```
// Button.cpp
Button::Button() {
  GetPropSignal<string>(kText).Connect(this, [] {
    optional<string> text = GetProperty<string>(kText);
    assert(text);
    SetNeedsLayout();
    m_elipsize = Elipsize(text.value_or(""));
    SetNeedsDisplay();
  };
  auto update = [this] { SetNeedsDisplay(); }
  GetPropSignal<Color>(kBGColor).Connect(this, update);

GetPropSignal<Color>(kBorderColor).Connect(this,update);
```

```
// Button.cpp
Button::Button() {
  GetPropSignal<string>(kText).Connect(this, [] {
    optional<string> text = GetProperty<string>(kText);
    assert(text);
    SetNeedsLayout();
    m_elipsize = Elipsize(text.value_or(""));
    SetNeedsDisplay();
  };
  auto update = [this] { SetNeedsDisplay(); }
  GetPropSignal<Color>(kBGColor).Connect(this, update);

GetPropSignal<Color>(kBorderColor).Connect(this,update);
```

```
// Button.cpp
Button::Button() {
  GetPropSignal<string>(kText).Connect(this, [] {
    checked_value<string> text =
GetProperty<string>(kText);
    m_elipsize = Elipsize(text);
    SetNeedsLayout();
    SetNeedsDisplay();
  };
  auto update = [this] { SetNeedsDisplay(); }
  GetPropSignal<Color>(kBGColor).Connect(this, update);

GetPropSignal<Color>(kBorderColor).Connect(this,update);
```

Canvas property types

- `int`
- `unsigned`
- `float`
- `bool`
- `Point`
- `Size`
- `Rect`
- `Color`
- `String`
- `Image`
- `FloatRange`
- `UIntRange`
- `Cursor`

Canvas Properties

Simplify!

User problems

- Imperative
- Run-time type mismatches
- Run-time declarations
- Confusing signal connections
- Forgetting to repaint
- Doesn’t encourage using new types

Solutions

- Declarative API
- Put all information together
- Typesafe, compile time errors
- Only create properties on construction

M4? C macros?

C++17 templates!

~~C++17 templates!~~
C++20 Concepts!

C++20 Concepts

an introduction...

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

C++20 Concepts

```
template <C1 T>
requires C2<T>
C3 auto Foo(C4 auto x) requires C5<decltype(Bar(x))>
```

Using Concepts to create overload sets...

...and constrain types used in templates

```
template <typename T> constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```
template <typename T> constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```
template <typename T> constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
constexpr float Half(float x) {
    return x / 2.f;
}
constexpr double Half(double x) {
    return x / 2.0;
}
constexpr long double Half(long double x) {
    return x / 2.0;
}
```

```
template <typename T> constexpr auto Half(T x) {
    if constexpr (std::is_integral_v<T>) {
        return x / T(2) + (x % T(2));
    } else {
        return x / T(2);
    }
}
```

```
constexpr auto Half(std::integral auto x) {
    return x / T(2) + (x % T(2));
}

constexpr auto Half(std::floating_point auto x) {
    return x / T(2);
}
```

```
template <typename T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T>
requires std::integral<T>
constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
template <CanHalf T> constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <typename T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T>
requires std::integral<T>
constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
template <CanHalf T> constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <typename T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T>
requires std::integral<T>
constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
template <CanHalf T> constexpr auto Half(T x) {
    return x / T(2);
}
```

```
template <typename T>
concept CanHalf = requires(T t) {
    { t / T(2) } -> std::same_as<T>;
};
template <CanHalf T>
requires std::integral<T>
constexpr auto Half(T x) {
    return x / T(2) + (x % T(2));
}
template <CanHalf T> constexpr auto Half(T x) {
    return x / T(2);
}
```

~~C++17 templates!~~
C++20 Concepts!

```
// Button.cpp
Button::Button() {
  GetPropSignal<string>(kText).Connect(this, [] {
    checked_value<string> text =
GetProperty<string>(kText);
    m_elipsize = Elipsize(text);
    SetNeedsLayout();
    SetNeedsDisplay();
  };
```


IT CAN ONLY BE ATTRIBUTABLE
TO
HUMAN ERROR
memegenerator.net

```
// Button.cpp
const char* const Button::kText = "Text";
const char* const Button::kBGColor = "BG Color"
const char* const Button::kBorderColor = "Border Color";
const char* const Button::kBorderWidth = "Border Width";

Button::Button() {
  CreateProperty<string>(kText, kStringProp, "");
  CreateProperty<Color>(kBGColor, kColorProp, kBlack);
  CreateProperty<Color>(kBorderColor, kColorProp, kBlue);
  CreateProperty<unsigned>(kBorderWidth, kUIntProp, 4);
// ...
```

It can be safe!


```
optional<T>
Deserialize<T>(string_view name,
```

```
string Serialize<T>(string_view
name, T value)
```

```
SetProperty<T>(string_view name, T)
```

```
GetProperty<T>(string_view name)
```

```
PropertyHolder
```

JSON StyleSheet Errors

```
{
      "Classes": [
            {
                  "ClassName": "FilterControlCurve",
                  "Properties": [
                        {
                              "PropertyName": "Point Selected Border",
                              "PropertyType": "Image",
                              "Value":
                           "png/ControlCurve_Point_Selected_Border.png"
                        }
                  ]
            }
    ]
}
```

JSON StyleSheet Errors

```
{
      "Classes": [
            {
                  "ClassName": "FilterControlCurve",
                  "Properties": [
                        {
                              "PropertyName": "Point Selected Border",
                              "PropertyType": "nonsense",
                              "Value":
                           "png/ControlCurve_Point_Selected_Border.png"
                        }
                  ]
            }
    ]
}
```

JSON StyleSheet Errors

```
{
      "Classes": [
            {
                  "ClassName": "FilterControlCurve",
                  "Properties": [
                        {
                              "PropertyName": "Point Selected Border",
                              "PropertyType": "Image",
                              "Value": ""

                        }
                ]
            }
    ]
}
```

It can be safe!


```
CreateProperty<T>(string_view name)
```

```
SetProperty<T>(string_view name, T)
```

```
GetProperty<T>(string_view name)
```

PropertyHolder

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
  using List = PropertyList<Text>;
}
class Button
  : public HasProperties<Button,
ButtonProperties::List> {

  didSet(ButtonProperties::Text);
}
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
  using List = PropertyList<Text>;
}
class Button
  : public HasProperties<Button,
ButtonProperties::List> {

  didSet(ButtonProperties::Text);
}
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;
   { T::defaultValue } ->
      std::convertible_to<typename T::property_type>;
};
```

```
namespace ButtonProperties {
  struct Text {



  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;



};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;


  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;



};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;


  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;


};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";

  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;


};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";

  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;
   { T::defaultValue } ->
      std::convertible_to<typename T::property_type>;
};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = std::string;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;
   { T::defaultValue } ->
      std::convertible_to<typename T::property_type>;
};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = StringPropertyType;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
}
template <class T>
concept PropertyDescription = requires(T t) {
   typename T::property_type;
   { T::name } -> std::convertible_to<std::string>;
   { T::defaultValue } ->
      std::convertible_to<typename T::property_type>;
};
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = StringPropertyType;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
}
template <class T>
concept PropertyTypeDescription = true;

template <class T>
concept PropertyDescription =
    PropertyTypeDescription<typename T::property_type> &&
    requires(T t) {
        { T::name } -> std::convertible_to<std::string>;
        { T::defaultValue }
           -> std::convertible_to<typename T::property_type>;
};
```

```
struct StringPropertyType {
   using type = std::string;
   static constexpr auto name = "std::string";
   static std::string serialize(std::string value);
   static optional<std::string> deserialize(
      const std::string& serializedValue);
};
template <class T>
concept PropertyTypeDescription = requires {
   typename T::type;
   { T::serialize(std::declval<typename T::type>()) } ->
      std::convertible_to<std::string>;
   { T::deserialize(std::string{}) } ->
      std::same_as<std::optional<typename T::type>>;
};
```

```
struct UIntPropertyType {
  using type = uint32_t;
  static constexpr auto name = "UInt";
  static std::string serialize(uint32_t value) {
    return format("{}", value);
  }
  static optional<std::string>
  deserialize(const std::string& serializedValue) {
    try {
      return
         boost::lexical_cast<uint32_t>(serializedValue);
    } catch (boost::bad_lexical_cast&) {
      return nullopt;
    }
  }
}
```

```
template <typename T>
struct OptionalProperty {
  using type = optional<T>;
  static auto name() {
    return std::format("Optional: {}", T::name);
  }
  static std::string serialize(const type& value) {
    if (!value) {
      return "nullopt"
    }
    return T::serialize(*value);
  }
// ...
```

```
template <typename T>
struct OptionalProperty {
  using type = optional<T>;
  static auto name() {
    return std::format("Optional: {}", T::name);
  }
  static std::string serialize(const type& value) {
    if (!value) {
      return "nullopt"
    }
    return T::serialize(*value);
  }
// ...
```

```
template <typename T>
struct OptionalProperty {
  using type = optional<T>;
  static auto name() {
    return std::format("Optional: {}", getName<T>());
  }
  static std::string serialize(const type& value) {
    if (!value) {
      return "nullopt"
    }
    return T::serialize(*value);
  }
// ...
```

```
template <class P>
concept HasNameValue = requires {
   { P::name } -> std::convertible_to<std::string>;
};

template <class P>
concept HasNameFunction = requires {
   { P::name() } -> std::convertible_to<std::string>;
};

template <class P>
concept HasName = HasNameValue<P> || HasNameFunction<P>;
```

```
template <HasNameValue T>
constexpr auto getName() {
    return T::name;
}

template <HasNameFunction T>
constexpr auto getName() {
    return T::name();
}
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = StringPropertyType;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
  struct BGColor {
    using property_type = ColorPropertyType;
    static constexpr auto name = "Background Color";
    static constexpr auto defaultValue = kBlack;
  };
  // ...
```

```
namespace ButtonProperties {
  // ...
  struct BorderColor {
    using property_type = ColorPropertyType;
    static constexpr auto name = "Border Color";
    static constexpr auto defaultValue = kBlue;
  };
  struct BorderWidth {
    using property_type = UIntPropertyType;
    static constexpr auto name = "Border Width";
    static constexpr property_type::type
defaultValue{4u};
  };
  // ...
```

```
namespace ButtonProperties {



  // ...
  using List = PropertyList<Text,
                            BGColor,
                            BorderColor,
                            BorderWidth>;
  //
}
```

```
template <typename... Ps>
struct PropertyList {};
```

```
namespace ButtonProperties {
  // ...
  using List = PropertyList<Text,
                            BGColor,
                            BorderColor,
                            BorderWidth>;
}
class Button
  : public Glass::View,
  , public HasProperties<Button, ButtonProperties::List>
{

  didSet(ButtonProperties::Text);
}
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
public:
  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  typename T::property_type::type GetProperty();

  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  void SetProperty(typename T::property_type::type);

private:
  PropertyHolder m_propertyHolder{};
  Trackable m_trackable{};
};
```

```
namespace detail {
   template <PropertyDescription... Ts>
   void PropertyListTest(PropertyList<Ts...>);
}

template <class T>
concept PropertyListConcept = requires(T t) {
   detail::PropertyListTest(t);
};
```

```
namespace detail {
   template <PropertyDescription T,
             PropertyDescription... LTs>
   constexpr bool is_type_in_list(PropertyList<LTs...>)
{
      return (std::same_as<T, LTs> || ...);
   }
}

template <PropertyListConcept L, PropertyDescription T>
constexpr inline bool PropertyListHasType =
 detail::is_type_in_list<T>(L{});
```

At the call-site...

```
using namespace ButtonProperties;

auto b = make_shared<Button>();

b->SetProperty<Text>("Say Hello");
b->SetProperty<BGColor>(kBlue);
b->SetProperty<BorderColor>(kGold);
```

```
class HasProperties<Button, ButtonProperties::List> {
public:
  template <>
  string GetProperty<Text>();

  template <>
  void SetProperty<Text>(string value);

  template <>
  Color GetProperty<BGColor>();

  template <>
  void SetProperty<BGColor>(Color value);

};
```

```
template <typename Derived, typename Ps>
class HasProperties {
public:
  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  typename P::property_type::type GetProperty() {
    using type = P::property_type::type;
    auto v = m_propertyHolder.template
                GetProperty<type>(getName<P>());
    assert(v);
    return *v;
  }
  // ...
};
```

```
template <typename Derived, typename Ps>
class HasProperties {
public:
  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  void SetProperty(
             typename T::property_type::type value) {
    using type = typename T::property_type::type;
    auto success =
      m_propertyHolder.template GetProperty<type>(
        getName<T>(), std::move(value));
    assert(success);
  }
  // ...
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
public:
  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  typename T::property_type::type GetProperty();

  template <PropertyDescription P>
           requires PropertyListHasType<Ps, P>
  void SetProperty(typename T::property_type::type);

private:
  PropertyHolder m_propertyHolder{};
  Trackable m_trackable{};
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
protected:
  HasProperties() {
    createProperties(Ps{});
  }
private:
  template <PropertyDescription... P>
  void createProperties(PropertyList<P...>);

  // ...
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
private:
  template <PropertyDescription... P>
  void createProperties(PropertyList<P...>) {
    (createProperty<P>(), ...);
  }

  template <PropertyDescription P>
  void createProperty();
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
private:
  template <typename P>
  void createProperty() {
    auto success = m_propertyHolder.CreateProperty(
                     getName<P>(),
                     getName<typename
                               P::property_type>(),
                     T::defaultValue);
    assert(success);
  }
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
private:
  template <typename P>
  void createProperty() {
    auto success = m_propertyHolder.CreateProperty(
                     getName<P>(),
                     getName<typename
                               P::property_type>(),
                     T::defaultValue);
    assert(success);
  }
};
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
private:
  template <typename P>
  void createProperty() {
    auto success = m_propertyHolder.CreateProperty(
                     getName<P>(),
                     getName<typename
                               P::property_type>(),
                     getDefaultValue<T>());
    assert(success);
    connectDidSet<P>();
  }
};
```

```
template <class P, type T>
concept HasDefaultValueValue = requires {
    { P::defaultValue } ->
        std::convertible_to<typename P::property_type::type>;
};

template <class P>
concept HasDefaultValueFunction = requires {
    { P::defaultValue() } ->
        std::convertible_to<typename P::property_type::type>;
};

template <class P>
concept HasDefaultValue = HasDefaultValuealue<P> ||
    HasDefaultValueFunction<P>;
```

```
template <HasDefaultValueValue T>
constexpr auto getDefaultValue() {
    return T::defaultValue;
}

template <HasDefaultValueFunction T>
constexpr auto getDefaultValue() {
    return T::defaultValue();
}
```

```
template <typename Derived, PropertyListConcept Ps>
class HasProperties {
private:
  template <typename P>
  void connectDidSet() {

m_propertyHolder.GetPropSignal(getName<P>()).Connect(
      &m_trackable, [this_ =
static_cast<Derived*>(this)] {
        this_->didSet(P{});
        this_->SetNeedsLayout();
        this_->SetNeedsDisplay();
    });
  }
};
```

```
struct LayoutProperty {};

struct DisplayProperty {};

struct UIProperty : LayoutProperty, DisplayProperty {};
```

```
template <typename P>
void connectDidSet() {
  m_propertyHolder.GetPropSignal(getName<P>()).Connect(
    &m_trackable, [this_ = static_cast<Derived*>(this)] {
      if constexpr (HasDidSet<Derived,P>) {
        this_->didSet(P{});
      }
      if constexpr (HasSetNeedsDisplay<U> &&
                    std::derived_from<P, DisplayProperty>) {
        this_->SetNeedsDisplay();
      }
      if constexpr (HasSetNeedsLayout<U> &&
                    std::derived_from<P, LayoutProperty>) {
        this_->SetNeedsLayout();
      }
  });
}
```

```
namespace ButtonProperties {
  struct Text : UIProperty {
    using property_type = StringPropertyType;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
  struct BGColor : DisplayProperty {
    using property_type = ColorPropertyType;
    static constexpr auto name = "Background Color";
    static constexpr auto defaultValue = kBlack;
  };
  // ...
```

```
namespace ButtonProperties {
  // ...
  struct BorderColor : DisplayProperty {
    using property_type = ColorPropertyType;
    static constexpr auto name = "Border Color";
    static constexpr auto defaultValue = kBlue;
  };
  struct BorderWidth : DisplayProperty {
    using property_type = UIntPropertyType;
    static constexpr auto name = "Border Width";
    static constexpr property_type::type
                                defaultValue{4u};
  };
  // ...
```

```
GLASS_PROPERTIES(ButtonProperties,
  (UIProperty, Text, ColorPropType, ""),
  (DisplayProperty, BGColor, ColorPropType, kBlack),
  (DisplayProperty, BorderColor, ColorPropType, kBlack),
  (DisplayProperty, BorderWidth, UIntPropertyType, 4u)
)

class Button
  : public Glass::View
  , public HasProperties<Button, ButtonProperties::List>
{

  didSet(Text);
}
```

What’s next?

...meta-reprise...

Thank you!

Roth Michaels
Principal Software Engineer
roth.michaels@native-instruments.com
@thevibesman

```
template <class P>
concept HasNameValue = requires {
   { P::name } -> std::convertible_to<std::string>;
};

template <class P>
concept HasNameFunction = requires {
   { P::name() } -> std::convertible_to<std::string>;
};

template <class P>
concept HasName = HasNameValue<P> || HasNameFunction<P>;
```

```
template <class P, type T>
concept HasDefaultValueValue = requires {
    { P::defaultValue } ->
        std::convertible_to<typename P::property_type::type>;
};

template <class P>
concept HasDefaultValueFunction = requires {
    { P::defaultValue() } ->
        std::convertible_to<typename P::property_type::type>;
};

template <class P>
concept HasDefaultValue = HasDefaultValuealue<P> ||
    HasDefaultValueFunction<P>;
```

```
namespace ButtonProperties {
  struct Text {
    using property_type = StringPropertyType;
    static constexpr auto name = "Text";
    static constexpr auto defaultValue = "";
  };
}

template <class T>
concept PropertyDescription =
    PropertyTypeDescription<typename T::property_type> &&
    HasName<T> &&
    HasDefaultValue<T, typename T::property_type::type>;
```

```
struct StringPropertyType {
   using type = std::string;
   static constexpr auto name = "std::string";
   static std::string serialize(std::string value);
   static optional<std::string> deserialize(
      const std::string& serializedValue);
};
template <class T>
concept PropertyTypeDescription = requires {
   typename T::type;
   { T::serialize(std::declval<typename T::type>()) } ->
      std::convertible_to<std::string>;
   { T::deserialize(std::string{}) } ->
      std::same_as<std::optional<typename T::type>>;
};
```

Appendix