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

ROTH MICHAELS

Roth Michaels

Principal Software Engineer, Architect Music Production Software

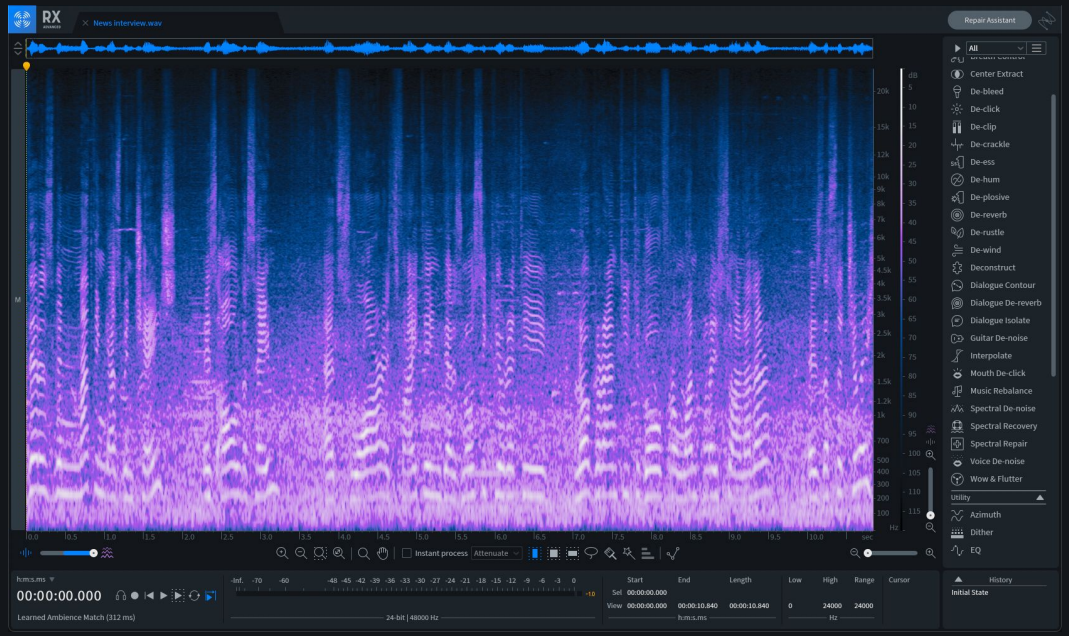
iZotope Inc.

iZotope

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

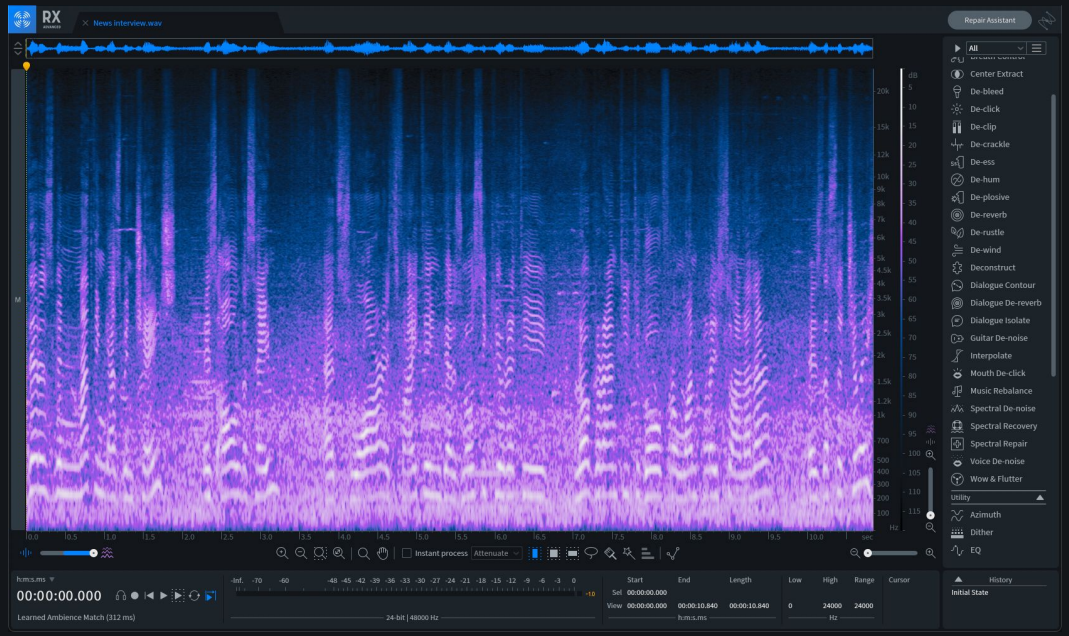
iZotope

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



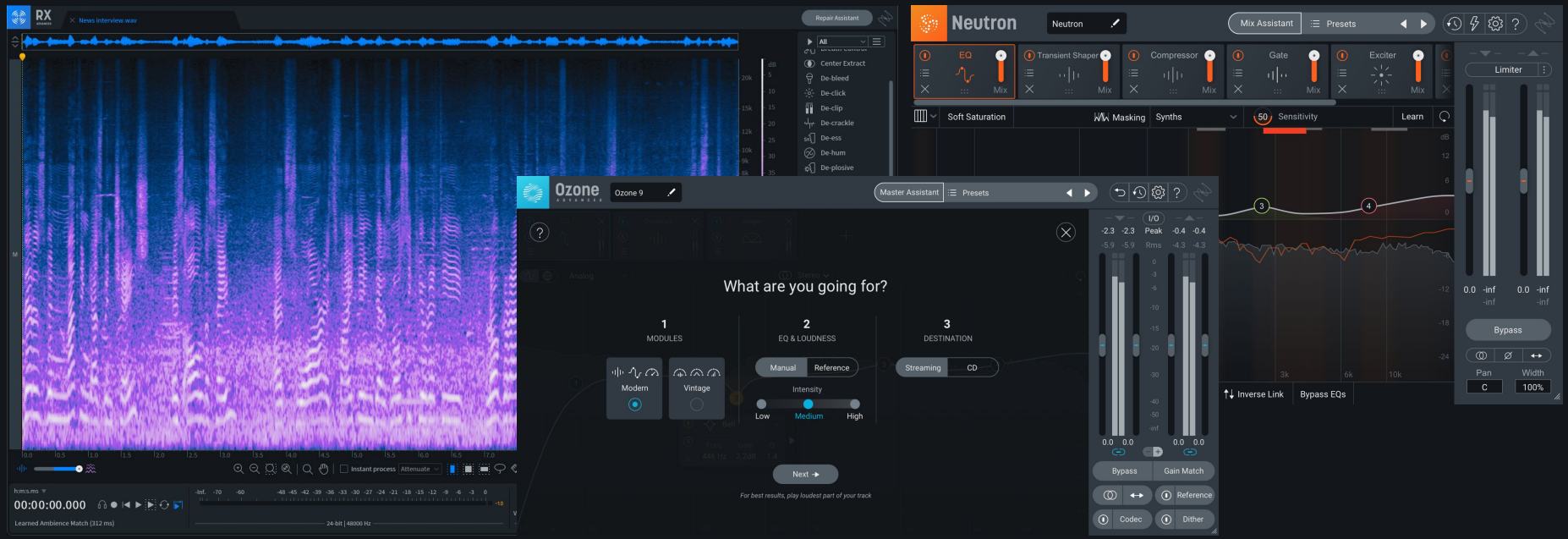
iZotope

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



iZotope

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



Glass Properties

Making a legacy run-time system compile-time safe

What do I mean by property system?

What are properties?

- 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...

```
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>
{
  // ...
```

```
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>());
}
```

Not invented here?

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)

iZotope Inc.

iZotope Inc.

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"
                        }
            }
    ]
}
```

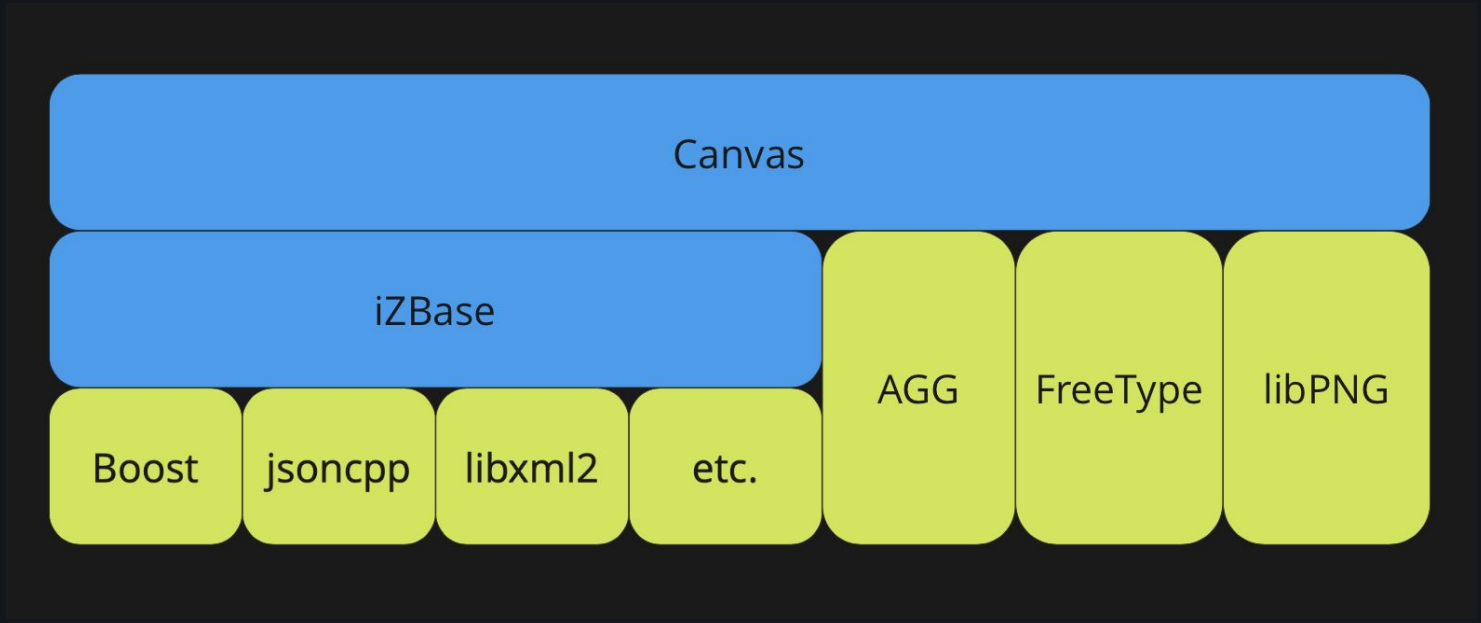

Neoverb
Neoverb
Reverb Assistant
CPPCon
Mod
Rate
2.07 Hz
Depth
51 %
-34.1 dB
Pre-Delay
Smooth
-37.2 dB
Dry/Wet
Level
Live
[NONE]
255
000000.00
Arrow
0000FF
FFFFFF
0.000000, 12.0000
10.000000
false
invalid
566068
Pre EQ
Reverb EQ
100 %
Freq 340 Hz
Gain 0.0 dB
Q 0.7

Which string?

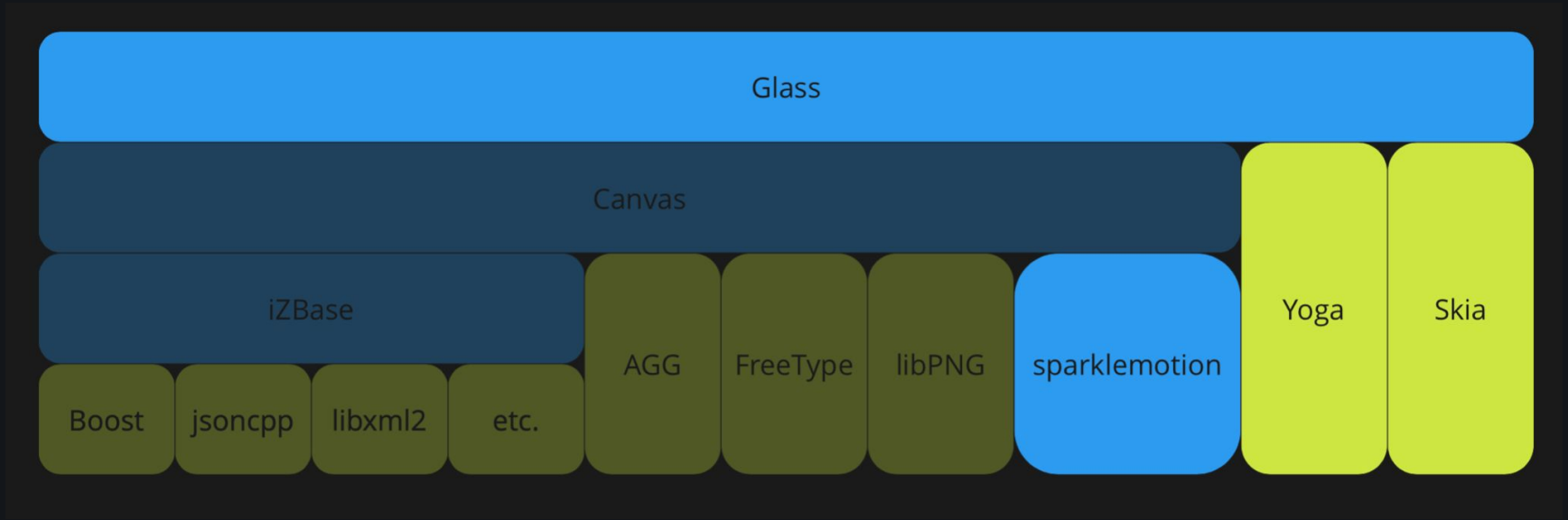
- `String`
- `QString`
- `juce::string`
- `std::string`

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

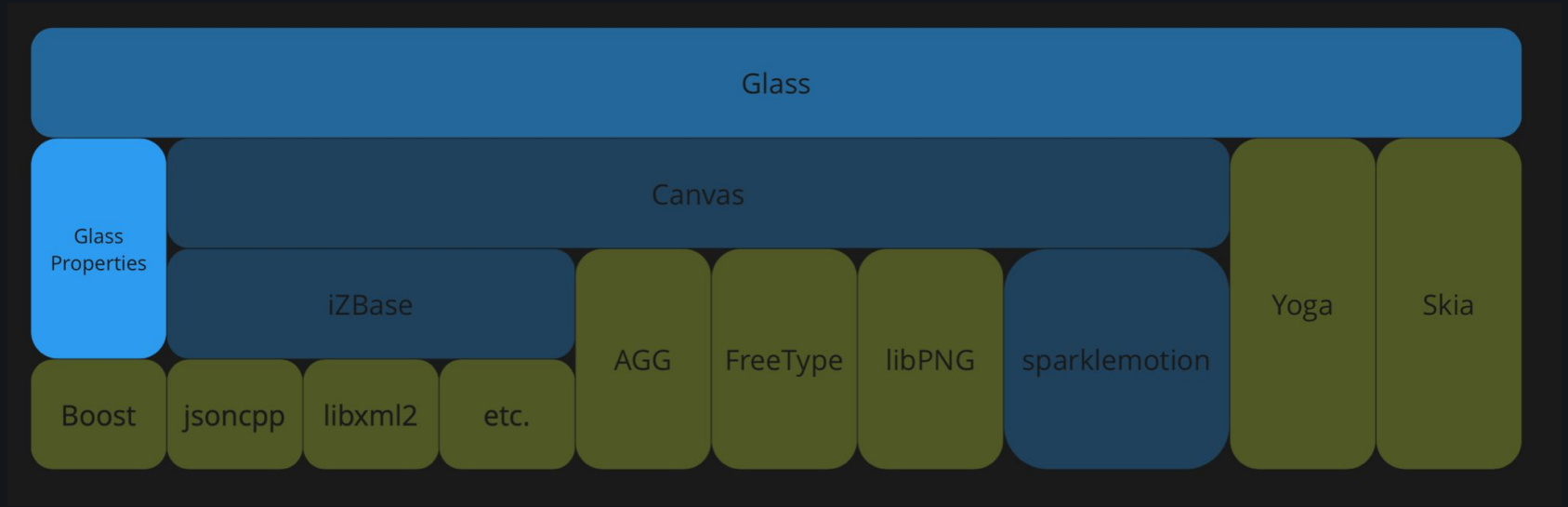
Porcelain rewrite



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 is wrong with Canvas properties?

```
// 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;
}
```

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

  Signal<> GetPropSignal(string_view name);

  // ...
};
```

```
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:
  // ...

  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, [] {
    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);
```

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

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


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 StyleSheets

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

JSON StyleSheets Errors

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

JSON StyleSheets Errors

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

JSON StyleSheets 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 = "";
  };
  using List = PropertyList<Text>;
}
class Button
  : public HasProperties<Button, ButtonProperties::List> {

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

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

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

```
struct StringPropertyType {
  using type = std::string;
  static constexpr auto name = "std::string";
  static std::string serialize(std::string value) {
    return value;
  }
  static optional<std::string>
  deserialize(const std::string& serializedValue) {
    return serializedValue;
  }
}
```

```
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 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 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 format("Optional: {}", getName<T>());
  }
  static std::string serialize(const type& value) {
    if (!value) {
      return "nullopt"
    }
    return T::serialize(*value);
  }
// ...
```

```
template <typename T>
constexpr auto getName() {
  if constexpr (is_function_v<T::name>) {
    return T::name();
  } else {
    return T::name;
  }
}
```

```
template <typename T>
constexpr auto
getName(enable_if_t<is_function_v<T::name>, T*> = nullptr)
{
  return T::name();
}

template <typename T>
constexpr auto
getName(enable_if_t<!is_function_v<T::name>, T*> = nullptr)
{
  return T::name;
}
```

```
template <typename T>
constexpr auto getName() {
  if constexpr (is_function_v<T::name>) {
    return T::name();
  } else {
    return T::name;
  }
}
```

```
template <typename T>
constexpr auto getName() {
  static_assert(HasName<T>, "T must have a 'name' member");
  if constexpr (is_function_v<T::name>) {
    return T::name();
  } else {
    return T::name;
  }
}
```

```
template <typename T, typename = void>
constexpr inline bool HasName = false;

template <typename T>
constexpr inline bool
HasName<T, void_t<decltype(T::name)>> = true;
```

```
template <typename T, typename = void>
constexpr inline bool HasName = false;

template <typename T>
constexpr inline bool
HasName<T, void_t<decltype(T::name)>> = true;
```

```
template <typename T, typename = void>
constexpr inline bool HasName = false;

template <typename T>
constexpr inline bool
HasName<T, void_t<decltype(T::name)>> = true;
```

```
template <typename T, typename = void>
struct HasName {
  static constexpr bool value = false;
};

template <typename T>
struct HasName<T, void_t<decltype(T::name)>> {
  static constexpr bool value = true;
};
```

```
namespace detail {
  template <typename T, typename = void>
  constexpr inline bool HasName = false;

  template <typename T>
  constexpr inline bool
  HasName<T, void_t<decltype(T::name)>> = true;
}

template <typename T>
constexpr inline bool HasName = detail::HasName<T>;
```

```
template <typename T, typename = void>
constexpr inline bool HasName = false;

template <typename T>
constexpr inline bool
HasName<T, void_t<decltype(T::name)>> = true;
```

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

```
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>;
  //
}
```

```
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, typename Ps>
class HasProperties {
public:
  template <typename T>
  T::property_type::type GetProperty();

  template <typename T>
  void SetProperty(T::property_type::type value);

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

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 <typename T>
  auto 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 <typename T>
  void SetProperty(T::property_type::type value) {
    using type = T::property_type::type;
    auto success =
      m_propertyHolder.template GetProperty<type>(
        getName<T>(), std::move(value));
    assert(success);
  }
  // ...
};
```

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

```
namespace internal {
  template <typename T, typename... LTs>
  constexpr bool is_type_in_list(PropertyList<LTs...>) {
    return (std::is_same_v<T, LTs> || ...);
  }
}


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

```
namespace internal {
  template <typename T>
  constexpr bool is_type_in_list(PropertyList<>) {
    return false;
  }

  template <typename T, typename LT, typename... LTs>
  constexpr bool is_type_in_list(PropertyList<LT, LTs...>)
{
    if (std::is_same<T, LT>::value) {
      return true;
    }
      return is_type_in_list<T>(PropertyList<LTs...>{});
    }
```

```
namespace internal {
  template <typename T, typename... LTs>
  constexpr bool is_type_in_list(PropertyList<LTs...>) {
    return (std::is_same_v<T, LTs> || ...);
  }
}


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

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

  template <typename T>
  void SetProperty(T::property_type::type value);

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

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

  // ...
};
```

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

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

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

```
template <typename T>
constexpr auto getDefaultValue() {
  static_assert(HasDefaultValue<T>);
  if constexpr (is_function_v<T::name>) {
    return T::defaultValue();
  } else {
    return T::defaultValue;
  }
}
```

```
template <typename T, typename = void>
constexpr inline bool HasDefaultValue = false;

template <typename T>
constexpr inline bool
HasDefaultValue<T, void_t<decltype(T::defaultValue)>>
=true;
```

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

```
template <typename Derived, typename 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();
    });
  }
};
```

```
template <typename Derived, typename Ps>
class HasProperties {
private:
  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{});
        }
        this_->SetNeedsLayout();
        this_->SetNeedsDisplay();
    });
  }
```

```
template <typename T, typename P, typename = void>
constexpr inline bool HasDidSet = false;

template <typename T, typename P>
constexpr inline bool HasDidSet<T, P,
  std::void_t<decltype(std::declval<T>().didSet(P{}))>> =
    true;
```

```
template <typename T, typename P, typename = void>
constexpr inline bool HasDidSet = false;

template <typename T, typename P>
constexpr inline bool HasDidSet<T, P,
  std::void_t<decltype(std::declval<T>().didSet(P{}))>> =
    true;
```

```
template <typename T, typename P, typename = void>
constexpr inline bool HasDidSet = false;

template <typename T, typename P>
constexpr inline bool HasDidSet<T, P,
  std::void_t<decltype(std::declval<T>().didSet(P{}))>> =
    true;
```

```
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 (IsLayoutProperty<P>) {
        this_->SetNeedsLayout();
      }
      if constexpr (IsDisplayProperty<P>) {
        this_->SetNeedsDisplay();
      }
  });
```

```
struct LayoutProperty {};

struct DisplayProperty {};

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

```
template <typename T>
constexpr inline bool IsLayoutProperty =
  is_base_of_v<LayoutProperty, T>;

template <typename T>
constexpr inline bool IsDisplayProperty =
  is_base_of_v<DisplayProperty, T>;
```

```
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};
  };
  // ...
```

```
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 (IsLayoutProperty<P>) {
        this_->SetNeedsLayout();
      }
      if constexpr (IsDisplayProperty<P>) {
        this_->SetNeedsDisplay();
      }
  });
```

```
template <typename P>
void connectDidSet() {
  if constexpr (HasDidSet<Derived,P> ||
                IsLayoutProperty<P>  ||
                IsDisplayProperty<P>) {
    m_propertyHolder.GetPropSignal(getName<P>()).Connect(
      &m_trackable, [this_ = static_cast<Derived*>(this)] {
      // ...
    });
  }
}
```

```
template <class Derived, class Ps, class PropertyHolder>
class HasProperties {
public:
  template <typename T>
  T::property_type::type GetProperty();

  template <typename T>
  void SetProperty(T::property_type::type value);
protected:
  HasProperties();
private:
  PropertyHolder m_propertyHolder;
  PropertyHolder::DidSetToken m_trackable;
};
```

```
class PropertyHolderConcept {
public:
  using DidSetToken = Trackable;
  void CreateProperty(string_view name,
                      string_view typeName,
                      any value);
  template <typename T>
  T GetProperty(string_view name);
  template <typename T>
  void SetProperty(string_view name, T value);
  template <typename T>
  void ConnectDidSet(string_view name, T&& didSetFn);
};
```

```
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);
}
```

Thank you.

Glass Properties

https://github.com/izotope/glassproperties

rmichaels@izotope.com
roth@rothmichaels.us
@thevibesman