



BUG-FREE JUCE UI

*STRUCTURE YOUR GUI CODE FOR STABILITY, TESTABILITY
& CLEAN ARCHITECTURE*

JAN WILCZEK

What is this talk about?

- Make your JUCE UI bug-free
- Applies to app and plugin UIs
- A way to think about JUCE GUI architecture in general

Slides:

<https://github.com/JanWilczek/adc24-talk>



Who is this talk for?

- Mid & senior devs working in JUCE
- Juniors with JUCE experience
- C++ developers who write GUIs

Who am I?

- Jan Wilczek
- Audio Programming Consultant & Educator
- Author of TheWolfSound.com blog & YouTube channel
 - youtube.com/@WolfSoundAudio
- Author of the DSP Pro online course on learning Digital Signal Processing from scratch (no maths, no programming background)
 - wolfsoundacademy.com/dsp-pro
- ADC Mentor



Why this talk?

- Teaching JUCE & C++ experience
- Consulting experience: this year's moderately-sized app project in JUCE
 - lack of high-quality reference GUI code
- Dev experience: Developing my own synth live

Definitions

Model: code representation of the domain.

- interacts with the outside world via *ports* (abstract interfaces) implemented with *adapters* (GUI, console commands, web requests, etc.)
- consists of *entities* and *use cases*

View: code responsible for displaying content to the user.

- e.g., subclasses of `juce::Component`

Component: `juce::Component`

Fake: unit test-only piece of code mimicking dependencies

- e.g., fake database, fake file, fake HTTP client etc.
- should be fast, ideally just return a value

Definitions

UI state: state of the UI at a given time

- Completely captures how the UI should look and behave

UI logic: code that controls the UI state (UI appearance and behavior)

- e.g., component positioning, button click handling, etc.

Problem 1

How to make sure your GUI works as expected?

Common GUI interactions

- interactions that change the Model, that are driven by the domain
 - e.g., preset selected
- interactions that change the GUI
 - e.g., theme switched from light to dark
- generic interactions
 - e.g., app opened, app closed
- updates from the Model
- updates from the runtime environment
 - e.g., screen resolution changed

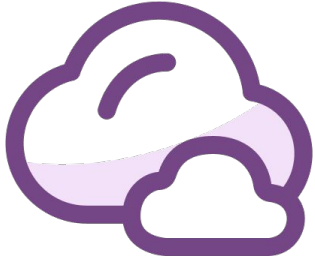
Is my GUI code correct?

- Is a correct value displayed?
- What if an error occurs?
- Is every possible interaction covered and meaningful?
- What happens when the app is resized?





slido



How can I ensure the GUI code works correctly?

- ① Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

How can I ensure the GUI code works correctly?

- Edit and Pray (~Michael Feathers)
- Manual tests
- QA testing
- End-2-end GUI tests that somehow mimic user UI actions (click, key press, etc.)

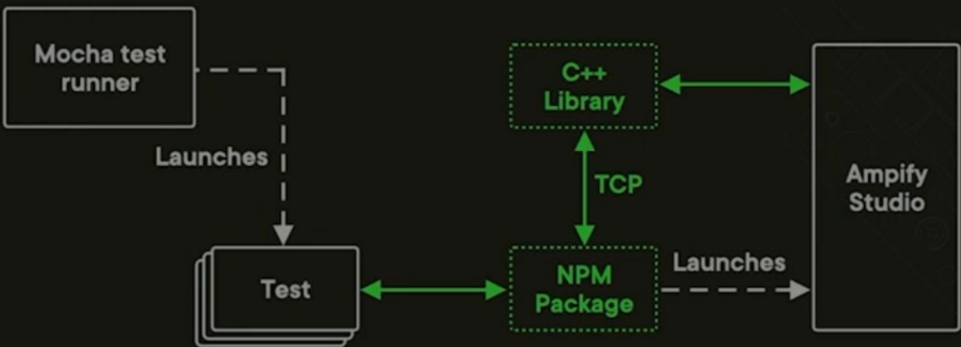
```
[TestMethod]
public void TestDoSomething()
{
    // given
    var app = Application.Launch("MyApp.exe");
    var window = app.GetWindow("My App Window Title", InitializeOption.NoCache);
    var btnMyButton = window.Get<Button>("btnMyButtonName");

    // when
    btnMyButton.Click();

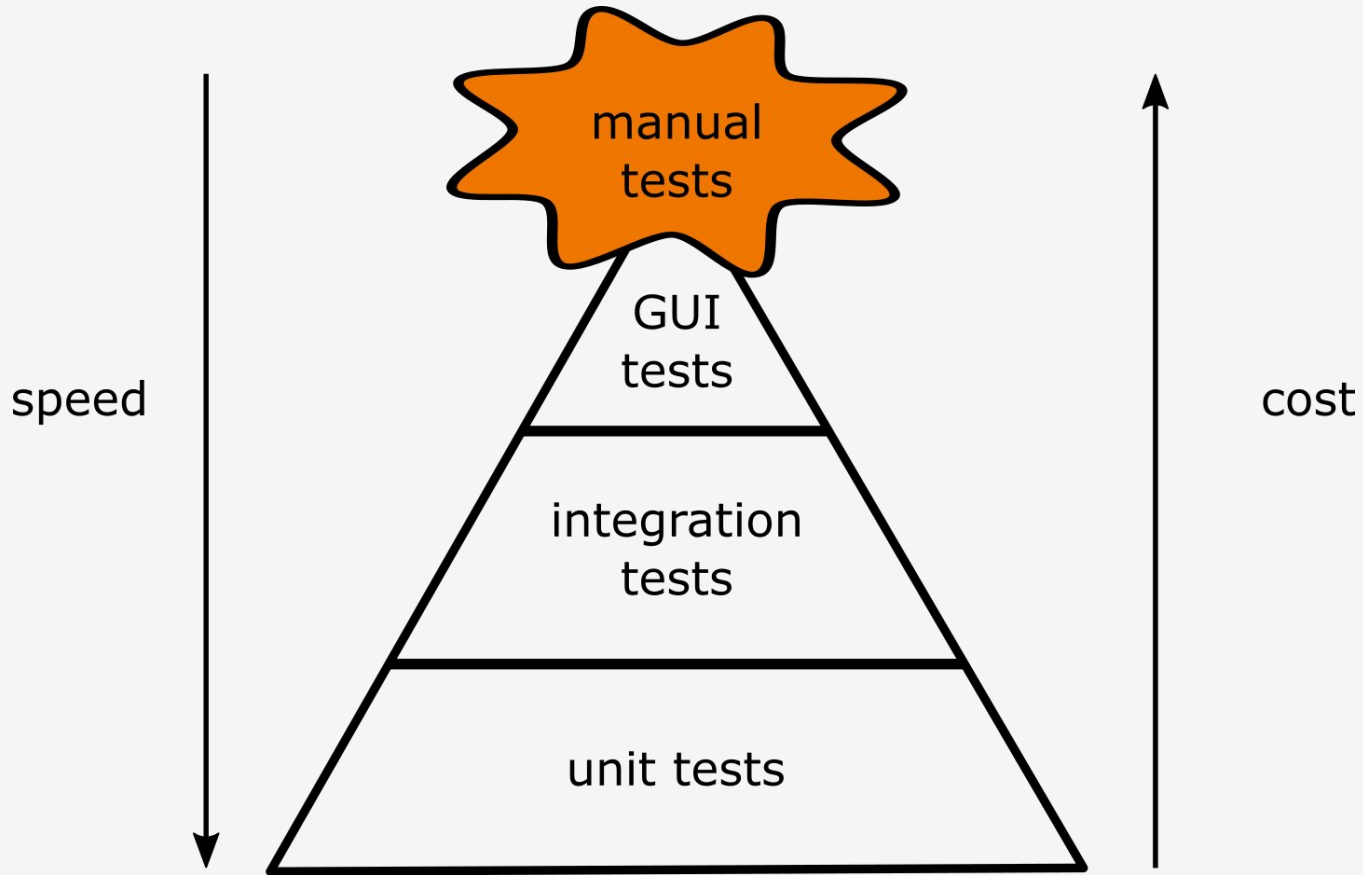
    // then
    var txtMyTextBox = window.Get<TextBox>("txtMyTextBox");
    Assert.IsTrue(txtMyTextBox.Text == "my expected result");

    // cleanup
    app.Close();
}
```


Our Solution



Joe Noël



Is there a better solution?

- fast
- standard C++
- no specialized testing framework
- assert all desired effects
- easy CI/CD integration

Problem 2

How to write automated GUI tests?

WebView UIs (JUICE 8)

- Established end-2-end testing frameworks for
 - DOM interaction
 - backend REST calls
 - triggering JavaScript events
- What about "traditional" JUICE Components?

Typical JUCE Component

```
class CustomComponent : public juce::Component {
public:
    CustomComponent() {
        button_.onClick = [] {
            // do stuff
        };
    }

private:
    juce::TextButton button_{"Click me!"};
};
```

Testability Obstacle 1

```
class CustomComponent : public juce::Component {  
public:  
    CustomComponent() {  
        button_.onClick = [] {  
            // do stuff  
        };  
    }  
  
private:  
    juce::TextButton button_{"Click me!"};  
};
```

we cannot call this callback
in unit test*



```
class CustomComponent : public juce::Component {
public:
    CustomComponent(juce::ValueTree model) : model_{model} {
        button_.onClick = [this] {
            if (auto child = model_.getChildWithProperty("NESTING", nesting); child.isValid()) {
                nesting++;
                child.addChild(juce::ValueTree{"CHILD", {{ "NESTING", nesting}}, {}}, nullptr);
            }
        };
    }

private:
    juce::TextButton button_{"Click me!"};
    juce::ValueTree model_;
    int nesting = 0;
};
```

```
class CustomComponent : public juce::Component {
public:
    CustomComponent(juce::ValueTree model) : model_{model} {
        button_.onClick = [this] {
            if (auto child = model_.getChildWithProperty("NESTING", nesting); child.isValid()) {
                nesting++;
                child.addChild(juce::ValueTree{"CHILD", {{ "NESTING", nesting}}, {}}, nullptr);
            }
        };
    }
};
```



Model manipulation in the View

```
private:
    juce::TextButton button_{"Click me!"};
    juce::ValueTree model_;
    int nesting = 0;
};
```

```
class CustomComponent : public juce::Component,
                        private juce::ValueTree::Listener {
public:
    CustomComponent(juce::ValueTree model) : model_{model} {
        model_.addListener(this);
    }

private:
    void valueTreePropertyChanged(juce::ValueTree& treeWhosePropertyHasChanged,
                                  const juce::Identifier &property) {
        if (treeWhosePropertyHasChanged == model_ && property == "BUTTON_TEXT") {
            button_.setButtonText(treeWhosePropertyHasChanged.getProperty("BUTTON_TEXT"));
        }
    }

    juce::TextButton button_{"Click me!"};
    juce::ValueTree model_;
};
```




```
class CustomComponent : public juce::Component,  
                        private juce::ValueTree::Listener {
```

```
public:
```

```
    CustomComponent(juce::ValueTree model) : model_{model} {  
        model_.addListener(this);  
    }
```

dependence on the Model
in the View



```
private:
```

```
    void valueTreePropertyChanged(juce::ValueTree& treeWhosePropertyHasChanged,  
                                  const juce::Identifier &property) {  
        if (treeWhosePropertyHasChanged == model_ && property == "BUTTON_TEXT") {  
            button_.setButtonText(treeWhosePropertyHasChanged.getProperty("BUTTON_TEXT"));  
        }  
    }  
}
```

cannot assert it



```
    juce::TextButton button_{"Click me!"};  
    juce::ValueTree model_;  
};
```

Typical JUCE Component's responsibilities

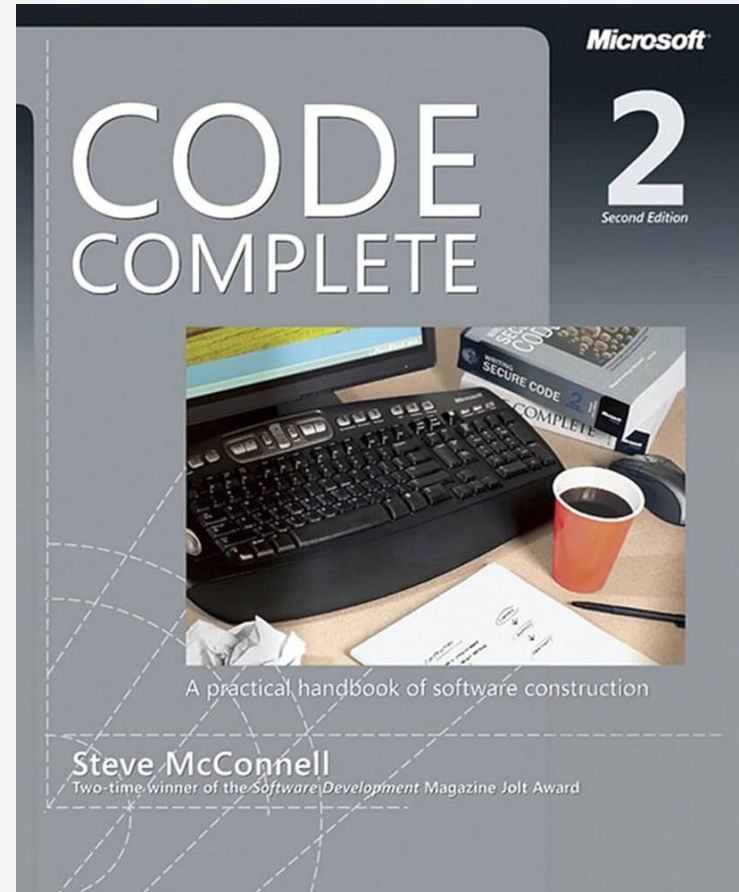
- holds UI widgets
- holds UI state
- positions the widgets
- paints additional stuff
- handles user actions
- manipulates the Model
- observes for changes to the Model
- manipulates other Components
- ...it simply does too much!





*"Program **into** your language,
not in it."*

Steve McConnell
Code Complete, 2nd edition

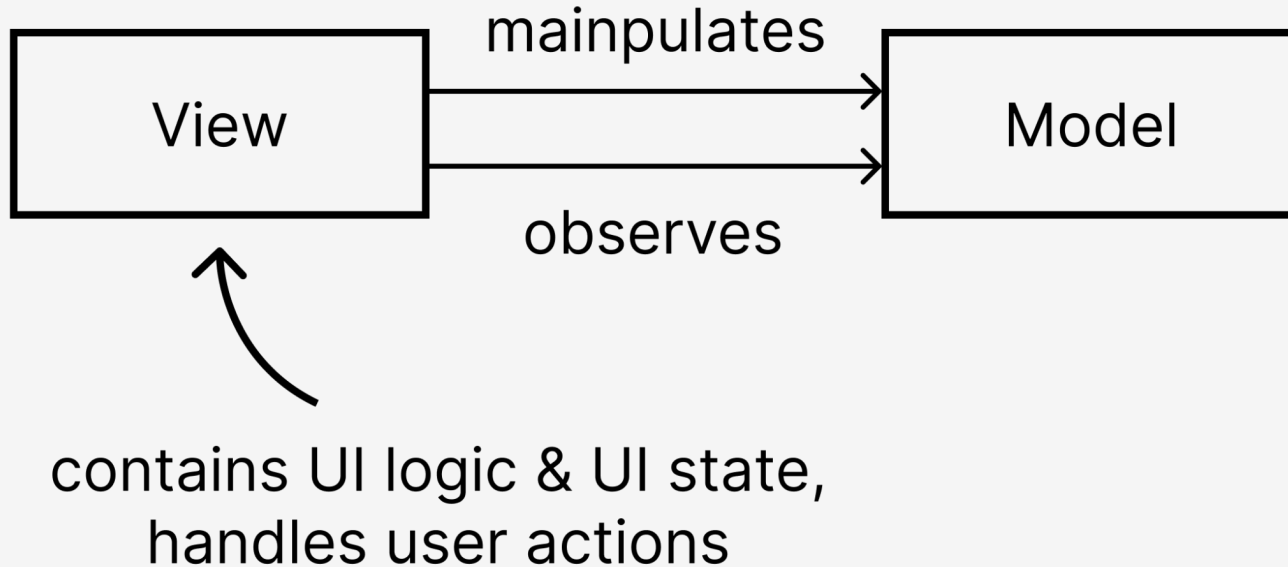


Model-View-ViewModel (MVVM) pattern

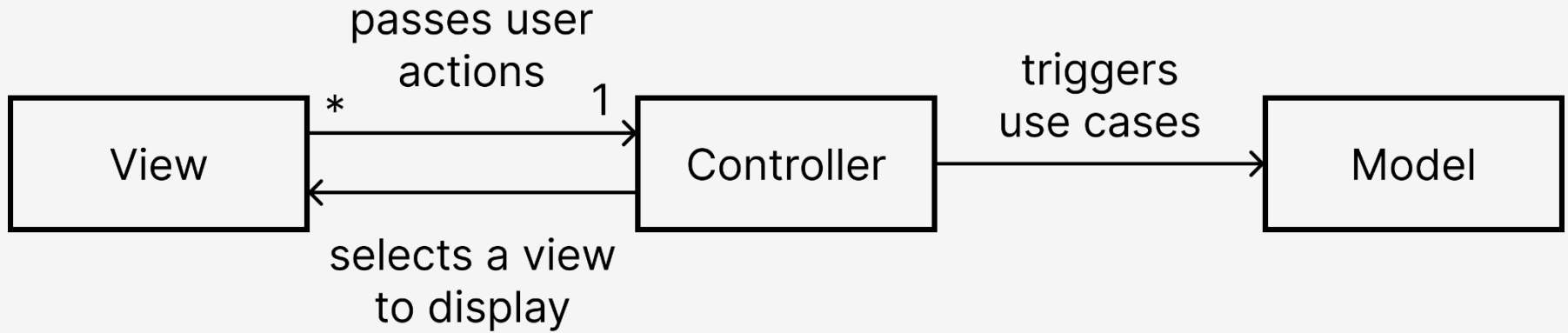


A tour of GUI architectures

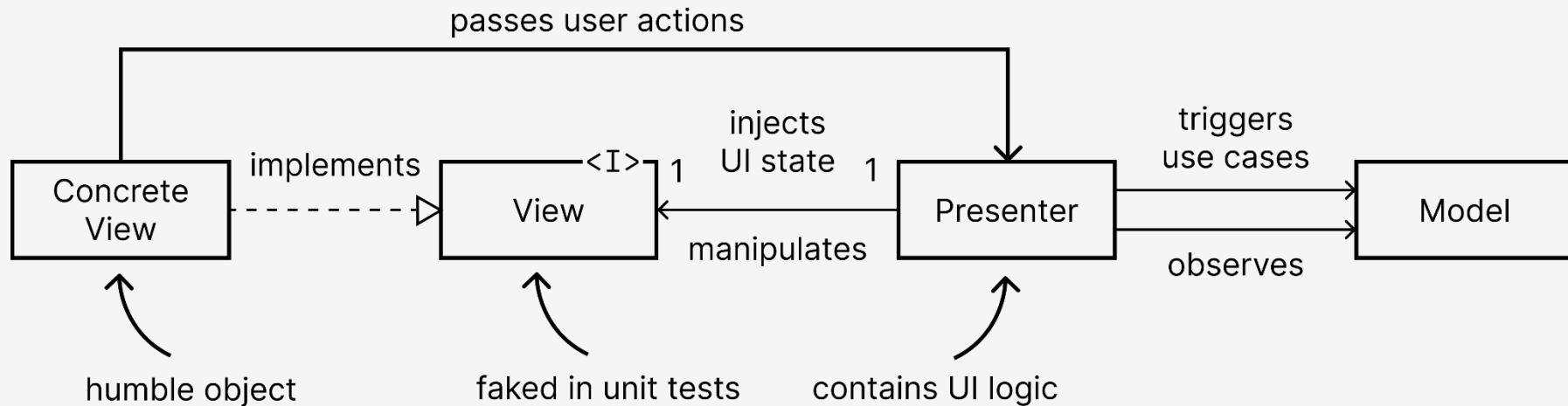
"Put It All In The Component" (PITA)



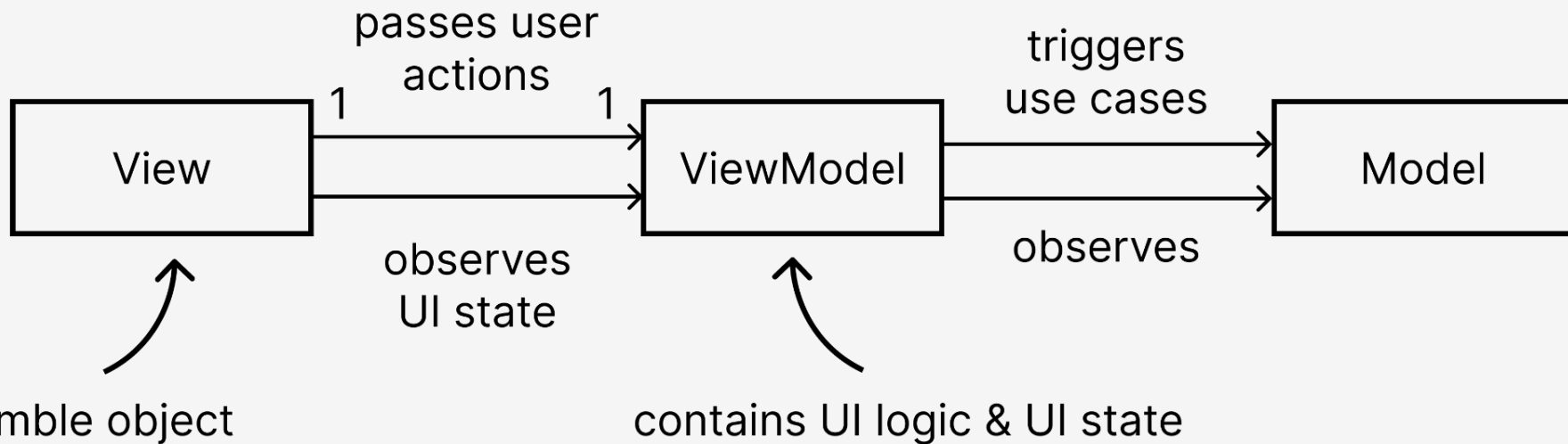
Model-View-Controller (MVC)



Model-View-Presenter (MVP) / Passive View

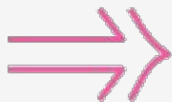
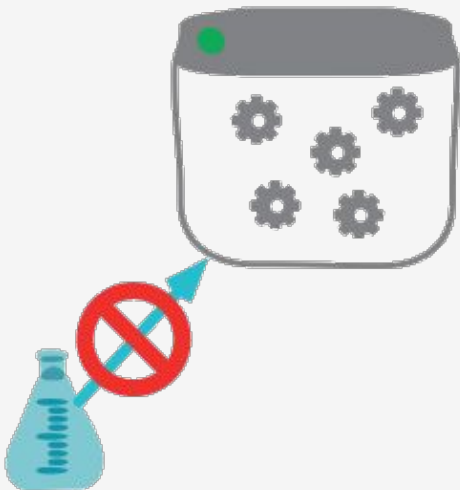


Model-View-ViewModel (MVVM) / Presentation Model

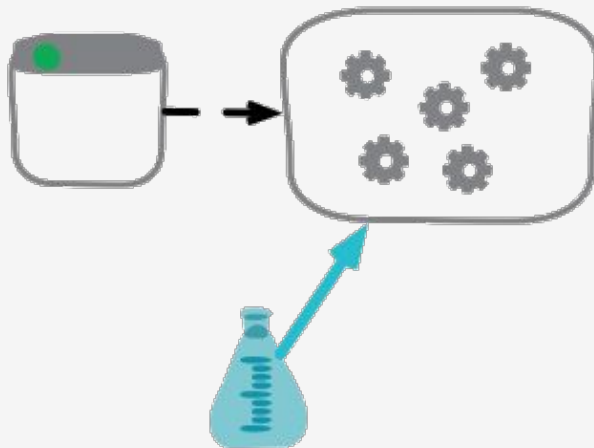


Humble object (~Michael Feathers)

*faced with a software element
that's difficult to test*



*move the logic into a separate
element that is testable, making
the original object **humble***



<https://martinfowler.com/bliki/HumbleObject.html>

```

class MainActivity : ComponentActivity() {
    @Composable
    private fun PlayControl(modifier: Modifier, synthesizerViewModel: WavetableSynthesizerViewModel) {
        val playButtonLabel = synthesizerViewModel.playButtonLabel.observeAsState()

        PlayControlContent(modifier = modifier,
            onClick = {
                synthesizerViewModel.playClicked()
            },
            buttonLabel = stringResource(playButtonLabel.value!!))
    }

    @Composable
    private fun PlayControlContent(modifier: Modifier, onClick: () -> Unit, buttonLabel: String) {
        Button(modifier = modifier,
            onClick = onClick) {
            Text(buttonLabel)
        }
    }
}

```

```
class WavetableSynthesizerViewModel : ViewModel() {  
    var wavetableSynthesizer: WavetableSynthesizer? = null  
        set(value) {  
            field = value  
            applyParameters()  
        }  
  
    private val _playButtonLabel = MutableLiveData(R.string.play)  
    val playButtonLabel: LiveData<Int>  
        get() {  
            return _playButtonLabel  
        }  
}
```

```

class WavetableSynthesizerViewModel : ViewModel() {
    fun playClicked() {
        viewModelScope.launch {
            if (wavetableSynthesizer?.isPlaying() == true) {
                wavetableSynthesizer?.stop()
            } else {
                wavetableSynthesizer?.play()
            }
            updatePlayButtonLabel()
        }
    }

    private fun updatePlayButtonLabel() {
        viewModelScope.launch {
            if (wavetableSynthesizer?.isPlaying() == true) {
                _playButtonLabel.value = R.string.stop
            } else {
                _playButtonLabel.value = R.string.play
            }
        }
    }
}

```

Solution: bug-free JUCE UI

Q: How to make your UI code bug-free?

A: Unit test it.

Q: How to test `juce::Component` subclasses?

A: You can't without breaking the invariants or a lot of boilerplate code.

Q: How to test the UI logic then?

1. Separate the UI logic and state to a ViewModel.
2. Make the `juce::Component` subclass a humble object.
3. Unit test the ViewModel.

Problem 3

How to use MVVM in C++ (JUICE)?

All You Need Is Data Binding

- Observer pattern
- Event-Subscriber
- Event-Listener
- Observer-Listener
- Publisher-Subscriber
- Callback
- Signals and Slots
- Events & Event Handlers

Data binding in WinUI (XAML)

```
<Window x:Class="Quickstart.MainWindow" ...>
  <Grid>
    <TextBlock Text="{x:Bind ViewModel.TextBlockContent}"
      HorizontalAlignment="Center"
      VerticalAlignment="Center"/>
  </Grid>
</Window>
```

juce::CachedValue<T> is not an option

*This class acts as a typed wrapper around a **property inside a ValueTree**.*

- Does not allow observing
- Exposes too much from the Model

juce::Value is not an option

*Contains a reference to a var object, and can get and set its value.
Listeners can be attached to be told when the value is changed.*

- Allows observing but...
- Is dynamically typed
- Is weakly typed
- Observing is unhandy (via the Listener interface)

```

using ScopedConnection = boost::signals2::scoped_connection;

template <typename T>
class ObservableProperty {
public:
    using SignalType = boost::signals2::signal<void(const T&)>;
    [[nodiscard]] ScopedConnection observe(const typename SignalType::slot_type& onChangedCallback) {
        return onChanged_.connect(onChangedCallback);
    }
    [[nodiscard]] const T& value() const noexcept { return value_; }
protected:
    explicit ObservableProperty(const T& initialValue) : value_{initialValue} {}
    void setValueAndNotify(const T& newValue) {
        value_ = newValue;
        onChanged_(this->value_);
    }
private:
    T value_;
    SignalType onChanged_;
};

```

```
template <typename T>
class MutableObservableProperty : public ObservableProperty<T> {
public:
    explicit MutableObservableProperty(const T& initialValue = {})
        : ObservableProperty<T>(initialValue) {}

    void setValue(const T& newValue) {
        if (newValue != this->value()) {
            setValueForced(newValue);
        }
    }

    void setValueForced(const T& newValue) { this->setValueAndNotify(newValue); }
};
```

```
// Calls f on the message (main, GUI) thread if is available. Otherwise, calls f directly.
void callOnMessageThreadIfNotNull(std::function<void()> f);

template <typename T>
class LiveObservableProperty : public MutableObservableProperty<T> {
public:
    explicit LiveObservableProperty(const T& initialValue = {})
        : MutableObservableProperty<T>(initialValue) {}

    void postValue(const T& newValue) {
        callOnMessageThreadIfNotNull(
            [this, newValue] { this->setValue(newValue); });
    }

    void postValueForced(const T& newValue) {
        callOnMessageThreadIfNotNull(
            [this, newValue] { this->setValueForced(newValue); });
    }
};
```



```
class Holder {
public:
    [[nodiscard]] ObservableProperty<int>& property() noexcept {
        return property_;
    }

    void incrementValue() { property_.setValue(property_.value() + 1); }

private:
    MutableObservableProperty<int> property_{0};
} holder;

const auto connection = holder.property().observe([&](const int& value) {
    // callback code
});

// holder.property().setValue(1);    // Doesn't work although property() is public.
holder.incrementValue();    // Tightly restricted API
```

```
class Holder {
public:
    Holder() : property_{0} {}

    OBSERVABLE_PROPERTY(property, int)

    void incrementValue() { property_.setValue(property_.value() + 1); }
} holder;

const auto connection = holder.property().observe([&](const int& value) {
    // callback code
});

// holder.property().setValue(1);    // Doesn't work although property() is public.
holder.incrementValue();    // Tightly restricted API
```

observable-property

<https://github.com/JanWilczek/observable-property>

- C++, header-only
- contains boost-signals2
- Boost license
(even more permissive than MIT)
- supports CMake workflow
- suggestions and PRs welcome!



Example of MVVM in a JUCE app

Plotting a filter's magnitude response

```
class EqFilterViewModel {
public:
    using CutoffFrequencyChangedUseCase =
        std::function<void(double newCutoffFrequencyHz)>;

    explicit EqFilterViewModel(
        CutoffFrequencyChangedUseCase onCutoffFrequencyChanged)
        : frequencySliderValue_{100.},
          cutoffFrequencyChanged_{std::move(onCutoffFrequencyChanged)} {}

    LIVE_OBSERVABLE_PROPERTY(frequencySliderValue, double)

    void onCutoffFrequencyChanged(double newValue) {
        cutoffFrequencyChanged_(newValue);
    }

private:
    CutoffFrequencyChangedUseCase cutoffFrequencyChanged_;
};
```

```
class EqFilterComponent : public juce::Component {  
public:  
    // ...  
private:  
    std::unique_ptr<EqFilterViewModel> viewModel_;  
    std::vector<ScopedConnection> connections_;  
    juce::Slider frequencySlider_;  
};
```

```
explicit EqFilterComponent(std::unique_ptr<EqFilterViewModel> viewModel)
    : viewModel_{std::move(viewModel)} {
    // ... (slider range, style setup, addAndMakeVisible(), etc.) ...
    frequencySlider_.setValue(viewModel_->frequencySliderValue().value(),
                               juce::dontSendNotification);
    frequencySlider_.onValueChange = [this] {
        viewModel_->onCutoffFrequencyChanged(frequencySlider_.getValue());
    };
    connections_.push_back(
        viewModel_->frequencySliderValue().observe([this](double newValue) {
            frequencySlider_.setValue(newValue, juce::dontSendNotification);
        }));
}
```

```
class EqFilter {
public:
    OBSERVABLE_PROPERTY(magnitudeResponse, dsp::MagnitudeResponse)

    void onCutoffFrequencyChanged(double newCutoffFrequency) {
        cutoffFrequency_ = newCutoffFrequency;
        magnitudeResponse_.setValueForced(calculateMagnitudeResponse());
    }
private:
    [[nodiscard]] dsp::MagnitudeResponse calculateMagnitudeResponse() const {
        /* magnitude response calculations */ return result;
    }

    double cutoffFrequency_{100.};
};
```



```

class MagnitudeResponsePlotViewModel {
public:
    // ...
    LIVE_OBSERVABLE_PROPERTY(plot, juce::Path)
private:
    void updatePlot() { plot_.setValueForced(calculateMagnitudeResponsePlot()); }

    [[nodiscard]] juce::Path calculateMagnitudeResponsePlot() const {
        /* calculations involving magnitudeResponse_ and plotBounds_ */ return result;
    }
    juce::Rectangle<int> plotBounds_;
    dsp::MagnitudeResponse magnitudeResponse_;
    std::vector<ScopedConnection> connections_;
};

```

```

explicit MagnitudeResponsePlotViewModel(
    ObservableProperty<dsp::MagnitudeResponse>& magnitudeResponse)
    : magnitudeResponse_(magnitudeResponse.value()) {
connections_.push_back(
    magnitudeResponse.observe([this](const auto& newResponse) {
        magnitudeResponse_ = newResponse;
        updatePlot();
    }));
}

void onPlotBoundsChanged(const juce::Rectangle<int>& newBounds) {
    plotBounds_ = newBounds;
    updatePlot();
}

```

```
class PlotComponent : public juce::Component {
public:
    // ...
private:
    void drawPlot(juce::Graphics& g) {
        g.setColour(juce::Colours::white);
        g.setOpacity(1.f);
        g.strokePath(plotViewModel_>plot().value(), juce::PathStrokeType{5.f});
    }

    std::unique_ptr<MagnitudeResponsePlotViewModel> plotViewModel_;
    std::vector<ScopedConnection> connections_;
};
```

```
explicit PlotComponent(  
    std::unique_ptr<MagnitudeResponsePlotViewModel> plotViewModel)  
    : plotViewModel_{std::move(plotViewModel)} {  
    connections_.push_back(plotViewModel_->plot().observe(  
        [this](const juce::Path&) { repaint(); }));  
}  
  
void paint(juce::Graphics& g) override { drawPlot(g); }  
  
void resized() override {  
    plotViewModel_->onPlotBoundsChanged(getLocalBounds());  
}
```

Wiring

```
EqFilter model;
```

```
EqFilterComponent eqFilterComponent{  
    std::make_unique<EqFilterViewModel>([&](double newCutoffFrequencyHz) {  
        model.onCutoffFrequencyChanged(newCutoffFrequencyHz);  
    }));  
};
```

```
PlotComponent plotComponent{  
    std::make_unique<MagnitudeResponsePlotViewModel>(  
        model.magnitudeResponse()));  
};
```

Example unit test

// given

```
MutableObservableProperty<dsp::MagnitudeResponse> magnitudeResponse{  
    /* frequencies and gains */};  
MagnitudeResponsePlotViewModel testee{magnitudeResponse};
```

// when

```
testee.onPlotBoundsChanged(juce::Rectangle<int>{0, 0, 100, 100});
```

// then

```
ASSERT_EQ(juce::Path{/* correct values */}, testee.plot().value());
```

ONE DOES NOT SIMPLY

USE MVVM EVERYWHERE

MVVM problems

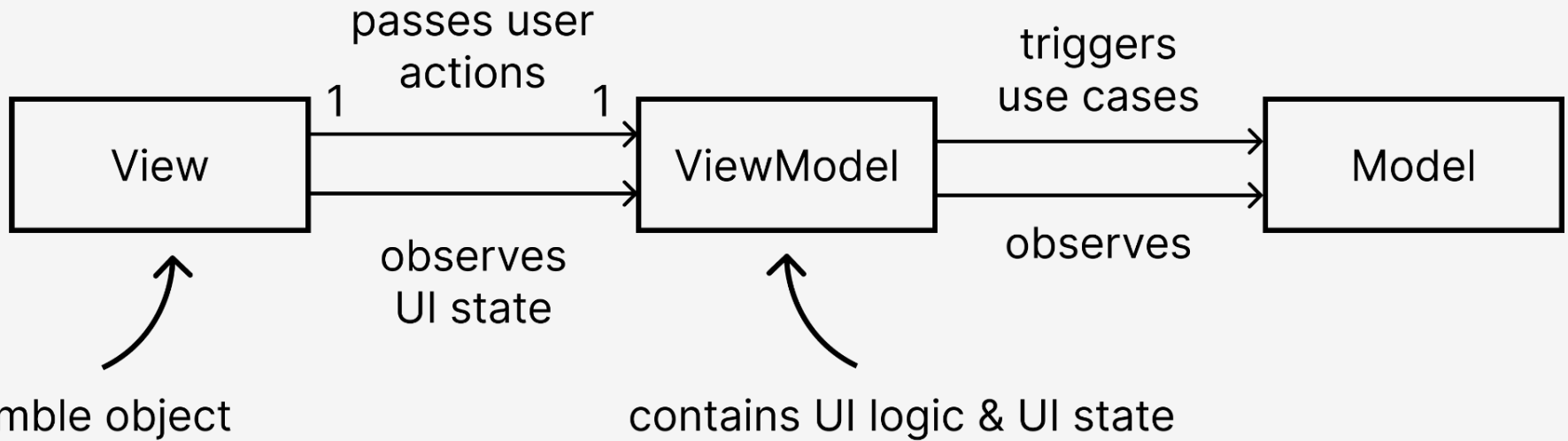
- Boiler-plate code
 - Storing connections
 - Exposing properties
- All bindings must be established
 - View → ViewModel
 - ViewModel → Model
- Model and ViewModel must not forget to notify their observers
- Requires 3rd-party libraries for implementing data binding
- Future-proof?

Recap

- You can create bug-free JUCE UIs by unit testing them.
- You can make JUCE GUIs easily testable by leveraging the MVVM pattern.
- *"Program into Your Language, Not in It"*
 - Examples: ViewModel for the Components, `ObservableProperty<T>`

Recap

- MVVM vs other GUI patterns



Recap

- MVVM enables testability of the GUI code by decoupling framework code from UI state and UI logic
- The core of MVVM is data binding
 - Examples: `ObservableProperty<T>`, `LiveObservableProperty<T>`
- MVVM can be safely applied to JUCE UIs
 - Make Components Humble Classes
 - Make Components
 - observe the ViewModel for UI state changes
 - notify the ViewModel on user actions
- MVVM supports multithreaded environments

References

- **McConnell, Steve.** *Code Complete: A Practical Handbook of Software Construction*. 2nd ed., Microsoft Press, 2004.
- **Martin, Robert C.** *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.
- **Feathers, Michael C.** *Working Effectively with Legacy Code*. Prentice Hall, 2004.
- **Bakker, Jelle.** ["Vars, Values, and ValueTrees: State Management in JUCE." *Audio Developer Conference \(ADC\)*, 2023.](#)
- **Noel, Joe.** ["End-to-end Testing of a JUCE Desktop Application." *Audio Developer Conference \(ADC\)*, 2021.](#)
- **Feathers, Michael C..** ["The Humble Dialog Box." *martinfowler.com*. Accessed 8 Nov. 2024.](#)
- **"Presentation Model."** [*Microsoft Learn*, Microsoft. Accessed 8 Nov. 2024.](#)
- **"Guide to App Architecture: the UI Layer."** [*Android Developers*, Google. Accessed 1 Nov. 2024.](#)
- **"Windows data binding overview."** [*Microsoft Learn*, Microsoft. Accessed 8 Nov. 2024.](#)
- **"Windows data binding in depth."** [*Microsoft Learn*, Microsoft. Accessed 8 Nov. 2024.](#)

Thank you for attending!

- Join my newsletter
thewolfound.com/newsletter
- Slides
github.com/JanWilczek/adc24-talk
- Subscribe to my YouTube channel
youtube.com/@WolfSoundAudio
- DSP Pro course
wolfoundacademy.com/dsp-pro

