Using JUCE in Large Projects

Tom Poole

Using JUCE in Large Projects

Codebase Modularity
Keeping Code Sustainable
Code Reviews

GIT Repo Tricks & Tips
CI Tricks & Tips
Team and Org Structure

Codebase Modularity

Modules: When to break up a codebase

- Just getting too large
- Slow build times
- New project started with some shared code
- Obvious library-like classes
- Premature librarification?

Modules: How to break up a codebase

- Minimise inclusion
- Minimise coupling
- Library dependencies should not contain cycles
- Always include a whole library, don’t cherry-pick sub-files
- How widely to use a single namespace? Use sub-namespaces?
- As far as possible, treat each folder as a kind of library

Using JUCE in Large Projects

Modules: JUCE modules vs static vs dynamic libs

- JUCE Modules (or just folders of .cpp & .h files)
- Static libs
- Dynamic libs
- Global configuration flags

Using JUCE in Large Projects

```
/*******************************************************************************
The block below describes the properties of this module, and is read by
the Projucer to automatically generate project code that uses it.
For details about the syntax and how to create or use a module, see the
JUCE Module Format.txt file.


BEGIN_JUCE_MODULE_DECLARATION

 ID:               juce_graphics
 vendor:           juce
 version:          5.2.0
 name:             JUCE graphics classes
 description:      Classes for 2D vector graphics, image loading/saving, font handling, etc.
 website:          http://www.juce.com/juce
 license:          GPL/Commercial

 dependencies:     juce_events
 OSXFrameworks:    Cocoa QuartzCore
 iOSFrameworks:    CoreGraphics CoreImage CoreText QuartzCore
 linuxPackages:    x11 xinerama xext freetype2

END_JUCE_MODULE_DECLARATION

*******************************************************************************/


#pragma once
#define JUCE_GRAPHICS_H_INCLUDED

#include <juce_core/juce_core.h>
#include <juce_events/juce_events.h>

#include "geometry/juce_AffineTransform.h"
#include "geometry/juce_Point.h"
#include "geometry/juce_Line.h"
#include "geometry/juce_Rectangle.h"
#include "placement/juce_Justification.h"
...
```

Modules: Reducing build times

- Full rebuild vs most common rebuild
- Libs vs source files
- Headers are often the culprit
- -ftime-report
- Sweet spot of about 10-20 kloc per compile unit
- Precompiled headers and other tricks

Using JUCE in Large Projects

Modules: Dealing with multiple apps, or app variations

- Common to have multiple projects with overlapping functionality
- Structure as separate projects, or #ifdef FEATURE_ENABLED ?
- If using a single project with versions, keep disabled code to a minimum

Modules: Adding 3rd Party Libraries

- Always hesitate before adding a dependency.
- Prefer libraries with more open and flexible build systems where possible: header-only vs closed-source DLL.
- Beware boost build times!
- Cautionary tale of ROLI's recent use of a closed-source static lib.

Using JUCE in Large Projects

Modules: Let's look at some real-world examples...

- NOISE & Equator
- Tracktion engine

Keeping Code Clean and Sustainable

Clean coding: Stick to some coding style guidelines

- C++ Core Guidelines:
  http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- More extensive rule-sets from JUCE,
  LLVM (https://llvm.org/docs/CodingStandards.html),
  Google (Abseil: https://abseil.io/tips/)
- Importance of both superficial and functional rules

Clean coding: Refactoring

- Rule of Three (Don Roberts):

- The first time you do something, just do it.

- The second time you do something similar, you wince at the duplication but still do it.

- The third time, you refactor.

Clean coding: Refactoring

- Hitting an actual performance/scalability problem
- Bad code smells
- Paying off accumulated technical debt
- New features or projects push the architecture in new directions
- You spot a better way to implement something
- Inheriting an old code base
- During a code review

Clean coding: When NOT to Refactor

- Aesthetics...?
- Boredom..?
- Finding the right level of refactoring effort

Clean coding: Finding the right abstraction level

- Are you an under- or over-abstractor?
- Symptoms of ‘under’: globals, large functions, god classes
- Symptoms of ‘over’: over-generality leading to code bloat/waste
- Both produce code that’s hard to maintain!

Clean coding: Optimisation

- The First Rule of Code Optimization: *Don't.*
- The Second Rule of Code Optimization: *Don't yet.*
- The Third Rule of Code Optimization: *Profile first.*

- The Fourth rule (for audio developers):

  You might as well start optimising the real-time bits straight away

Clean coding: Optimisation

- Understand hard real-time (see Timur Doumler's talk)
- Learn when to trust the compiler
- Use Godbolt
- Compiler optimisations aren't always intuitive (e.g. pass-by-value, devirtualisation)
- "Optimised" != "Unreadable"
- Learn about CPU architectures and how they might affect performance on different platforms
- Take advantage of choosing your compiler and build tools

Clean coding: Unit tests

- JUCE contains the UnitTest class
- We use the UnitTestRunner project
- Test code in JUCE modules is included via a preprocessor macro

Clean coding: Tools to help you

- High warning levels
- Asan, Tsan, undefined behaviour checker, valgrind
- Performance tools
- Static analyser
- JUCE_LEAK_DETECTOR, JUCE_PREVENT_HEAP_ALLOCATION
- WeakReference, not_null, spans
- TRANS and Projucer’s translation tools

Clean coding: Encourage a culture of clean coding within the team

- Code reviews
- Pair programming
- Dev group discussion and teaching sessions

Clean coding: Top 10 code rules to help with large codebases

1. Have a **strict** set of coding rules!
2. Warnings are errors!
3. Appropriate, self-explanatory naming
4. RAII
5. Composition > Inheritance
6. Optimise for readability
7. Decouple as much as possible
8. #ifdef as little platform-specific code as possible
9. JUCE_LEAK_DETECTOR and friends
10. Use override, final, constexpr, [[nodiscard]], enum class (these are all safety-focused modern C++ features)

Code Reviews

Code reviews: Don’t just take our word for it!

- Characteristics of Useful Code Reviews: An Empirical Study at Microsoft (an analysis of 1.5 million review comments!)
- Best Kept Secrets of Peer Code Review
- Code Complete

Code reviews: Code Complete

“... software testing alone has limited effectiveness – the average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 and 60 percent.”

“In a group of 11 programs developed by the same group of people, the first 5 were developed without reviews. The remaining 6 were developed with reviews. After all the programs were released to production, the first 5 had an average of 4.5 errors per 100 lines of code. The 6 that had been inspected had an average of only 0.82 errors per 100. Reviews cut the errors by over 80 percent.”

Code reviews: Why are effective code reviews important?

- Code reviews take time

  > “Inspection rates of less than 300 LOC/hour result in the best defect detection, and rates under 500 are still good, but expect to miss a significant percentage of defects if LOC are reviewed faster than that.”

- Code reviews are most effective when performed by senior software developers from the same team

Code reviews: Reviewer selection

- Get developers to re-check their own work first!
- No more than two reviewers
- Reviewers **without domain specific knowledge** can only spot **superficial** flaws
- Do not give important reviewing tasks to new starters
- Your best developers, team leads and technical architects should spend the most time reviewing code

Code reviews: Focus on the important things

- High risk parts of the codebase
- **Not** formatting; use a code style guide/reference!
- Functional correctness
- Coding errors (“<=” instead of “<”)
- Design mistakes
- Omissions or corner cases
- Anything else that might be difficult to test

Code reviews: Maintain short feedback cycles

- Context switching is hard, especially when you're thinking about the subtleties of someone else's changes (possibly embedded in a third party's section of the codebase)
- Aim to complete both code reviews and corrections within 24 hours of the original code review request

Code reviews: A teaching tool - useful, but...

- Pair programming is more effective
- “Over the shoulder” quick reviews

  The reviewer looks over the author’s shoulder (or watches via remote desktop) and the author walks them through the code changes. This shouldn’t be treated as a drop-in replacement for a formal code review though.

Code reviews: Other bits

- Consider a code of conduct or similar - people can be defensive of their work and a reference detailing what to expect from all parties may be useful
- Keep changesets small - the rate at which defects are detected decreases rapidly after around 90 minutes of reviewing.

Interactive code-review session

Using JUCE in Large Projects

```
void MainComponent::resized()
{
    const int height = 50;

    for (int i = 0; i < Audio::NumFilePlayers; i++)
        filePlayerGuis[i].setBounds (0, i*height,
                                     getWidth(), height);

    mastGain.setBounds (0, 300, 80, 3*height);
    mastText.setBounds (0, 300 + 3*height, 80, height);

    const int halfWidth = getWidth()/2;
    slvGain.setBounds (halfWidth, 300, 80, height);
    slvText.setBounds (halfWidth, 300 + height, 80, height);
}
```

Using JUCE in Large Projects

```
void MainComponent::resized()
{
    const int height = 50;

    for (int i = 0; i < Audio::NumFilePlayers; i++)
        filePlayerGuis[i].setBounds (0, i*height,
                                     getWidth(), height);

    mastGain.setBounds (0, 300, 80, 3*height);
    mastText.setBounds (0, 300 + 3*height, 80, height);

    const int halfWidth = getWidth()/2;
    slvGain.setBounds (halfWidth, 300, 80, height);
    slvText.setBounds (halfWidth, 300 + height, 80, height);
}
```

```
void MainComponent::resized()
{
    auto area = getLocalBounds();
    auto topPanel = area.removeFromTop (300);
    constexpr int height = 50, columnWidth = 80;

    for (auto& gui : filePlayerGuis)
        gui.setBounds (topPanel.removeFromTop (height));

    auto rightPanel = area.removeFromRight (area.getWidth() / 2);

    auto leftColumn = area.removeFromLeft (columnWidth);
    masterGain.setBounds (leftColumn.removeFromTop (3 * height));
    masterText.setBounds (leftColumn.removeFromTop (height));

    auto rightColumn = rightPanel.removeFromLeft (columnWidth)
    slaveGain.setBounds (rightColumn.removeFromTop (height));
    slaveText.setBounds (rightColumn.removeFromTop (height));
}
```

Using JUCE in Large Projects

```
NBase::Result Demo::openMidiDevice()
{
 auto devices = MidiInput::getDevices();
 int deviceIndex = -1;
 int index = 0;
 for (auto& device: devices) {
   if (SMatchSubStringNoCase(device.toStdString(), "ableton push 2")) {
     deviceIndex = index;
     break;
   }
   index++;
 }

 if (deviceIndex == -1) {
   return NBase::Result("Failed to find input midi device for push2");
 }

 // Try opening the device
 auto input = MidiInput::openDevice(deviceIndex, this);
 if (!input) {
   return NBase::Result("Failed to open input device");
 }

 // Store and starts listening to the device
 midiInput_.reset(input);
 midiInput_->start();

 return NBase::Result::NoError;
}
```

Using JUCE in Large Projects

```
NBase::Result Demo::openMidiDevice()
{
 auto devices = MidiInput::getDevices();
 int deviceIndex = -1;
 int index = 0;
 for (auto& device: devices) {
   if (SMatchSubStringNoCase(device.toStdString(), "ableton push 2")) {
     deviceIndex = index;
     break;
   }
   index++;
 }

 if (deviceIndex == -1) {
   return NBase::Result("Failed to find input midi device for push2");
 }

 // Try opening the device
 auto input = MidiInput::openDevice(deviceIndex, this);
 if (!input) {
   return NBase::Result("Failed to open input device");
 }

 // Store and starts listening to the device
 midiInput_.reset(input);
 midiInput_->start();

 return NBase::Result::NoError;
}
```

```
NBase::Result Demo::openMidiDevice()
{
   auto deviceIndex = MidiInput::getDevices().indexOf ("ableton push 2", true);

   if (deviceIndex < 0)
       return NBase::Result ("Failed to find input midi device for push2");

   if (auto* input = MidiInput::openDevice (deviceIndex, this))
   {
       midiInput.reset (input);
       midiInput->start();
       return NBase::Result::NoError;
   }

   return NBase::Result ("Failed to open input device");
}
```

Using JUCE in Large Projects

```
class DoSomethingWidget
{
   // foobarHandler must not be null; it will NOT be owned by this object
   DoSomethingWidget (FoobarHandler* foobarHandler);
   DoSomethingWidget (const DoSomethingWidget&);

   // foobarHandler must not be null; it will NOT be owned by this object
   void setFoobarHandler (FoobarHandler* foobarHandler);
   // This will never return a nullptr
   FoobarHandler* getFoobarHandler();

   // The Splangle will be owned by this object. It may be null.
   void setSplangle (Splangle* splangle);
   Splangle* getSplangle();

   // Returns the parent widget, or null if there isn't one
   DoSomethingWidget* getParentWidget();

   enum State
   {
       stopped,
       started
   };

   void setState (const State&);
   void setBounds (const Rectangle<int>& newBounds);

   // returns true if the widget is activated
   bool activated();

   Thingy* getThingy(); // Creates and returns a new Thinhy - caller must take ownership
   Splong* getSplong(); // Returns a pointer to the current Splong. This will never be null.
   Wungle* getWungle(); // Returns a pointer to the Wungle, which is a ref-counted object

   void getPositionOfWoggle (int& x, int& y);
   void setWongleRange (double start, double end);
   void addWoggle (Woggle* woggle, int x, int y);

private:
   FoobarHandler* handler;
   Splangle* splangle;
};
```

Using JUCE in Large Projects

```
class DoSomethingWidget
{
   // foobarHandler must not be null; it will NOT be owned by this object
   DoSomethingWidget (FoobarHandler* foobarHandler);
   DoSomethingWidget (const DoSomethingWidget&);

   // foobarHandler must not be null; it will NOT be owned by this object
   void setFoobarHandler (FoobarHandler* foobarHandler);
   // This will never return a nullptr
   FoobarHandler* getFoobarHandler();

   // The Splangle will be owned by this object. It may be null.
   void setSplangle (Splangle* splangle);
   Splangle* getSplangle();

   // Returns the parent widget, or null if there isn't one
   DoSomethingWidget* getParentWidget();

   enum State
   {
       stopped,
       started
   };

   void setState (const State&);
   void setBounds (const Rectangle<int>& newBounds);

   // returns true if the widget is activated
   bool activated();

   Thingy* getThingy(); // Creates and returns a new Thinhy - caller must take ownership
   Splong* getSplong(); // Returns a pointer to the current Splong. This will never be null.
   Wungle* getWungle(); // Returns a pointer to the Wungle, which is a ref-counted object

   void getPositionOfWoggle (int& x, int& y);
   void setWongleRange (double start, double end);
   void addWoggle (Woggle* woggle, int x, int y);

private:
   FoobarHandler* handler;
   Splangle* splangle;
};
```

```
class DoSomethingWidget
{
   DoSomethingWidget (FoobarHandler& foobarHandler);
   DoSomethingWidget (const DoSomethingWidget&);
   DoSomethingWidget (DoSomethingWidget&&) = default;

   void setFoobarHandler (FoobarHandler& foobarHandler);
   FoobarHandler& getFoobarHandler() const noexcept;

   void setSplangle (std::unique_ptr<Splangle> splangle);
   const Splangle* getSplangle() const noexcept;

   non_owned<DoSomethingWidget> getParentWidget();

   enum class State
   {
       stopped,
       started
   };

   void setState (State);
   void setBounds (Rectangle<int> newBounds);
   bool isActivated() const noexcept;

   std::unique_ptr<Thingy> createThingy() const;
   Splong& getSplong() const noexcept;
   ReferenceCountedObjectPtr<Wungle> getWungle() const;

   Point<int> getPositionOfWoggle() const noexcept;
   void setWongleRange (Range<double> range);
   void addWoggle (std::unique_ptr<Woggle> woggle, Point<int> position);

private:
   FoobarHandler& handler;
   std::unique_ptr<Splangle> splangle;

   JUCE_LEAK_DETECTOR (DoSomethingWidget)
};
```

Using JUCE in Large Projects

```
class DoSomethingWidget
{
   // foobarHandler must not be null; it will NOT be owned by this object
   DoSomethingWidget (FoobarHandler* foobarHandler);
   DoSomethingWidget (const DoSomethingWidget&);

   // foobarHandler must not be null; it will NOT be owned by this object
   void setFoobarHandler (FoobarHandler* foobarHandler);
   // This will never return a nullptr
   FoobarHandler* getFoobarHandler();

   // The Splangle will be owned by this object. It may be null.
   void setSplangle (Splangle* splangle);
   Splangle* getSplangle();

   // Returns the parent widget, or null if there isn't one
   DoSomethingWidget* getParentWidget();

   enum State
   {
       stopped,
       started
   };

   void setState (const State&);
   void setBounds (const Rectangle<int>& newBounds);

   // returns true if the widget is activated
   bool activated();
```

```
class DoSomethingWidget
{
   DoSomethingWidget (FoobarHandler& foobarHandler);
   DoSomethingWidget (const DoSomethingWidget&);
   DoSomethingWidget (DoSomethingWidget&&) = default;

   void setFoobarHandler (FoobarHandler& foobarHandler);
   FoobarHandler& getFoobarHandler() const noexcept;

   void setSplangle (std::unique_ptr<Splangle> splangle);
   const Splangle* getSplangle() const noexcept;

   non_owned<DoSomethingWidget> getParentWidget();

   enum class State
   {
       stopped,
       started
   };

   void setState (State);
   void setBounds (Rectangle<int> newBounds);
   bool isActivated() const noexcept;
```

Using JUCE in Large Projects

```
    Thingy* getThingy(); // Creates and returns a new Thinhy - caller must
take ownership
    Splong* getSplong(); // Returns a pointer to the current Splong. This
will never be null.
    Wungle* getWungle(); // Returns a pointer to the Wungle, which is a
ref-counted object

    void getPositionOfWoggle (int& x, int& y);
    void setWongleRange (double start, double end);
    void addWoggle (Woggle* woggle, int x, int y);

private:
    FoobarHandler* handler;
    Splangle* splangle;
};
```

```
    std::unique_ptr<Thingy> createThingy() const;
    Splong& getSplong() const noexcept;
    ReferenceCountedObjectPtr<Wungle> getWungle() const;

    Point<int> getPositionOfWoggle() const noexcept;
    void setWongleRange (Range<double> range);
    void addWoggle (std::unique_ptr<Woggle> woggle, Point<int> position);

private:
    FoobarHandler& handler;
    std::unique_ptr<Splangle> splangle;

    JUCE_LEAK_DETECTOR (DoSomethingWidget)
};
```

GIT Repository Tips & Tricks

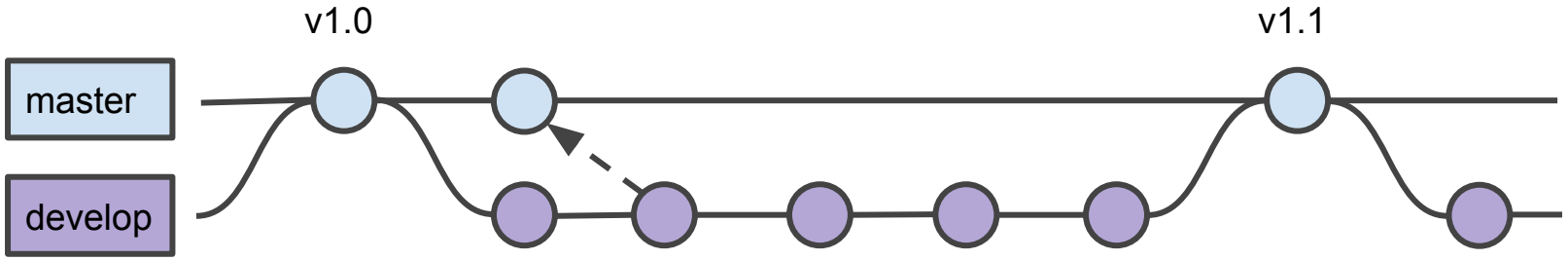
GIT repos: One big repo?

- Google (2 billion lines) + Facebook
- Our experience at ROLI
- Changing shared code makes it easier to spot the repercussions
- Branch hygiene rules needed
- Commit message rules
- Locking branches
- Website code also in the same repo?

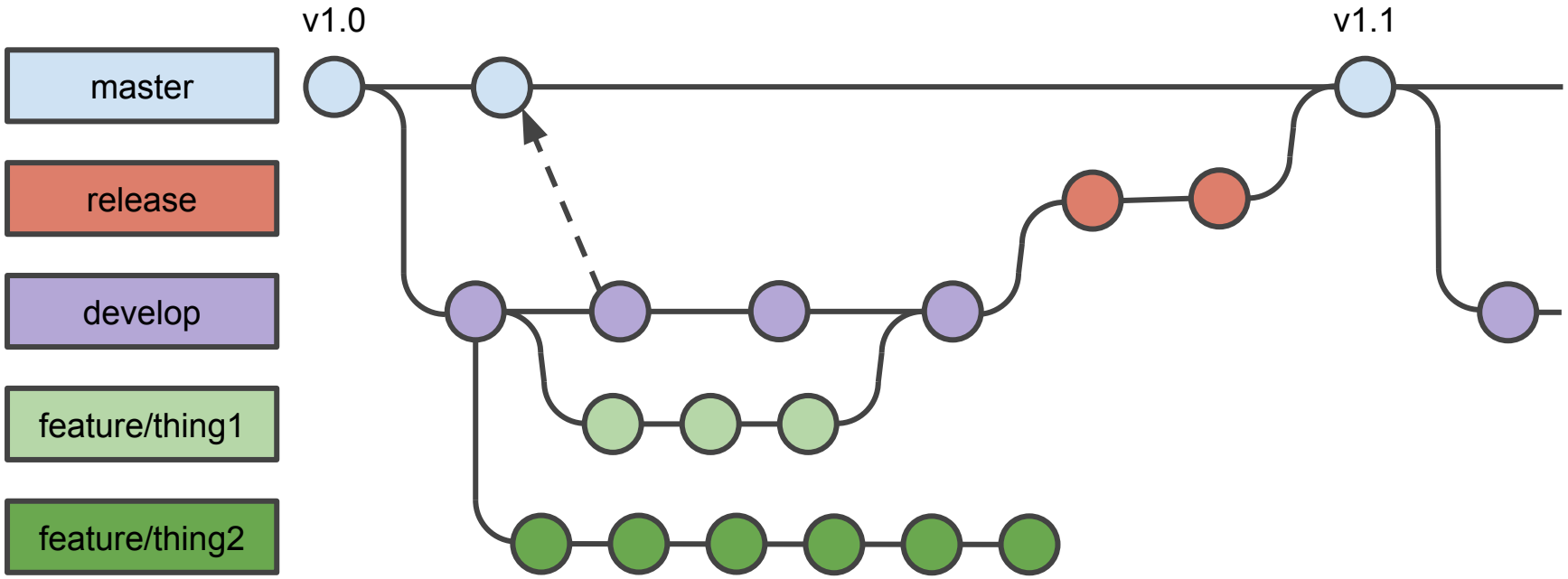
GIT repos: Where and how to host

- Github vs an internal server
- Pull requests
- Familiar to new team members

GIT repos: Workflow - JUCE from the outside



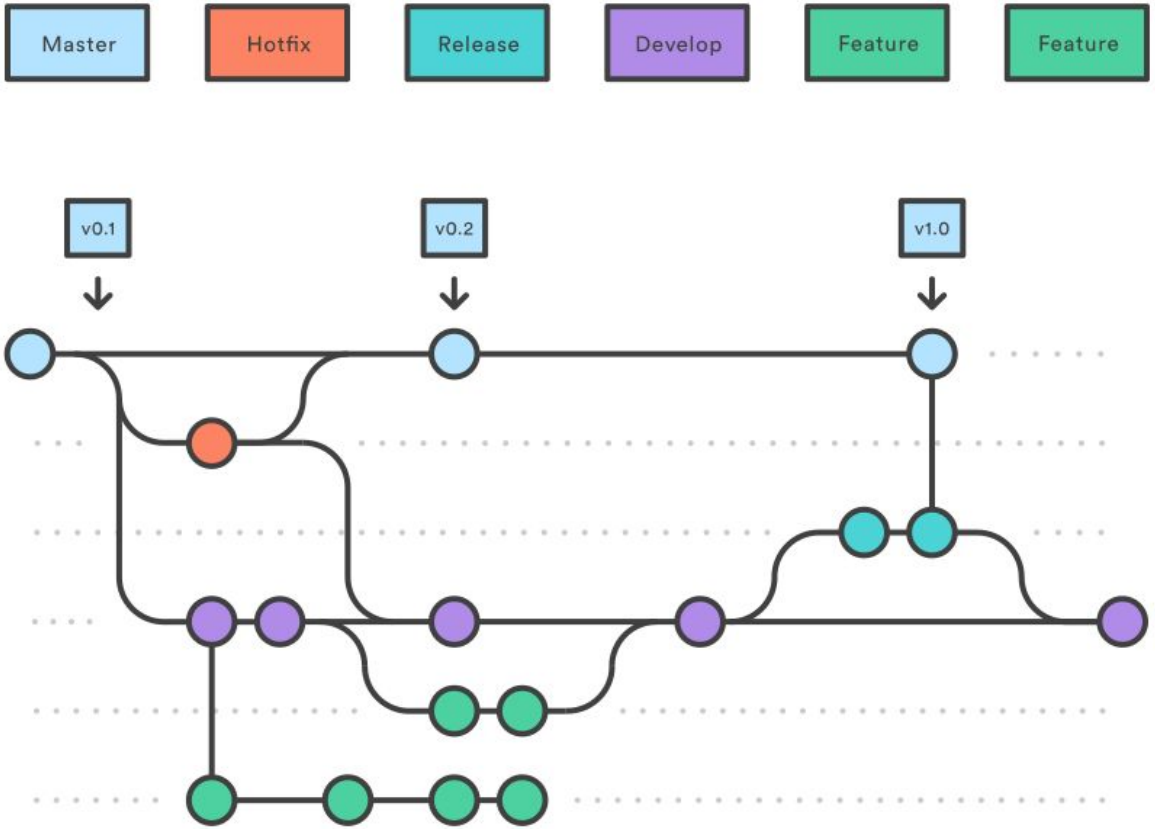
GIT repos: Workflow - JUCE from the inside



GIT repos: Gitflow

Image from Atlassian's excellent guide at

https://www.atlassian.com/git/tutorials/comparing-workflows



GIT repos: Using Submodules

- 3rd party libs that must be in their own repo
- Also consider git subtrees if everything is internal
- Syncing
- Can be a solution to handling large binary files

GIT repos: Dealing with large binaries

- Use a throwaway sub-module
- Use a large-file-system add-on
- Use your own binary file server and a custom build step

CI servers and build automation

CI servers and build automation: Tools

- Use the Projucer's command line interface
- Build all things on all platforms
- Frameworks: Jenkins, Buildbot, ...
- Use framework-independent configuration, if possible

CI servers and build automation: Using branches

- Run tests selectively based on branch name
  - master
  - develop
  - bugfix/*
  - feature/*

- Only allow pushes to public or sensitive branches after CI complete

Team and Organisation Structure

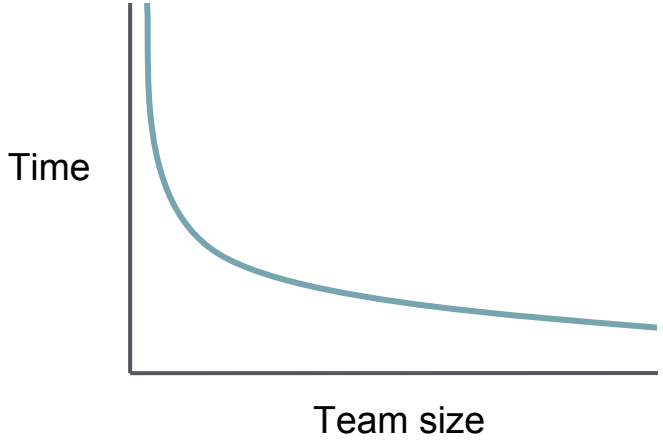
Team and Organisation Structure: Importance

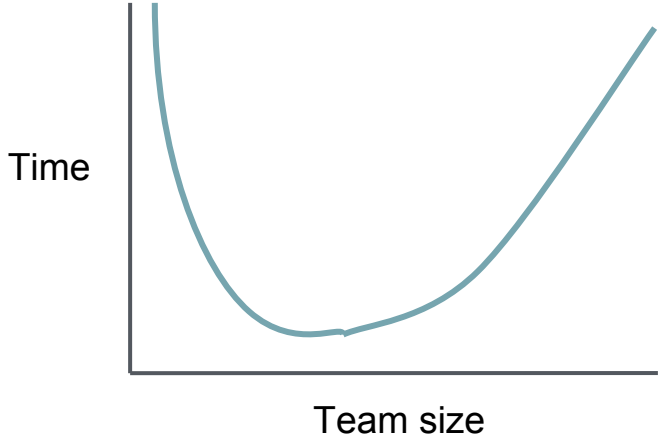
“Organisations which design systems are constrained to produce designs which are copies of the communication structures of these organisations."

Melvin Conway - How Do Committees Invent?

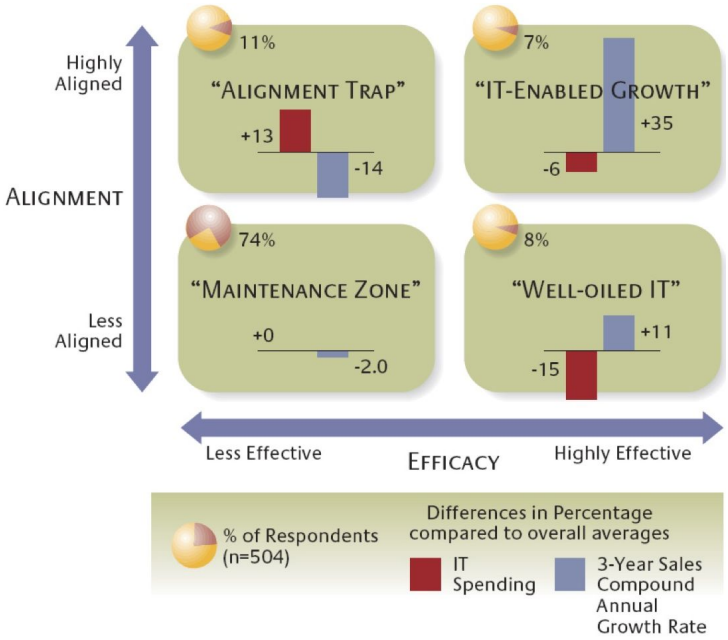
Team and Organisation Structure: Amdahl's Law

- Keep team sizes small
- $T = T_1 (1 - p (n - 1) / n) + k n (n - 1) / 2$

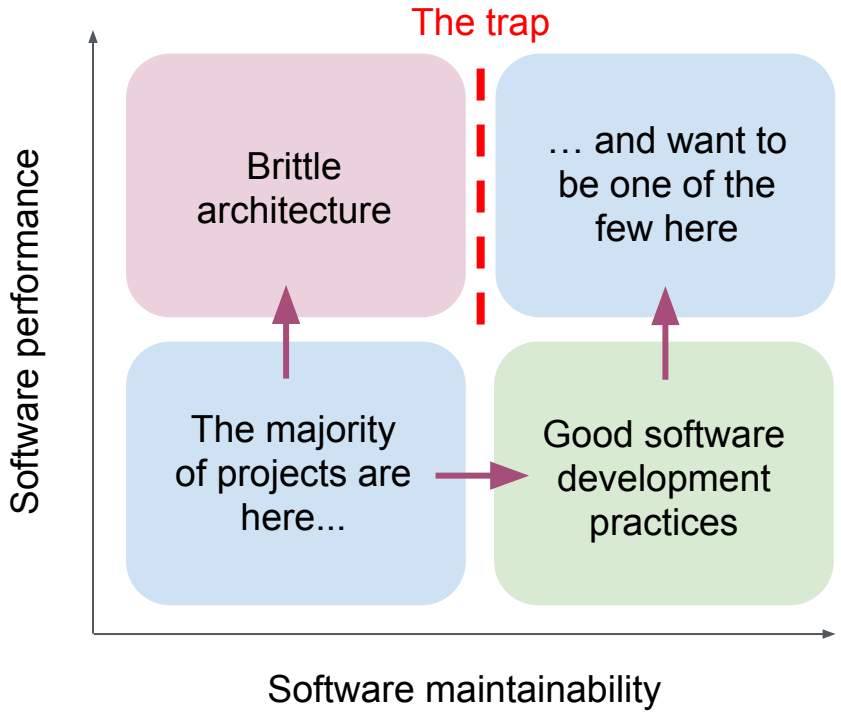




Team and Organisation Structure: Avoiding the "Alignment Trap"



"Avoiding the alignment trap in IT" - MIT Sloane Management Review



Org Structure: Software

Everyone should use the same tools

- Slack
- Jira

- SmartGit
- Parallels

Using JUCE in Large Projects

Thank you!