In software development we are constantly required to solve complex problems. We often find ourselves implementing complicated solutions to deal with these complex problems. As a result we end up with code that is hard to follow, difficult to reason about, and a maintenance nightmare.

The best quote I've heard in that vein is: "You want simple solutions to complex problems. If you have a complex solution to a complex problem you’re f\*\*\*\*\*!"

Truer words have never been spoken.

Implementing complex solutions to complex problems always result in a sea of edge cases, unexpected behavior, constant maintenance, and of course, bugs. This isn't ground breaking news. We've all worked on systems like this and probably implemented a few ourselves.

There are lots of common approaches to dealing with complex problems. Break the problem down into smaller chunks, modularize, etc. Many of these can be found in design pattern books. I'd like to take a moment and discuss a recent simple solution I encountered while starting work in UI (something I've done very little of in my career) using the Qt framework.

Qt use the well-known ModelView design pattern. All data is defined in a Model and Views are used to display and request edits to data in a Model. This split between data and view helps to prevent the UI code from having an impact on the data design.

What makes Qt's ModelView usage so powerful is the ability to use just about any view to display data from just about any model. Doing this requires the models and views to have a shared API that they both interface with to describe data. This API needs to work with all types of data from a simple list of strings to a complex node graph. I've seen lots of complex solutions to this type of problem, but really liked Qt's solution: QModelIndex.

Every piece of data in a model can be represented by a QModelIndex. A QModelIndex contains a row, column, and parent field. A QModelIndex is just an index into a cell in a table, with a parent. The parent allows hierarchy of data to be represented.

For example, suppose your model had three pieces of data: A, B, C.  Let’s imagine C happens to be a child of A. We could represent those pieces of data with the following QModelIndicies:

QModelIndex root\_table\_index\_a = QModelIndex( 0, 0, QModelIndex()      );  //  QModelIndex( row, column, parent )

QModelIndex root\_table\_index\_b = QModelIndex( 1, 0, QModelIndex()      );

QModelIndex child\_table\_index  = QModelIndex( 0, 0, root\_table\_index\_a );

That’s it. You can use this scheme to index any piece of data in your model. Yes, for complex models the data->QModelIndex mapping will need to be more sophisticated. Though that logic is contained inside the model which knows about the data, instead of needing to inject explicit knowledge of the data layout inside custom UI elements.

What’s really cool about this approach is once you have the data->QModelIndex mapping implemented for a model, that model can then be used with just about any view. Pretty awesome!