



Modernizing Legacy Systems

Your 10 Step Guide to Software Migration

Matthias Kalle Dalheimer

The word “migration” makes most of us think about birds leaving a cold, barren landscape and flying on a long, arduous journey to a warm, more fertile location. Software migration takes a metaphorical journey not unlike this – it’s a lengthy process that sees hundreds of modules move from an obsolete framework to a much more preferable one. However, migration isn’t instinctive for humans like it is for geese. There are a number of pitfalls in a large porting effort that can significantly increase time, cost, and complexity, leading to risk of project derailment.

Regardless of whether your current system was built using MFC, Motif, Photon, Delphi, or Tcl/Tk, moving the user interface portion can be one of your most difficult tasks. You may have languages, frameworks, and windowing systems that are no longer supported or that leave you unable to find knowledgeable talent. What’s more, legacy UIs tend to look very dated compared to today’s fluid animations, context-sensitive controls, and responsive interactions, requiring more than just a simple port to bring them up to date.

This whitepaper is a detailed guide to help you effectively evaluate whether a migration makes sense for your current system, and to help you outline and execute your own.

**All software
eventually
becomes
obsolete and
it's only
natural to
forestall its
replacement**

At KDAB, we've fine-tuned the migration process over 15 years, successfully migrating software from a wide variety of frameworks – with the majority migrating to Qt. Based on our experience, we know that regardless of the framework, operating system, or language, all migrations share common steps that ensure success and have common pitfalls that derail the best of efforts. This whitepaper distills our expertise in a short but detailed guide to help you effectively evaluate whether a migration makes sense for your current system, and to help you outline and execute your own.

Identifying the tipping point

If your organization is responsible for a large legacy project, you know the pain of accumulated technical debt. Perhaps this shows itself as an inability to change hardware platforms, make a timely response to bug fixes, or improve a dated user interface. You may have been limping along in maintenance mode for a while – how do you know when you definitively need to make a change?

A sign that your software's current incarnation may soon pass the point of diminishing returns is when a majority of scheduled work is no longer spent on bug fixes or preventative maintenance but rather on adapting to new platforms or adding new requirements. As the system continues to be modified for purposes it was never originally designed, successive changes become more prone to introducing new issues, resulting in a continual expansion in engineering estimates. The underlying software becomes brittle, which shows up in several ways – new bugs after every added feature, increasing time to make simple fixes, or even reluctance by engineers to make changes. Tracking the volume and type of maintenance requests, timing estimates, and actual work length can help identify when the situation has gone from a mere nuisance to an engineering sinkhole.

It's important to remember that all software eventually becomes obsolete and it's only natural to forestall its replacement as long as possible. Unfortunately, the longer you wait, the more difficult it becomes. And without an immediate ROI or visible improvement, it's often very difficult to convince organizational stakeholders to invest in a migration effort. Tracking and reviewing metrics can help justify the cost of undergoing a software modernization effort.

It's very tempting to completely rewrite your legacy software to use modern techniques and revisit requirements – but there's a clear downside.

Paths for handling legacy software

Let's say you've identified that your aging software system is becoming too costly to maintain. Are you still actively selling it or just maintaining it for a handful of core customers? Are you under contractual obligations to maintain or extend it? Does it contain specially developed assets or complex algorithms that would be very costly to lose? How long can you afford to be without a new release or without new features?

After answering these questions, you'll be better prepared to choose among the options available.

A. Limp along

Although it's only feasible for some markets, customers, and competitors, remaining in a holding pattern may be an alternative that can keep costs to a minimum. Continue stringing on the old software as long as you can: incentivize employees who still know how it works to stay onboard, avoid adding new features whenever possible, and fix bugs only when absolutely necessary. The downside to this approach is that you keep accumulating technical debt, as well as becoming more vulnerable to employee turnover, changing technologies, and cybersecurity vulnerabilities.

B. Abandon

If it's not economical to maintain the old code base any more, you may be better off spending your resources on something else. Performing an end-of-life for a product may alienate some long-standing customers but it could be a trade-off that is the lesser of two evils. You may also want to consider replacement strategies for stranded customers – whether the replacements are in your product portfolio or outside it.

C. Rewrite

If there are enough issues with your legacy software, it can be very tempting to re-design and re-code everything from scratch, using modern development environments, modern libraries, modern coding techniques, and modern UI paradigms. Certainly this option is the most appealing from a developer's standpoint – a greenfield development will create a product that's not bound by technical decisions made decades ago. From the business side too, you get the chance to revisit requirements or limitations that don't fit with today's marketplace, so there is a clear upside for this choice.

Writing code is less work than testing and debugging it but the testing effort can be many times more expensive than the initial development.

Unfortunately there is a clear downside, too. Of all options, this leaves you without a working, sellable product for the longest time period. If your customers cannot easily consider other options, they might be able to wait for two years until the next release.

You can also consider renewing your software while you string along the old software at the same time. This will keep you on a regular release cadence at the cost of duplicating engineering effort and significantly increasing cost. This can often result in a highly motivated A-team working on new software while a B-team maintains the legacy system. Unfortunately, the latter is drudgery work that can be rather demoralizing, so your B-team may eventually leave – perhaps finding jobs in similar areas at competitive companies.

There is another less-obvious risk with a rewrite strategy. Your software is likely to contain modules that embody your domain knowledge with tricky areas that were hard to write and get right. Examples of this are complex algorithms, sophisticated data storage, import and export of external data formats, network protocols, and many more. Of course writing code is less work than testing and debugging it, and the testing effort can be many times more expensive than the initial development. Brand-new code very frequently lives through the same mistakes that were made when the original code was written 20 years ago, which can make your customers adopting the new suite feel like beta testers for buggy pre-release quality software.

D. Surgery

The last option is to be selective: keep the good and throw out the bad. Here you perform a careful surgery that keeps your most expensive or specially developed assets and focuses on replacing the out-dated codebase. To limit the scope and prevent the modernization effort from becoming a rewrite, you want to carefully select the smallest bits of code that are the most critical to replace. Usually the most out-dated portion is the user interface but this is also the time to identify any other areas that generate more than their fair share of trouble tickets, customer complaints, or engineer wrath. This strategy often works surprisingly well because the parts of your software that have stood the test of time – pieces that are still valid and useful – will often continue to hold up even in future versions with little or no changes. An example of this is a complex algorithm – once you've gotten it right once, you may not need to rework it at all. If you can manage to separate

The best way to handle your legacy software should be based on your goals, your team's collective expertise, your engineering load, your competitors, and your customers.

A surgical approach to migration is the most oft-selected path as it carefully balances risks and benefits

out these bedrock modules first, you can end up with less work, less cost, and a quicker time-to-market. Even if your software wasn't designed with perfect modularity that isolates the solid parts you wish to keep, it's often worth it to extract those core bits and rewrap them with a new interface or API to preserve their embodied knowledge.

The downside of a surgical approach is that you tend to need a higher than average skillset in your team – they need to understand both the old and the new. It might also be less appealing to your team to start by refactoring your software into what will be preserved and what will be removed. After all, the key skills of your team are not in forklifting ancient code with archaic libraries into a new toolkit but in your domain – industrial automation, oil and gas exploration, medical imaging, etc.


Making a migration

Your assessment of the best way to handle your legacy software is based on your goals, your team's collective expertise, your engineering load, your competitors, and your customers. At KDAB, we've seen all four of the possible options exercised by our customers at one point or another. However, we've discovered that the surgical approach (option D) is the most oft-selected path. Because it's not an all-or-nothing solution like some of the other choices, this approach allows a careful balancing of the risks, impact, and benefits. Option D is the heart of a migration effort and so the remainder of this whitepaper will assume that's the path you've chosen for your project.

What is migrating?

A code migration does not need to include a graphical UI and it need not require moving away from a defunct framework. As an example, we've worked with companies who wanted to move from Java to C++/Qt for performance and maintenance reasons. Although this guide is applicable to any type of large software project migration, we find C++ with Qt to be the predominant destination toolset in most of our migrations. And because the GUI tends to be the portion of the software that's most desperately in need of a refresh – many of our examples will focus on GUI changes.

We've examined many of our migration projects to analyze what factors make the most difference in migration efficiency. Here's what we've discovered.



Step 1) Develop an estimate

It's not easy to develop an estimate for any significant program but you'll need one to understand the scope of your migration project and to track progress against it. Migration estimates have a large plus compared to estimates on developing an application from scratch. That's because with a migration you already have the perfect "specification" at hand – the existing legacy application. Not only do you have a running model of the finished product, you have the complete source code repository, running your "prototype" user interface, user and training manuals, and any other assorted documentation that will help you understand the full scope of the project.

Step 2) Assess your skills

To turn your workload estimate into a schedule, you need to determine how much engineering firepower your team possesses. In other words, how long does it take them to complete a feature, including writing it, debugging it, documenting it, and writing unit tests for it. The logs of your source code repository can help you develop metrics for your team. You could also ask key employees to migrate smaller parts of your application. While doing these "prototype" ports, measure the time taken with ample notes of what was required. This will come in handy not only in developing an estimate for the migration but detailed notes may be useful for the entire team when planning the actual work.

A very common question we often encounter from clients is whether or not their engineering team has the skill to perform a successful migration. In general, for most teams the answer is yes – as long as your team has sufficient skills in the destination environment. Perhaps a better question is how efficient will your team be. We've examined many of our migration projects to analyze what factors make the most difference in migration efficiency. What we discovered was that there are three large categories of skills that matter for migration efficiency, in increasing order of importance:

A. Knowing your source framework

This means if (for example) you are migrating from MFC to Qt, you know MFC. If you can immediately say what each bit of code is doing without having to look it all up in a reference manual, you are going to be faster. However, knowledge of the source framework seems to matter least. No matter how well designed, code in large-scale applications tends to be repetitive. It's easier to read code than write it, and once you've looked up what a certain

We've seen clients overlook changes that were trivial to engineering staff – yet fundamental to customers – leaving them mired in customer-relations issues.



The biggest factor in a productive migration is how well your developers know migration techniques

method invocation does, you'll be able to recognize that particular pattern when you see it again as well as what construct you migrated it to. And even if you can't precisely remember, you've got the previously migrated snippet for reference. If you are doing well, you'll also have added support for your editor configuration to help with similar transformations in the future (see automation step).

B. Knowing your target framework

In our example MFC to Qt port, this means you know the target framework, Qt. Understandably, this is a skill you cannot do without. If you don't know the target framework, you will not be able to produce good quality code and you may not be able to recreate all of your system's original functionality. Better skills means better productivity so the more of the target framework you know, the faster you are going to migrate. And yet, we found that even this skill was not the most critical.

C. Knowing migration techniques

This turned out to be the biggest factor in achieving great productivity when performing a migration. In fact, it turned out to be a lot more important than knowing the target framework. Also, the productivity difference between somebody who had a lot of migration experience and somebody who did not was greater than the productivity difference between somebody who knew Qt well and somebody who did not.

Step 3) Note deliberate omissions

Before you start actual work on a migration, your team needs to come to an understanding about what you're trying to achieve. If one goal is to make an application that feels like a native app on the new target platform, it's better to identify any possible pitfalls early. For example, if your customers are used to old UI paradigms on the source platform that work differently on the new platform – like a scrollbar on Motif that changes to a slider on OSX – you'll find that cross-platform portability comes with some trade-offs. However, if you recognize ahead of time what won't be identical between your original application and the migrated version, you can mindfully determine the right course of action: ignore the minor differences, develop specialized controls, or update user manuals and training materials.

We verify code division decisions by trying to compile and link the code on what we call a hostile platform – one that does not support the old framework or library.

**There will
be code files
that crash
and burn
loudly. But
don't make
changes or
you risk
getting
confused**

It may seem obvious that changing platforms will inherently change an interface. However, we've seen clients overlook changes that were trivial to their engineering staff – yet fundamental to customers – leaving them mired in customer-relations issues after a migration release. A quick survey of the user interface and some amount of customer involvement beforehand can avoid this issue.

Step 4) Identify the divisions

The next step in your migration is to determine what code should be preserved and what should be replaced. Your team will likely know which parts depend on the old libraries or frameworks that you're trying to excise, and which parts are solely algorithmic using standard C or C++. But it's always a good idea to let a computer verify your assumptions.

At KDAB, we verify code division decisions by trying to compile and link the code on what we call a hostile platform – one that does not support the framework or library that you want to discard. For example, if the codebase is using MFC, we will try to build it on a Unix or OSX system; if it is using Motif, we will try to build it on Windows. How far removed the hostile platform should be will depend on your intentions. If you're using the migration to help extend your software to support multiple platforms, you should probably select a hostile platform that's as far away from the original as possible. Otherwise, uninstalling the framework (or hiding it from the build) may be sufficient.

Of course a build on a hostile platform will fail but the goal is to determine two things: how significantly does it fail and where does it fail. With any luck – or with a good dose of engineering foresight – there may be a few files that are platform-agnostic enough to compile successfully although there is likely not going to be a whole lot of them. Others will just have a few simple problems like different include files, standard library functions that have been moved into a different namespace, or the odd string handling function that has different names on Unix and Windows.

And then there will be the code files that just crash and burn loudly. Carefully examine the compiler errors and rate each file and module. Do not make any changes just yet, even if they seem trivial – like changing the name of an include file – otherwise you risk getting confused about your changes. If you want to confirm that a quick edit would fix an issue, test small changes but revert them once you see the results.

Many years of software industry experience have shown that so-called big-bang integrations are risky and likely to fail altogether.

Part of this analysis will also use text-searching tools to identify modules and files using the old framework by searching for characteristic substrings. For example, if you are migrating away from MFC, search for class names starting with a capital 'C', followed by another capital letter like CString, CDialog, CButton, etc. If you are migrating away from Motif, search for strings starting with 'Xm', or 'Xt', or 'X' followed by another capital letter. Recognize that identifying old framework dependencies by grepping isn't going to be perfect. You may have many false positives – for example, there could be other class names other than MFC ones that start with a capital 'C'. Or you may have subclasses that wrap your old toolkit classes, hiding a strong legacy dependency scattered throughout the code. In the end, it's your compiler that decides whether your code compiles, not your text-searching tool.

At the end of this step you will have a much better idea of which parts in your code will need a lot of intervention and which can remain largely unchanged. This should also give you an indication of the effort required to actually complete the migration.

Step 5) Make it build

At this point, you may feel ready to start the migration. You have a list of modules and files to work with so you can distribute work packages to your team. They will work away on their assigned tasks and in three months time everyone will get back together to integrate their work and create a shiny new version of your application.

Unfortunately, you would have to be extremely lucky for this approach to work. Many years of software industry experience have shown that so-called big-bang integrations are risky and likely to fail altogether. The agile paradigm of continuously making small improvements applies to migrations as well. You want to be able to integrate and test small changes as soon as possible. How can you possibly do that if 90% of your application doesn't build?

The answer is simple: make it build! Do whatever it takes to make your application build. Stub out the bits that do not build, temporarily remove modules from your make files if they are too large to just stub out, use the pre-processor to remove blocks of lines from the build – just keep commenting away until it builds. The secret is to do this in a controlled, reproducible, and reversible way because you have to know exactly which bits you have tem-

You need to establish a hierarchy in your code that identifies the fundamental pieces of your architecture and work towards completing those first.

porarily removed so they can go back in again after they have been properly migrated. You might not need to migrate all of that stubbed out code – most codebases on long evolved projects contain a fair amount of dead code, and a migration is the perfect opportunity to uncover and remove it.



Migrations of long evolved projects give developers the perfect opportunity to uncover and remove dead code

To make sure you're properly tracking migration-related comments, use a very distinct label, and not TODO. Although it may be a bit extreme, something like `I_AM_STUBBING_THIS_OUT_BUT_IT_NEEDS_TO_GO_BACK_IN_LATER` is a far better choice. Why? Chances are that your code is already littered with TODOs and you need to be able to clearly distinguish migration stubbing in the source. Don't worry about making your code look ugly; these will all be removed once you are done!

Once you've gotten things to build cleanly, you may have nothing left but a 'main()' function – and possibly even that will contain stubbed out code. That is fine at this point; you are going to bring back things quickly in the next step.

Step 6) Migrate the core

Now that you've gotten things to build, you may be tempted once more to think now is a good time to distribute the work across the team and integrate it as soon as possible. Not just yet! We need to be selective. To show why, let's assume you are migrating a CAD application that allows you to create technical drawings. It's not very helpful to work on a 'Preferences' dialog that lets the user configure how the drawing is displayed because you can't create a drawing yet. Ideally you'd be able to load a well-defined demo file on which to test the 'Preferences' dialog – but the loading module won't be done either.

What you need to do is establish a hierarchy in your code that identifies the key features of your application that are fundamental to everything else and work towards completing those pieces. In our CAD example, that could be opening, saving, and to some degree modifying the document. Core functionality will be dependent on some low-level modules that you'll absolutely need – such as string or container handling – and these will need to be worked on first.

You don't want over-optimistic schedule guesses to disappoint waiting customers or put pressure on your engineering team that results in release-unready software.

Once you have migrated the core features and their dependencies, chances are that you will have a lot fewer dependencies in the rest of the code. That's the point when you can farm out work in parallel.

Step 7) Track your status

Now that your team is happily working away on your migration, completing a few modules every week, integrating and testing as they go, how do you know if you're on track? You don't want over-optimistic schedule guesses to disappoint all those waiting customers or, worse yet, put pressure on your engineering team that results in release-unready software.

We know from experience that providing detailed, realistic numbers is something that instils confidence in your progress. At KDAB, we rely on tools and algorithms we've developed for migration status tracking that automatically harvest the code base and generate status report spreadsheets and nice burn-down charts. Even if you don't have a sophisticated tool, you can manually assess your progress – by periodically sampling the commits or grepping for migration comments. How you execute your tracking may be dependent on your code base.

Whatever approach you use we strongly advise you to come up with a way to track your progress. And the more you automate these progress reports, the more likely it is that you will actually use them. While you develop your measurement procedure though, a word of warning: it is easy to forget about the long tail of migration. Your development speed is going to dramatically slow down towards the end of the project when parallelization becomes less possible and only bug-fixing tasks remain. Don't assume that you'll be able to maintain the same pace even when there's just a few items left to address – those may be among the most difficult to fix.

Step 8) Create a test plan

Whether you perform a migration yourself or choose to outsource it, a test plan is an asset of tremendous value. It does not necessarily require automation – it could simply be a document with steps and expected results. (And something simple like this may be the best approach at the beginning stages of a GUI migration.) Even if you do not have a test plan, you might have something that you can turn into one. For example, you may have a user's manual. Can you perform everything the user's manual describes with

We often get asked if we could create a tool that automates the entire migration but we believe that human engineers are still the best qualified to do software migrations.

your new version? While you are checking that, write down the steps that you take to verify and voilà, there's your new test plan. (As a next step, you can turn this into an automated test suite.)

Assuming your test plan is complete, it allows you to check your migration feature set, gives you another means of tracking progress, and tells you when you are done. If you're confident that your test plan is complete and you can perform all tests with the expected outcome, then congratulations – you've completed your migration!

As a confirmation that you're finished, look for any remaining code that is still stubbed out. Its presence indicates either that your test plan is not as complete as you thought it was (and you still have a bit of work to do), or that this particular code is not needed any longer. Double-check, ask for a review, and, if the code really isn't needed, go ahead and remove it.

Step 9) Automate what you can

We often get asked at KDAB if we could create a tool that automates the entire migration. That thought has often occurred to us but unfortunately it's an idea that's doomed to fail because no matter how sophisticated the tool, it would be primarily driven by mechanical text substitutions. Understanding the intent behind the code is often necessary to make a successful transformation. Even if you were able to migrate half of the software automatically – an amazing achievement – you'd be left with a number of difficult-to-find bugs in the auto-translated portion, not to mention the remaining half that needs to be cleaned up. We believe that today human engineers are still best qualified to do software migrations.

This isn't to say that there isn't room for software tools in a migration. Let people migrate the software line-by-line but give them all the tooling help you can. At KDAB, we've developed countless internal migration tools over the years that allow our engineers to be as productive as possible. For example, in a migration from Motif to Qt, code like the following would likely occur rather often:

```
XmToggleButtonGadgetSetState(_onoff, sc && sc->isEnabled(), false);
```

This can be automatically migrated to

```
ui->_onoff->setChecked(sc && sc->isEnabled());
```

It's natural to think about making other desirable changes while you're moving software into a new framework but you should avoid the temptation.

While modern IDEs easily locate all references, many don't function unless the source builds properly

It's not desirable for an engineer to type in all the input parameters each and every time – it's time consuming and error-prone. However, an editor that's configured with lots of intelligent macros to automate these types of transformations can be of tremendous help. Once the engineer identifies the situation, she can trigger the code transformation and review the outcome instead of having to manually look up parameter order, and copy and paste the separate bits.

Another example of tooling help is switching versions between the original unchanged application and the new one under migration. We typically keep the two code trees next to each other, and use KDAB-designed tools to allow an engineer to quickly jump between the two versions of the same file in order to check on what any given piece of code looked like in the old version, open another window with the diffs, and so on. Search tools are also an indispensable part of a migration engineer's toolbox. As an example – it's a great help for an engineer to place the cursor over an identifier and hit a key combination to instantly locate its definition or any references throughout the source tree. While most modern IDEs offer these capabilities, many don't function unless the source builds properly – which makes them useless for migration work. Text-based searches using complex regular expressions can provide many other ways to make pointed analysis even on poorly-formed source code.

Step 10) Resist feature creep

It's natural to think about making other desirable changes while you're moving the software into its new framework. You're already getting your hands dirty in the code – why not add a new oft-requested feature, refactor some ill-formed code, or face-lift the UI to make it look like an iPhone?

Of course you could do any of these things but in our experience you should avoid the temptation. Replacing one UI framework with another or bringing an application from one operating system to multiple ones – or doing both – is a major undertaking like a heart transplant. No surgeon would fix a broken bone while performing a heart transplant because there's a great risk of messing up one or the other operation, or not completing either properly. Software migration is just the same – make all the changes you want but do them sequentially with complete and thorough tests in between. The elapsed time will be shorter and the risk will be a far less.

Although many details can only be learned by experience, if you should have to perform more than one migration, you'll be much better at it on the second project.

It can be a hard sell to management to spend a lot of money on a migration that has no visible improvements. Still, you should fight that fight to safeguard the future. Explain that you have accumulated technical debt that needs to be paid off now or you'll have to pay off more of it later because of the accumulated interest. The more technical debt accumulates, the harder it will be to make changes of any kind without breaking something else. Feature releases and bug fixes will take more time to implement, creating delays, stress, and reduced customer satisfaction with a continually increasing time-to-market. If you are coming to Qt from a single-platform toolkit such as MFC or Motif, there is one big new feature you're adding that's inherent in the migration effort itself, which is availability on multiple platforms.

Similarly, you usually shouldn't bother cleaning up the code before migration. Logically, the cleaner your code base, the faster and more cost-efficient your migration will be. However, if you refactor and restructure it beforehand, you won't have a defined standard baseline to work from. This is true even if the majority of work is done by a third-party like KDAB. If you have a test suite that guarantees the cleaned code meets all tests, you might consider doing a bit of clean-up work first. But if you're not prepared to make an official release for the cleaned version (even if it's internal), don't bother – you always want to have a sanitized starting point. Mitigate your risk by serializing tasks; you can always refactor after the migration.

Conclusion

We have given you some steps and strategies on how to plan, prepare, and execute a migration. Although many of the details can only be learned by experience, if you should have to perform more than one migration, you'll be much better at it on the second (or subsequent) projects. If you have any questions about your migration that you'd like to discuss with an expert, please feel free to contact us at info@kdab.com.

If you have any questions about your migration that you'd like to discuss with an expert, please feel free to contact us at info@kdab.com.



Matthias Kalle Dalheimer

Kalle is the President and CEO of KDAB. He has actively developed with Qt since 1996 and is a founding member of the KDE project. He has written numerous books, both in English and in his native German, including "Running Linux" and "Programming with Qt". Kalle holds an MS in Computer Science and has taught more than thirty Qt classes for companies like Ericsson, Motorola, and J.D. Edwards. In his spare time, he enjoys orienteering, cross-country skiing, and reading history books.



www.kdab.com

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt. Our experts build run-times, mix native and web technologies, and solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages. Founded in 1999, KDAB has offices throughout North America and Europe.