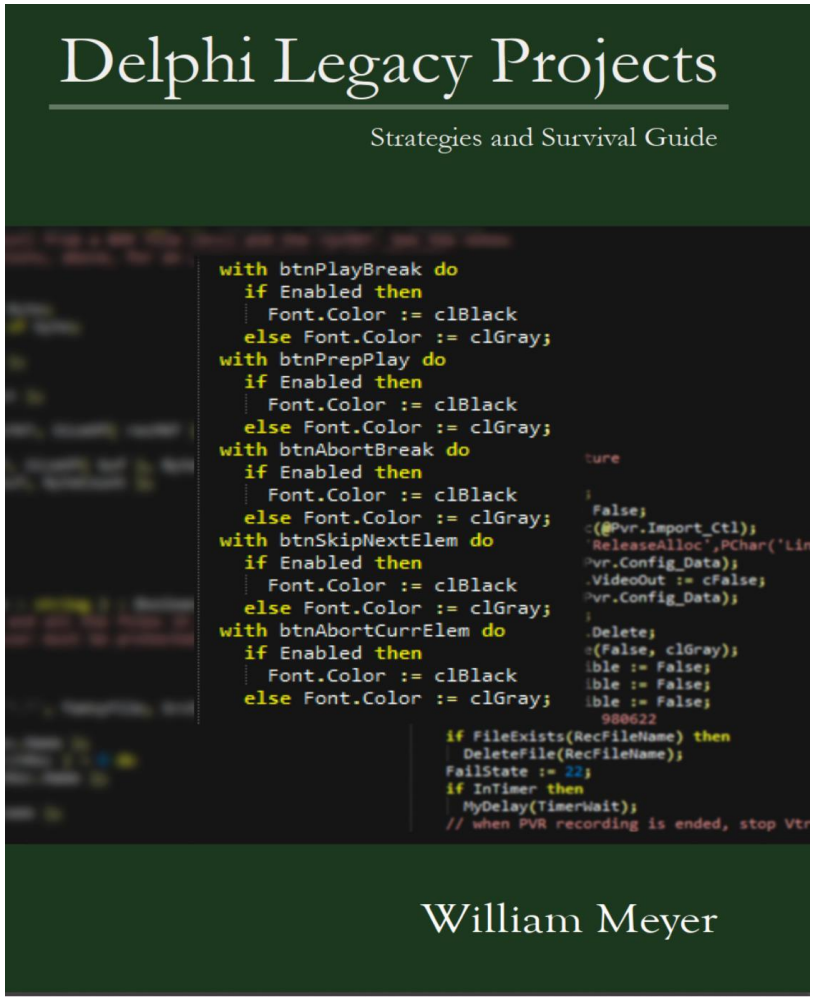
Upgrading and Maintaining Delphi Legacy Projects

Some approaches to the challenge, with reference to my book:

Delphi Legacy Projects

Strategies and Survival Guide



Where to begin?

- Legacy projects tend to be large.
- Finding a starting point can seem difficult.
- All projects are different, yet many share common problems.
- Any Delphi version can be involved; I know of current products which are still based in Delphi 6. Some may be still older.
- Patience and perseverance are essential!
- I will not offer much code here. I can’t use proprietary code, and writing meaningful bad code is an unpleasant task!

Mixed Projects - 1

- Often a source tree may contain multiple projects.
- Shared modules, other than libraries, increase difficulty.
- Even libraries may be problematic.
- Bottom line:
  Create a new folder tree for your target project. Put in it all the essential files, but no more.
  It must contain only one project!

Mixed Projects - 2

- Isolating a single project minimizes the problems.
- Your work will introduce incompatibilities with the other projects.
- There are solutions to updating those projects, as well.
- Create a new folder tree:

  NewSource

    Libraries

    Project1

Mixed Projects - 3

- Your NewSource folder is now the focus.
- Project1 is your main concern in this work
- The Libraries folder is secondary, but essential.
- Libraries must comply with stringent rules, if you will avoid creating future problems.
- Do not think about how your work in Project1 may affect other projects; in this separated tree, it will not.
- New releases of this project will be from tree. Keep it clean!!

Library Modules

- Library modules must be rock solid and clean.
- “Clean” means:
  - They must not create issues for other modules.
  - They must not use modules from Project1.
  - They must be as usable in isolation as any Delphi library.
- These goals will not be as easily met as you may think.
- If your library modules are not “clean” in this sense, keep them in Project1 for now.

Long Build Times

- At some point, large projects may begin to suffer long build times.
- These build times are the result of Unit Dependency Cycles (UDCs).
- A UDC is created when UnitA uses UnitB, which uses UnitA.
- In a large project with many UDCs, once the build times begin to increase, the problem will multiply.
- UDC effect on builds is an exponential burden. The exponent is small, but any exponent will eventually become a burden.

Reducing the Size of the Problem - 1

- Legacy projects did not become problematic overnight, neither will they be cured overnight.
- The first step in reducing the *apparent* complexity is in creating the NewSource folder tree.
- In an isolated tree, the effect of any work you do is limited to that tree; you need not worry about breaking other projects.
- Be incremental; be pragmatic.
- If you have not read *Refactoring*, by Martin Fowler, do that. Now.

Reducing the Size of the Problem - 2

- Once you begin, any strategy you devised will change:
  One of the big dangers is to pretend that you can follow a predictable process when you can't. – Martin Fowler
- Leave any file you edit better than you found it.
- Keep scope as narrow as possible.
- Remove global variables, where possible.
- Global types and global constants are not problematic.

Challenges of Testing - 1

- If your project includes unit testing, that’s great!
- Many legacy projects can’t really be unit tested.
- To achieve testability, your project will need refactoring.
- Some barriers to unit testing:
  - Code on forms
  - Excess complexity of routines
  - Unfocused code

Challenges of Testing - 2

- Q: Which test framework is best?
  A: The one you will use.
- Code on forms as little as possible.
  - Keep event handlers short.
  - Factor out business logic from forms into separate units.
  - Factor out data manipulation from forms into data modules.
- When business logic is written to be in a separate module, it is written in a more testable way.
- Testability is not achieved overnight.

Unit Dependency Cycles

- The bad news of UDCs:
  - They exist because of design errors.
  - They can be reduced only through design changes.
  - Once UDCs are present, they metastasize.
  - No tool will reveal which modules cause the problems.
- The good news of UDCs:
  - They yield to persistence.
  - Sometimes a small change brings a large benefit.
- Like all your work in legacy projects, UDCs will not be resolved quickly.

Helpful Tools

- Tools I consider essential:
  - CnPack
  - GExperts
  - FixInsight
  - MMX
- The order here is alphabetic, not by value.
- Only FixInsight is commercial, the others are free.
- You may also wish to consider:
  - Pascal Analyzer, by Peganza

No Silver Bullet

- Difficult tasks:
- Demoting uses references from **interface** to **implementation**. Only Pascal Analyzer helps you identify those.
- Upgrading any non-trivial legacy project will always be iterative.

Separation of Concerns - 1

- A primary example of mixing concerns would be a Delphi form which contains:
  - Business logic
  - Datasets and data manipulation
  - Large and confusing routines
- A better approach to separation of concerns would make use of:
  - A UI form
  - A business logic unit
  - A datamodule
- And each of these may well call routines from Delphi library modules, and from your own libraries.

Separation of Concerns - 2

“If you have to spend effort looking at a fragment of code and figuring out what it’s doing, then you should extract it into a function and name the function after the “what”.” — Martin Fowler

- Excellent advice, and it leads logically to extraction of nested routines, as a first approximation.
- Often in large routines, you will find blocks of code repeated, which also favor the use of nested routines.
- As you proceed, you may find nested routines which appear in multiple larger routines, as they were probably copy and paste. These should become private methods of the class.

Incremental Attack - 1

- Any large project will yield to incremental improvements.
- Legacy projects which cannot be unit tested present challenges to improvement.
  - How to make changes without introducing defects?
  - How to reach a unit testable level of code.
- Redesign and rewrite is massively risky.
- Refactoring:
  - Offers a much less risky approach to cleaning up code.
  - Is probably faster than a rewrite from scratch.
  - Avoids the all or nothing approach of replacing an entire module.

Incremental Attack - 2

- Process outline:

1. Create separate modules for business logic and data.
2. Factor out nested routines.
3. Collect similar nested routines and replace with methods.
4. Extract private methods from forms and move to business logic.
5. Extract datasets to datamodule.
6. Extract data manipulation to datamodule.

Incremental Attack - 3

- Iterate over your new modules, improving, cleaning, renaming, making them great examples of the code you wish you had already.
- Small steps will win: Always leave a file better than you found it.

Refactoring - 1

"If you're afraid to change something it is clearly poorly designed." — Martin Fowler

- Refactoring is at the core of legacy project rework.
- You will inevitably improve your refactoring skills in updating legacy projects.
- Refactoring is not a step, but an endless process.

"In almost all cases, I’m opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts." — Martin Fowler

Refactoring - 2

- Let’s look at a very small code example:

```
try
  n := StrToInt(AString);          TryStrToInt(AString, n);
except
end;
```

These two snippets are functionally equivalent. The code on the left has been seen in numerous legacy projects. We can argue that it does no harm, but is certainly bad practice.

One issue is the empty except clause. Another is the pointless use of try/except where there is no need. Another solution could involve pre-checking the string, but the TryStrToInt is safe and effective.

Refactoring - 3

- As developers, we all have different points of focus. Sometimes a developer is focused only on his area of responsibility, and doesn’t explore new releases of Delphi.
- Simplify your life; study new releases, especially the libraries. In some there have been enormous changes. These can simplify your code.
- Your own libraries may be improved by study of Delphi libraries. Some of your old routines may be replaced by functions shipped with Delphi. Others can be made cleaner and simpler using library calls to replace things coded by hand years ago.

Coherent Component Use

- When we first used Delphi, many of us were excited to get new components, and (over)eager to use them.
- Legacy projects may often include a disordered array of components which create a somewhat inconsistent appearance.
- Consolidating components will likely help make code more consistent.
- Consolidating your primary components may also help reduce your annual license fees.

Using Tools with Delphi - 1

- There are tools which will enhance the productivity of the Delphi IDE.
- Those already mentioned which ought to be considered essential:
  - CnPack https://cnpack.org
  - FixInsight https://www.tmssoftware.com/site/fixinsight.asp
  - Gexperts https://blog.dummzeuch.de/experimental-gexperts-version/
  - MMX https://www.mmx-delphi.de/

Using Tools with Delphi - 2

- Custom tools. You can certainly build your own.
- Reasons to do so:
  - None of the tools you have found provide the operations you need.
  - The operations you need are relatively simple.
  - Building the tools will be a learning experience.
- Reasons not to:
  - You lack the time to do so.
  - The operations of interest are too complex.
  - You can’t estimate the risk or development time.

Summary

- This presentation cannot begin to present the details of legacy project rework and the methods and strategies needed.
- If you need to update one or more legacy projects, I suggest you read my book on the subject, which covers much more ground.
- Paperback: https://smile.amazon.com/Delphi-Legacy-Projects-Strategies-Survival/dp/B0B2TY6ZZ4
- PDF: https://wmeyer.tech/books/

Books

- If you have not read Martin Fowler's book: *Refactoring: Improving the Design of Existing Code*
  Do so. The examples are not in Delphi, but the value of the book is in the mindset and processes.
  https://smile.amazon.com/Refactoring-Improving-Existing-Addison-Wesley-Signature/dp/0134757599/
- Also useful is *Working Effectively with Legacy Code* by Michael Feathers
  https://smile.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052

Webinar Discount

From today through 6 November 2022, you can purchase the PDF from of *Delphi Legacy Projects* at a discount.

https://www.wmeyer.tech/books/

Coupon code: DLP-WEB