Article from

The Modeling Platform

November 2020

The Importance of Centralization of Actuarial Modeling Functions, Part 4

DevOps and Automated Model Governance

By Bryon Robidoux

This is the fourth and last article dedicated to providing guidance and a road map for centralizing modeling within the organization. This series shows how simple overlooked behaviors, which appear harmless at the lowest level of the corporation, are causing tons of organizational complexity, time and money when aggregated across the organization.

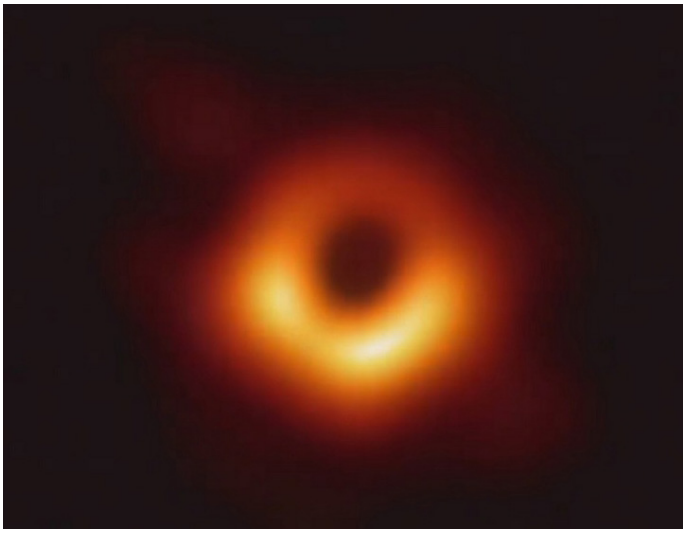
In Part 1 of this article, software engineering principles were used to show that decentralizing models comes with extremely high cost. It showed that centralization of a modeling department is a step in the correct direction, but it is not enough. The key to running a smaller, better, faster and cheaper modeling department is to focus on modularity and work-product reuse according to software engineering principles. Part 2 introduced the reader to the major components of DevOps and how it is the basis for actuarial modernization. Part 3 explained how to build a data-driven Axis model for the most amount of data reuse and automation possible. Lastly, this article addresses how to use DevOps with the Formula Table code within Axis to increase the quality of the models and the throughput of new enhancements to help overcome the monolithic-system problem. It will do so by focusing on the tools used to implement a full stack of DevOps for Axis model code.

Even though specific tools are mentioned in this article, it is not an endorsement. These are tools that I have used in the past and am familiar with. It also makes the explanation less abstract to use actual tools as examples. It is highly recommended that readers research other tools before jumping on board with the tools mentioned. Your IT department is probably already using these types of tools and should be consulted.

PUTTING IT ALL TOGETHER

Let us step back and look at the big picture on what these articles have been trying to accomplish. A colleague explained that there are two types of models: thick and thin. A thick model is when all the work is performed and stored inside the model. A thin model is created when all work is performed and stored external to the model. Furthermore, only at the last possible minute before runtime is the model assembled and executed.

The thick model is the root cause of the monolithic-system problem. These previous articles explained that current actuarial modeling practices create models that are as thick and dense as the Messier 87 black hole.



Event Horizon Telescope Collaboration. Photo courtesy *NASA/JPL-Caltech*.

The real goal should be to create models that are razor thin. Thin models promote building reusable components so that the organization can achieve economies of scale by maximizing work-product reuse and using Agile project management. Therefore, they promote the consolidation of the modeling function.

Thin models are important because they allow actuaries to use the best tools for the job so they can efficiently build robust processes and models. They allow the modeling platform vendors to stay concentrated on building software, where they have a competitive advantage. Modeling vendors should strive to make third-party DevOps tools as easy as possible for actuaries to use so that their work products seamlessly plug into the

IT infrastructure. This will give actuaries and IT the ability to work harmoniously together to achieve new levels of efficiency. This will promote the entire organization to implement continuous testing, integration, development, deployment and other DevOps practices. An organization that could pull this off would dominate the industry because it could make more informed decisions and execute faster.

AXIS BACKGROUND

Non-Axis users may need a frame of reference for its two major components: E-Link and the dataset. E-Link's main goal is to manage the collection of the organization's models and orchestrate their execution. It has a very Windows Explorer feel. E-Link can be automated with scripts to externally manipulate datasets and customize orchestration using Axis Jobs and E-Link scripts, respectively. One of the most important enhancements to E-Link in the past three years or so is Formula Link. This extension allows users to create reusable libraries that can be shared among multiple models and E-Link scripts.

Formula Link was a necessary enhancement that allows the outside world's libraries to be referenced from within the Axis world and shared among all datasets. I highly prefer Axis because all its customization uses the Microsoft .NET language. This opens a whole new world of possibilities because a plethora of DevOps tools become immediately available once the code is extracted and then referenced through Formula Link. With a little creativity and planning, it is possible to make a thin model.

There are two types of custom code in an Axis: code that heavily interfaces with Axis and mostly stand-alone calculation code. The former should stay in code snippets in Formula Link for maximum reuse and is beyond the scope of this article. The latter is where this article is targeting because it can be transformed into external reusable libraries. These libraries can use the full stack of DevOps tools that IT uses.

CODE EXTRACTION

The first step of thinning the model is externalizing all the formula table code to a Dynamic Link Library (DLL) with an integrated development environment (IDE), such as Visual Studio, outside of the model and Moody's environment. DLLs are no harder to write than a code snippet in the Axis dataset. Given that Axis uses VB.NET as its preferred language, the formula tables can be moved over to Visual Studio library solution with ease. It all depends on how much the calculations are tied to functions on the Input, Output and Common tabs of the formula table in Axis.

The code remaining in the formula table should be only what is defined as pump-and-dump code. It should be the minimal amount of code possible to pump data out of the formula table input variables, shove into the external library or libraries and dump back into Axis output variables. The less code that exists in the formula tables, the more automated unit testing and user acceptance tests (UAT) that can be performed and the more stable the model will be to change. (For a more detailed article on how to clean up formula tables while data is being extracted, read "Building a Modularized and Reusable Formula Table Code in Moody's Axis Using Formula Link.")

SETTING UP THE IDE

Each Visual Studio solution should be divided into two projects. One is for the API library that Axis will call. The other is a unit-testing project that references a unit-testing framework such as NUnit. NUnit is available to all developers through Visual Studio's NuGet Package Manager. NuGet Package Manager allows users to easily manage all their references to both internal and external libraries. Within the unit-testing framework, there should exist both unit test and UAT to facilitate automation for continuous testing.

CONTINUOUS TESTING

NUnit allows the developer to write automated tests that can be run in and out of the IDE all external of Axis; once the code is in the Axis model, it becomes much harder and time consuming to test and find problems. By testing in the IDE using Axis, the developer can get feedback in milliseconds instead of minutes, hours or days.

Actuaries should strive to perform test-driven development (TDD). TDD requires that the developers, testing team and stakeholders supply data and tests that the developer must pass **before** the code can be developed for and released to the Axis model. The test becomes part of the design process at the beginning of the project instead of an afterthought on the back side after development is complete. This greatly speeds up the development cycle because stakeholders cannot produce a list of impossible requirements. They must provide the tests for validating requirements, which leads to a conversation about feasibility. Further, this forces the model design to be modular, so that all functionality can be easily tested. This mode of working works nicely within an Agile project management framework. Once the testing team receives the library, it can focus on integration testing to make sure the model and libraries are working together properly.

To better perform model life cycle practices and testing, it is recommended to use SpecFlow with NUnit for behavior-driven development (BDD). BDD aims to create a shared understanding of how an application should behave by discovering new features based on concrete examples. Key examples are then formalized with natural language, called Gherkin, following a given/when/then structure. SpecFlow helps teams bind automation to feature files and share the resulting examples as living documentation across the team and stakeholders. To produce nice-looking testing documentation, Pickles can be used along with SpecFlow. Pickles is a living documentation generator: it takes your specifications (written in Gherkin, with Markdown descriptions) and turns them into an always up-to-date docu-

mentation of the current state of your model or software and in a variety of formats. Results produced by Pickles become documentation and communication to auditors, controllers and validators on how each unit of the model must behave.

REFACTORING

Now that testing is set up, it is time to clean up the code! JetBrains Resharper is a Visual Studio plug-in for refactoring code and making it easier to read, abstract and organize. Refactoring should never be done as a separate project. It should be done every time the code is touched. Now that the unit tests are available, the developer can move around code and change the model without worrying about changing results. Code and models are just like bushes: They need to be constantly pruned and maintained in order stay looking their best. Otherwise, they will get unruly and it will require a large job to get back in order.

AUTOMATED STANDARDS ENFORCEMENT

All aspects of the model should have standards that are followed. Standards enforcement and code review are very manual and tedious processes inside the Axis model. Visual Studio has another plug-in called SonarLint, which statically analyzes code for standards compliance and coding styles that will potentially cause bugs. It will enforce that the actuaries' code is written to the corporate IT standards. This plug-in boils a one- or two-week code review process down to one minute! This also makes sure that standards are uniformly applied across the organization. This reduces the slower manual standards enforcement processes to the bigger-picture architecture issues and avoids manual standards on the high-velocity minutiae.

DOCUMENTATION

Now that the code is better, cleaner, tested and up to standards, it is important to document it. This is where a plug-in like VSdocman will come in handy. In Visual Studio, if you use three comment characters in a row, it will generate XML tags categorized by common types of documentation. VSdocman will use these XML tags to generate professional-looking documentation that resembles Microsoft's code documentation. There is even a switch to allow the actuary to include the code with the documentation, so the library calculations are completely transparent. (These documentation XML tags are also available in Formula Link, but there is no utility like VSdocman within Axis to export them, unfortunately.) VSdocman has a stand-alone application that accepts command-line parameters, so it can be called independently of Visual Studio to generate documentation and export to a wiki.

Now that the code is cleaned up and documented, it needs to be version controlled.

VERSION CONTROL

Git and GitHub were created so open-source developers could collaborate on writing code, regardless of location. GitHub is the graphical user interface that sits on top of Git to make it more user friendly. Git handles the versioning and pushing and pulling changes to the server. It has everything an actuary could want for controlling code and tracking changes in repositories. All modern IDEs will have plug-ins to make Git easy to use for the most common tasks. Git has built-in model steward functionality, called a pull request, to sign off and approve changes to a development branch before it can be merged to the master branch. The changes can be annotated so everyone can get a clear understanding of their purpose.

TRACKING WORK

JIRA allows all members of the modeling team to track the progress of a project and its development tasks. They have many canned reports, which makes adopting Agile project management much easier. The actuary can put any files, comments, decisions or other information that are relevant to the task into a ticket. This ties together the evolution of the code with the evolution of the task that created it. It is very handy to go back and look at the JIRA to find all the details on why a set of changes occurred in the code and how ambiguities in requirements were resolved.

CONTINUOUS INTEGRATION

Now it is time to integrate the DLL library with the model. All the Visual Studio plug-ins I explained earlier, except Resharper, can be executed in a server environment in a DevOps pipeline, such as Jenkins Pipeline. Jenkins Pipeline—with the execution of a script—will:

- download the library's repository from GitHub,
- compile it,
- version it,
- execute the review with SonarLint's companion SonarQube, and
- run all unit tests and UAT.

These are the same unit tests built into the Visual Studio project. Once the code passes all the automated review and automated tests, the pipeline will:

- generate the documentation with VSdocman and update the wiki,
- store the DLL in a work product repository like Artifactory, which in turn makes them available in NuGet Package Manager,
- move the DLL from the insurance organization to the Moody's environment,
- use an E-Link script to import the DLL into a Formula Link library, and
- use E-Link to import the Formula Link library into a dataset.

This will make using the external DLL library as seamless as if it were a code snippet created in Axis. To link external libraries through Formula Link, there is a little setup required.

1. The DLL and its dependencies must be stored within a folder within the Formula Link library.

2. The AssemblyInfoAndReferences file inside the Formula Link library must be modified with the following commented line *'REFERENCE_DST folder\your.DLL*, where

   REFERENCE_DST is an Axis keyword,
   folder is the name of the folder in the Formula Link library, and
   your.DLL is the name of DLL to be called from the Formula Link library.

The *AssemblyInfoAndReference* file is what makes this article possible. It is the most valuable feature of Formula Link!

CONTINUOUS DEPLOYMENT

When deploying changes, there needs to be:

- user-based privileges in Jenkins,
- landing locations in the Moody's environment and
- Formula Link libraries for each development, QA and production environment.

For example, if a developer built a development branch of a library in Jenkins, it would land in a development folder in the Moody's environment, be loaded into the development Formula Link library and be loaded to a development dataset. A tester would have the same process, but the library would be moved to the equivalent QA instances. Once QA is finished, only the head model steward can approve the pull request into the master GitHub branch and build the master branch on Jenkins. The library would then land in the production folder in the Moody's environment, be placed in the production Formula Link library and be loaded into the master dataset. There needs to be a Formula Link library for each environment; otherwise, all the development, QA and production changes would be stacked on top of each other. This is annoying to maintain and does not scale well.

CONCLUSION

It is important to externalize the code and build libraries to eliminate the monolithic-system problem. But once there is an effort to do this, as the article demonstrates, the massive quantity of software engineering tools at the actuary's disposal will make development much easier by automating unit tests, UAT, refactoring, documentation, enforcement of standards, integration and deployment of code into the Axis dataset. All these enhancements will speed up throughput of model features, immensely improve model governance and make the development way more agile.

The third-party tools mentioned in this article are used by millions of developers, so they are robust and easy to use. They are constantly enhanced to improve the efficiency of developers. It is important that actuaries have the same access to these tools to make them as efficient as possible. Somewhere, somehow and someway, actuaries have diverged from using software engineering tools. It is imperative that we close this gap sooner rather than later to manage the changes instigated by competition and regulation. ■

Bryon Robidoux, FSA, CERA, is an actuary at The Standard. He can be reached at *bryon.robidoux@standard.com.*

TECHNOLOGY WEBSITES

Artifactory. *https://jfrog.com/artifactory/*.

Confluence. *https://www.atlassian.com/software/confluence*.

Git. *https://git-scm.com/*.

GitHub. *https://github.com/*.

Jenkins Pipelines. *https://jenkins.io/doc/book/pipeline/*.

JIRA. *https://www.atlassian.com/software/jira*.

NuGet Package Manager. *https://www.nuget.org/*.

NUnit. *https://nunit.org/*.

Pickles. *http://www.picklesdoc.com/*.

Resharper. *https://www.jetbrains.com/resharper/?fromMenu*.

SonarLint. *https://www.sonarlint.org/*.

SonarQube. *https://www.sonarqube.org/*.

SpecFlow. *https://specflow.org/*.

Visual Studio. *https://visualstudio.microsoft.com/free-developer-offers/*.

VSdocman. *https://www.helixoft.com/vsdocman/overview.html*.