Article from

The Modeling Platform

November 2018
Issue 8

Applying FinTech and IT Practices to Building Actuarial Models

By Igor Nikitin and M. Crew Sullivan

My name is Igor, and I have held a variety of business and technical roles as an actuary. An acquaintance approached me with an idea of starting a FinTech company. Impressed with his enthusiasm and knowledge, coupled with my own curiosity and desire to obtain experience in lauching a startup, I decided to join the company as a part-time technology co-founder. At the time, I felt confident in my programming knowledge.

The most shocking thing that I quickly learned in a startup is best described by a quote from the Game of Thrones*: "You know nothing, Jon Snow."*

- The culture was very different, until I realized that it must be.
- The cost constraints looked impossible, until I learned that they were not.
- The problems I had to solve were all new and uncomfortable, until I got used to them.
- And my awesome programming . . . Well, I fired myself from technology lead for . . . knowing nothing.

At the same time, I started to learn furiously the knowledge I discovered I lacked, which seemed to be literally everything at first! I also quickly noticed a great amount of synergy between my work in a startup and my work at both jobs. Here is my story of applying knowledge from a FinTech startup to modeling in a big insurance company.

IN-HOUSE MODEL OR VENDOR-BUILT?

Actuaries need models to do their work. Models are just software, which can be bought from a vendor or built in-house. For most companies, vendor systems offer a positive user experience at a reasonable price; but for some companies, in-house systems are the only way to satisfy business needs.

Common reasons for requiring an in-house model are innovation and speed of modification.

- No vendor would develop software for products that don't exist in the marketplace. Only the company innovator knows what these products are and can develop its platform accordingly.
- Innovation in the institutional space is often deal-driven, and includes the risk of being unable to transact due to waiting for vendor modifications of the modeling platform. In-house software is faster to change.

On the other hand, an in-house platform can have its own drawbacks and dangers, such as high cost of original build, scalability, transparency, build quality and maintainability. My perspective is that of a modeling actuary in a highly innovative company that opted for an in-house platform. In this article, my colleague Crew Sullivan and I will detail how to build an in-house modeling platform using a mix of technologies and processes to overcome typical drawbacks of an in-house system.

PROCESS STRUCTURE

Here are the steps we followed to build a successful in-house modeling platform:

1. Study the past. Why do we do modeling platform conversions?
2. Do we have necessary resources?
   a. Right skill sets
   b. Time
3. Do we have management buy-in and commitment?
4. Design phase (aka trade-offs, trade-offs, trade-offs)
   a. Design goals
   b. Object-oriented vs. procedural programming
   c. Software design principles
   d. Design patterns
   e. System blueprint (UML)
   f. Language choice
5. Execution phase
   a. Style guide
   b. XML doc
   c. Version control system (Git)
   d. Input structure
   e. Error handling
   f. Build order
   g. Unit testing framework
   h. Optimization
6. Testing phase
   a. Automate your regression test process
   b. Build a quality test bank
7. Maintenance phase
   a. Maintain UML
   b. Don't make a mummy! If you need surgery, don't use a bandage.
   c. Maintain documentation

STUDY THE PAST: WHY DO WE DO MODELING PLATFORM CONVERSIONS?

Companies switch modeling platforms for a number of reasons, which may include:

- **Scalability.** Excel models tend to run into this limitation, and the story typically goes like this: "My original platform priced my first contract in a day of runtime, and it was awesome. But now I have dozens of contracts in our pricing pipeline and a growing valuation block. My existing runtime is unacceptable, and I have no way of scaling it."

- **Flexibility.** "I came up with an awesome new product feature that a client wants, but there is no way to model this in my modeling platform. Modification will take a long time and/or will be very expensive to develop." Closed vendor systems tend to suffer from this the most.

- **Transparency.** There are two equally important flavors of this: user transparency and developer transparency. If users can't see and easily control calculations in an innovative business (think research and development), they will push for platform change. If developers don't know how to modify the platform, or you have a single person who knows it, you have an unacceptable operational or key person risk.

DO WE HAVE NECESSARY RESOURCES?

There are two critical resources that are necessary to build a quality in-house modeling platform: people with the right skill sets, and time. Let's examine both.

People With the Right Skill Sets

Would you go to a brain surgeon to fix a toothache? Then maybe you should think twice before going to an actuary to design and build software for you. We are great at insurance and many other things, but we generally know very little about programming and nothing about software engineering. To build a modeling platform, you need a dedicated team possessing the collective knowledge of actuaries, software engineers and programmers.

- **Software engineers** know how to build software and understand all the considerations that go along with it. They can take your business goals and design a system tailored to meet them. Software engineers will need help with business knowledge, but they can educate you on what is possible, the different techniques of achieving your goals, and the trade-offs involved.

- **Actuaries** possess the necessary business knowledge, but need help with software design, programming and software shop operations.

- **Programmers** are needed to do the actual work of writing the code. They need the help of both actuaries and software engineers to write code in an optimal way.

A stable team that cross-trains on actuarial, software engineering and programming topics can become a development powerhouse, requiring minimal business explanations and displaying impressive efficiency.

Time

Following a proper software development process requires extra time up front but pays off in faster speed of change and less maintenance over time. It is important to explain this to stakeholders and ensure they are onboard with giving you necessary time. Failing to do so results in one of two things:

- Burning out the development team with an unrealistic delivery schedule, which wastes a lot of time on hiring and training replacements.

- Cutting corners and failing to deliver the advertised quality, which can manifest as maintainability, flexibility, clarity and/or runtime issues.

MANAGEMENT BUY-IN AND COMMITMENT

Management buy-in is critical and can be a challenge to obtain. Building a maintainable model platform requires up-front investment in design, training and testing capabilities. The trade-off is that building quality software is an investment that pays off in lower-cost maintenance over time. When considering resource levels over time, usage of the model platform should be considered. The flexibility of the design may mean different functional areas could leverage the platform (as it did with us). It should also be identificed whether the modeling

team can act as a pooled resource across user groups in an efficient and cost-effective way. The challenge here is to make believers of the long-term value that the home-grown platform will provide.

Beyond the investment to build the system, management may also be concerned with having the resources with the right skills to support the model platform over time. After all, what good is a fast, flexible system if no one can interpret the design and make changes. Management support of a dedicated staff and robust model governance practices is important to the long-term success of this approach.

DESIGN PHASE (AKA TRADE-OFFS, TRADE-OFFS, TRADE-OFFS)

When creating a large or sophisticated piece of software, it is crucial to spend time to think through the software design before any code is written. Most actuaries have experience with writing relatively small pieces of code (100 to 2,000 lines) while being the sole developer. In contrast, software companies develop much larger systems with dozens of developers working on various parts of the code simultaneously. On this scale, multiple issues arise that most actuaries never experienced.

Design Goals

In-house systems can and should be designed to meet specific business goals. For example, business can desire the fastest possible model runtime (systems that need to do real-time market analysis), fastest possible development time (prototypes for new products/markets), clarity of the code (mature systems that will be maintained for a long time), flexibility of the code (pricing systems, systems that support multiple products, systems that change often), or some other priority. These goals often contradict each other, but a software engineer can make trade-offs to tailor the system to meet business goals. The first step in designing software is to pick your main design goal. This provides guidance for the software engineer regarding the qualities of the platform that should be maximized.

We selected flexibility and clarity as primary design goals for our actuarial pricing modeling platform.

Object-Oriented vs. Procedural Programming

Code organization can be broadly described as procedural or object-oriented. Procedural code is typically used for applications requiring extreme performance, such as sophisticated real-time market data analysis, or small applications with only a few hundred lines of code. Object-oriented code sacrifices some speed for clarity and maintainability. Generally, the object-oriented approach should be favored since it is easier to maintain. Design of large object-oriented systems requires an experienced software engineer who understands your specific business application.

We selected object-oriented design for our pricing platform since it provides superior flexibility and clarity over procedural design. Notice how our choices are driven by our design goals.

Software Design Principles

Software design principles are a set of the most general and highest-level aspirations for software. No system could or should strictly follow these. These principles are useful to keep in mind when making design decisions. They describe what makes a design good or bad, and help in understanding the tradeoffs being made. A good introductory discussion is available at *https://wiki.base22.com/display/btg/Core+Software+Design+Principles*. Here are the principles:

- Separate code that varies from code that stays the same.
- Program to an interface, not an implementation.
- Favor composition over inheritance.
- Strive for loose coupling.
- Classes should be open for extension, but closed for modification.
- Depend upon abstractions, not concretes.
- Principle of Least Knowledge (interact only with your immediate friends).
- The Hollywood Principle (don't call us, we'll call you).
- A class should have only one reason to change.
- Design to avoid rigidity, fragility and immobility.
    - It is hard to change because every change affects too many other parts of the system (rigidity).
    - When you make a change, unexpected parts of the system break (fragility).
    - It is hard to reuse in another application because it cannot be disentangled from the current application (immobility).

Applying these principles takes some practice; hence the experience of the software engineer matters in applying these correctly.

Design Patterns

A design pattern is a general repeatable solution to a commonly occurring problem in software design. You can think of design patterns as proverbial wheels that you can use to build a vehicle, without having to invent them. Design patterns introduce technical ways of achieving the aspirations laid out in software design principles. An outstanding introduction to design patterns is *Head First Design Patterns* by Eric Freeman and Elisabeth Robson. Just like with design principles, experience is required to apply design patterns correctly.

For our pricing platform, we heavily used strategy and factory patterns. We also used a modified command pattern to achieve our flexibility goal of supporting multiple products in a single platform.

System Blueprint (UML)

Unified Modeling Language (UML) is commonly used by software engineers to design software. It is a standardized diagram convention that lets you describe structures of software, database, use cases and so on. UML class diagram describes the objects in the system, the responsibility of each object, and the data flow between objects. For a sophisticated system, such as a complex pricing platform, a detailed UML class diagram must be completed before any code writing takes place. Completion here also means review with business partners to make sure the platform can support changes in the foreseeable future. The main reasons for completing UML before writing code are:

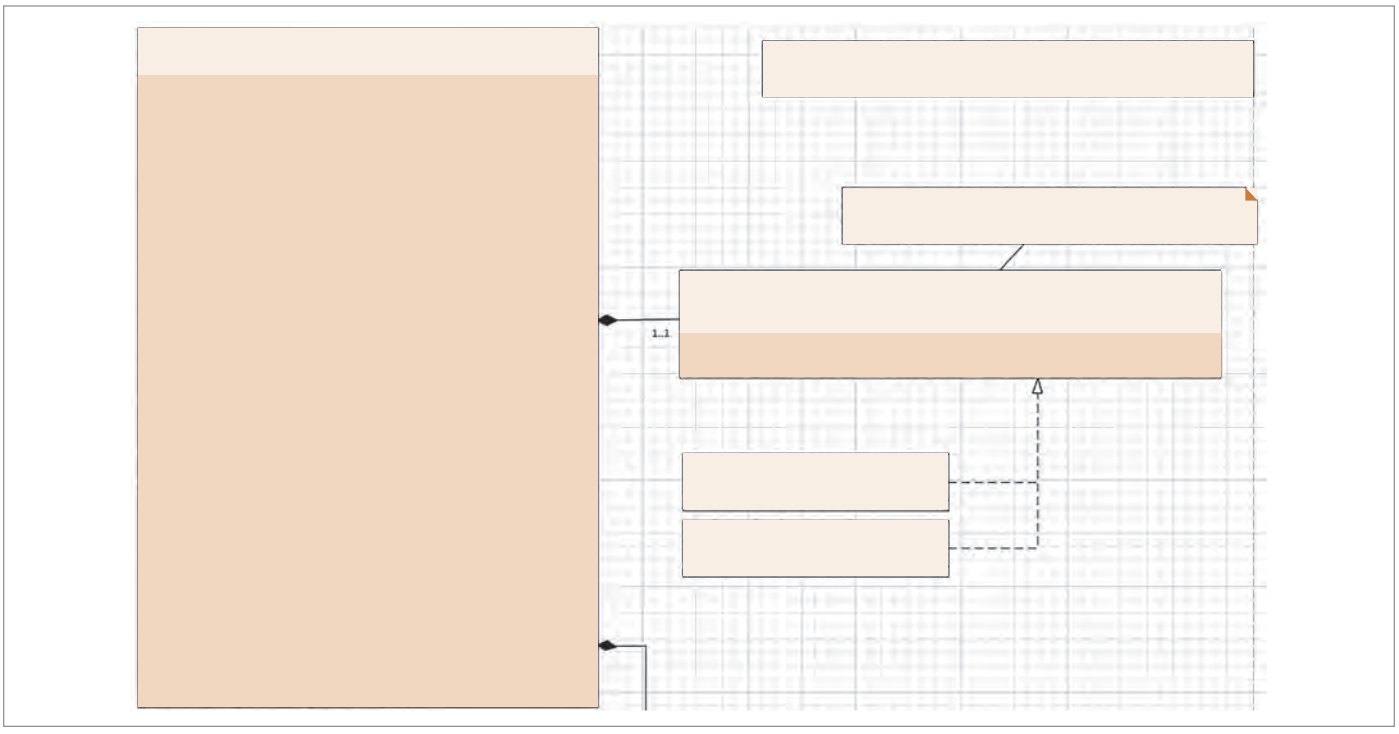
- Identify the responsibility of each object and how each will communicate. This may not seem important for a small system with a handful of objects, but it is critical to document this for a large system with hundreds of objects and multiple developers.

- Multiple developers writing code simultaneously need to rely on communication protocols described in the UML to ensure smooth code integration.

- Conduct role play to identify how various products (including new products) will fit into the platform.This will reveal design flaws that can be addressed much more easily on paper than in the code.

Our completed UML diagram was 70 pages long and could be hung neatly on a large wall (4 meters wide and 2 meters tall). We used Visio to create it, however, you can also use other online tools such as Cacoo. (See Figure 1.) After reviewing it with our business partners, we discovered a major computational inefficiency, which was resolved in three weeks of fixing UML. Fixing

Figure 1
Example of UML Class Diagram



this in the code would have taken far longer and, without this fix, we would have been unable to meet our runtime goal.

Once the UML was finalized we began writing code with a team of five actuaries who had technical guidance from the software engineer. The team worked in parallel using UML as the technical specifications. The model build went very smoothly with no integration issues.

Language Choice

For calculation engines, programming language choice mostly boils down to the following considerations:

1. **Higher-level or lower-level programming language?** Lower-level languages (like C) are faster, but less clear, harder to debug, and require much more programming experience to employ effectively. Manual memory management is a powerful tool, but there are a lot of things that can go wrong, and it requires appropriate training and experience. Higher-level languages (C#, Java, Python) are slower but easier to use, have more "guardrails," and are recommended for less-experienced developers.

2. **Will you need to use specialized libraries that have better support in a certain language?** If you need GPU support, then Python and C++ have some robust free libraries. If you heavily use linear algebra, then MATLAB might make sense. If you don't really need anything special, then pick a free language (MATLAB is not free, for example).

3. **What does my grid support? Can I easily install things on the grid?** If you don't have control over what gets installed on the grid, then it might be necessary to pick a language that caters to the grid. For example, C# would work if your grid already has the .Net framework on it, but you would have a problem if your grid runs Linux and you can't install Mono on it.

For our pricing platform, we picked C#. It is one of the easiest programming languages to learn and use. We did not need any libraries outside of .Net, and our grid already had .Net framework on it.

EXECUTION PHASE

We finally have our detailed UML; we know our design goals; we picked the language to use; and we are now ready to write some code! Here are the ideas and technologies that will greatly speed up the process.

Style Guide

Everyone intends to write "great code," but not everyone interprets that the same due to personal experience and preferences. Great code from a novice programmer usually looks outright awful to an experienced developer. A style guide is a document that lays out the rules of how to write the code. A good style guide includes variable naming conventions, commenting requirements and so on. One of the most popular style guides is Google C++ style guide, which is available at *https://google.github.io/styleguide/cppguide.html*. The benefits of using a style guide include:

- Code written by different developers looks and feels the same. This accelerates the code review process and onboarding of new hires.

- Developers can work with and debug each other's code much easier, since all code looks similar.

- Using a style guide makes code transparent. It reduces the chances of ending up with cryptic code that only the original author understands. Cryptic code is one of the main reasons for model platform changes.

Adherence to a style guide should be continually enforced. It takes time to see the benefits for a programmer who never worked in a team-based development environment.

We used Google's C++ style guide with slight modification for our pricing platform since we used C# and not C++.

XML Doc

XML doc is a technology that allows storage of documentation directly in the code. You can then use programs like Doxygen or write your own interpreter to have HTML documentation generated from tags in your code. XML doc allows for tighter integration of code and documentation. It also allows the use of programmatic hyperlinks, which are easier to maintain compared to hyperlinks in a Word document.

For our pricing platform we wrote our own XML interpreter that generates Microsoft-like HTML documentation.

Version Control System (Git)

Version control software like Git is the most important technology required for development in a team. It tracks versions of the software, allows for very efficient review and merging of the code from multiple developers, and saves enormous amounts of time and effort. Git enables each developer to work in their own branch of code. The project lead can then easily review branches from different developers, and either send them back for additional development or merge them into your most current accepted code. Vincent Driessen wrote a good article on how this should all work (*http://nvie.com/posts/a-successful-git-branching-model/*). You can use online repositories like GitHub and BitBucket, or you can use Git functionality built into Visual Studio and many other development environments.

Input Structure

Most actuaries are familiar with Excel and csv files. However, for large systems these may not be good programmatic inputs; especially Excel, since its data access is very slow. When designing a large system with long-term maintenance in mind, you may want to consider specific file formats like JSON or more robust data storage solutions such as a database. The benefit you are after is generic programmatic data transfer between your user interface and your calculation engine. Adding or expanding a table or adding a new switch should require no coding related to data transfer since all data should be transferred generically. Since data in the JSON file is tagged, your code can handle its transfer from one media to another generically by using metadata. Databases would require some setup of metadata, but a similar approach can be used.

For our pricing platform the Excel user interface creates XML parameter files, which are used by the C# calculation engine. The process is fully automated, so that adding a set of brand-new input tables for a new product requires only defining them in the Excel interface and tagging them with several named ranges. The Excel code and C# code are fully generic for all input tables, and hence require no modification!

Error Handling

You can use exceptions to generate a call trace of the error. This will greatly improve user experience and reduce the time spent on user support. To do this, you simply wrap all your methods in a try catch block generating an error message that contains call description appended with the existing error message from downstream objects. The resulting error message would look something like this.

> Calculate method of Benefit object found negative benefit amount -43. Only positive benefit amounts are allowed for this benefit category.
> Calculate parameters were: benefitName = salariedPlan, category = JointAndSurvivor, amount = -43.
>
> Calculate method of Policy object encountered an exception.
> Policy parameters were: policyNumber = 103945.
>
> Calculate method of Contract object encountered an exception.
> Contract parameters were: contractName = TestCo.

This was one of the favorite features of our model users since it was now very clear why the model didn't run and how to fix it.

Build Order

When building a system from scratch, the most common build order is simply trying to get a runnable skeleton, and then adding actual calculations. Our build order looked like this:

1. **Launcher** is an interface that launches your code locally or on the grid. It verifies that you will not have compatibility issues between your user interface, programming language and grid. Launcher is your simplest runnable model.

2. **Base classes and interfaces to read in inputs** will let you read in your input files, connect to your databases and APIs. This furthers your verification of compatibility and connectivity.

3. **Empty base classes** enable you to have a runnable skeleton of the model.

4. **Detailed report** is a dump of all vectors produced by the model. This will come in very handy for developers since all of them will need an easy way to verify their calculations.

5. **Implementations of actual calculations** should be written last. This step can be written in parallel by multiple developers. At this point each developer can run a model, get inputs

and produce a nice report that includes the vectors that he or she is working on. Very efficient!

Unit Testing Framework

Unit testing framework offers an efficient way to run tests for each class. For example, if you have a class responsible for age calculation, you can use unit testing framework to write a test class that will instantiate your age class with a variety of parameters and check the calculation results. This is very useful as a quick regression test that verifies that code modifications did not break something in an existing class. This works very well on classes that sit at the bottom of the call hierarchy. For top-level classes, such as Launcher or Contract, setting up unit tests is too complicated and hence impractical. Top-level classes should be tested using full model runs and analysis of the outputs.

Optimization

Optimization should be done after each functionality goal is achieved. It is faster and easier to optimize the smallest possible amount of code. Some of the more sophisticated integrated development environments (IDEs) have built-in optimization support. For example, Visual Studio has Performance Profiler, which will record time spent on each calculation and show you what took the longest to run. It will also provide counts of each method call. You get the biggest benefit from optimizing objects that get called the most. Here are some ideas of what could cause performance drag:

- **Searching for something more times than you need.** Examples include looking up the same value inside of the loop or on a lower level than you could and getting items one at a time from a vector in a dictionary, when you could get a vector from the dictionary once and then use indexing to get individual items.

- **Inefficient calculation reuse.** For example, if you compute mortality for every benefit on a policy, it might be much more efficient to compute mortality once and reuse it for all benefits. This is a design issue, though, and may or may not be possible to address easily once you have the code written.

We did two optimization rounds for our pricing platform that yielded runtime improvements of 25 times and further four times, for a cumulative 100 times faster runtime. It really pays to spend time on this!

TESTING PHASE

Once the model is complete, it is time to make sure it is production-worthy. Generally, testing can be broken into two parts. Regression testing makes sure a new version of the model didn't break anything. New functionality testing makes sure additions work as expected. There is not much that can be done to improve testing of new functionality, but there is a lot that can be improved for regression testing.

Automate Your Regression Test Process

You should consider automating your testing process since its efficiency or inefficiency will drive the quality of your regression testing. Ideally, you want to be able to run a full regression test of the new model with a click of a button and get a comprehensive report on the test cases that didn't match. You can then compare the test numbers to your test names and see a pattern. ("Aha! All tests containing a particular benefit failed!") Your automation goal should be that the number of tests does not matter and your tools can handle one test case just as easily as 1 million.

We wrote a custom utility that compares model outputs in two folders and produces a report on the largest mismatch in each test case. We then compared this report to our test bank description to see if there was a pattern.

Build a Quality Test Bank

In theory a regression test should cover all possible input combinations, but practically you need to be able to run and analyze it in a reasonable amount of time. The inputs that should not be used should be programmatically blocked with an appropriate error message. The test bank should include tests that verify that the model will not run with prohibited inputs. The task of constructing the test bank can be done in parallel with development of the model. It is a good idea to have a well-organized document describing the tests, since it will greatly speed up analysis of the mismatches.

Our regression test bank contains more than 6 million test cases, since our automation enables us to run and analyze the results in about a day. The test cases cover all combinations of various toggles and switches for all the products that the system supports.

MAINTENANCE PHASE

Once the model is in production, it is important to maintain and preserve its original qualities. Poor maintenance will deteriorate the model and may result in issues that cause the next model conversion. It is important to be diligent on maintenance.

Maintain UML

Larger model changes should be approached similarly to original model design and hence should be first implemented in UML. This will help you think through the various possible ways to implement your changes and identify possible implementation issues. Make sure that the changes you make flow

well with overall model design. UML also helps with training new developers as it provides an uncluttered way of walking someone through the code flow.

Don't Make a Mummy! If You Need Surgery, Don't Use a Bandage

Many changes can be implemented in several different ways in an object-oriented system. Make sure to pick the way that is most sound from the software design principles and design patterns perspective. It may not be the fastest way to implement the change, but it will save you from having to work with a patchwork of various approaches and implementation styles a year into the model's life.

Maintain Documentation

Documentation should be maintained as part of the development process, especially if you use XML doc. It is easy to describe what you did and why you did it in the code as you write it. It is much harder to do later.

CONCLUSION

There is a lot to learn from practices of IT and FinTech industries when it comes to model building. Some tasks are best handled by integrated cross-functional teams. Technology provides a lot of efficiency, but unlocking its potential requires very close cross-functional collaboration. ■

Igor Nikitin, ASA, MAAA, is a director, actuary at Prudential Financial. He can be reached at *igor.nikitin@prudential.com*.

M. Crew Sullivan, FSA, is a VP and actuary at Prudential Financial. He can be reached at *crew.sullivan@prudential.com*.