Software Archeology
Modernizing Old Systems

by Bryon Moyer, Embedded Technology Journal

They'd been digging for only a couple weeks, with laborious care to make sure they didn't inadvertently destroy that which had lain in quiet repose for centuries. Even so, the outlines of a once-thriving community were starting to reveal themselves. The job right now was to remove the dirt – and only the dirt – and then map out the structures and artifacts as they found them. Creating an outline of the habitation was more or less straightforward – anyone with a reasonably organized mind could come in and identify walls, openings, firepits, and other bits and pieces that, together, formed buildings and a village. And yet, even referring to a village was making too much of an assumption. For that's where the hard part started: interpreting these structural pieces. Was it a village or, in fact, a garrison? Was this building a house? A storage shelter? A stable? Was that fire for cooking? For smithing? What was the intent of those who built all of this?

Re-use of legacy code is a given. Only a startup bears the burden of writing new code from scratch (ignoring that Flash stick that accidentally conveniently jumped into your pocket as you were exiting your last company). And for most companies, legacy code is a "don't ask, don't tell" affair. You don't know what's in it. You don't care what's in it. All you know is that it works. Or seems to. And if anyone so much as breathes on the code, it might break. So it's treated gently, lavished with treats, and bolted onto new software in the least burdensome way possible, without making any direct eye contact that might be considered a challenge.

But there are some areas where more is expected of software before it can be accepted. When making life-or-death equipment, traceability and provability are critical before software can be put into production. When generating code for use in areas of high security, you need to

ensure that there are no hidden gotchas in the code that could compromise a system.

New code can be generated with the requisite paper trail in place, but if you're planning on grandfathering in code that predates such practices, well, it just might not pass muster. That being the case, how can you go back and trudge through tens of thousands of lines of undocumented code to figure out whether it's clean?

This brings us into the realm of "software modernization": taking old code and bringing it and/or its documentation up to the level expected of new software. A specific structure and approach to this has been standardized as the Knowledge Discovery Metamodel (KDM). What is a metamodel? Well, let's use it in a sentence as an example, "I never metamodel I didn't like." OK, sorry, I couldn't resist. It's essentially a structure that can be used for building a model of a specific piece of software. Actually, it's more than that: the resulting model can represent a system, including the code and other resources provided to the code.

KDM can be philosophically compared with a related, and likely more familiar, standard, the Unified Modeling Language (UML). UML is used when generating new code by working from the top down, starting from very high-level abstract constructs and refining until specific finished code can be created. In contrast, a process involving KDM starts from the finished code and tries to build up to a higher level model. As we'll see, it's not strictly a bottom-up process in practice, but, conceptually, it can be thought of as that.

The metamodel is illustrated as having four concentric layers. The inner two represent structure; the outer two represent, more or less, intent. And while, in the model, these just look like different layers, structure and intent are fundamentally different attributes of a program and give rise to different ways of trying to understand what's going on in the code. In particular, structural elements can be automatically derived by parsing the code; intent cannot be: it involves human judgment.

The structure of a program is represented in the Infrastructure and Program Elements layers. The abstractions are completely language-independent, and code artifacts are generalized as much as possible. KDM defines a number of "common elements" that are shared across all languages. According to Hatha Systems's Nick Mansourov, about 80% of a language can be represented through common elements.

But reality is such that each language has very specific semantics that give, for example, a variable in one language characteristics that might not apply in another language. Such realities are accommodated through extra "meta-info" attributes that can be included to represent language-specific semantics. In this manner, qualifiers can be tagged onto common elements without compromising their higher-level "common" nature. These tags are extensions to the standard and aren't themselves standardized. Various tools can define their own; it's that middle ground between having a useless standard because it's too specific and having no standard at all.

In addition, some details are referred to as "micro-KDM"; this allows a more abstract representation of an artifact as, for example, being an action, with the μKDM giving the next level of detail as to what the action was. These elements, unlike the meta-info tags, are part of the standard.

As an example, take the C language statement "`x=A+1`". This would be modeled as an Action Element (having the μKDM attribute "Add") with Read Relations to a Storable Unit `A` and a Value `1` and a Write Relation to a Storable Unit `x`. Storable Units `A` and `x` are further tagged with meta-info "C variable" to layer on the precise semantics of a variable as handled in C. Thus we have four entities – 1 Action Element, 2 Storage Units, and 1 Value – and three relations – 2 Read Relations and 1 Write Relation.

While this example might look like there's enough information to actually execute the action, that's not the general case. KDM allows the creation of relationships between items but doesn't specify things like the order in which actions might be taken; the result isn't executable. An illustration of this can be found in the fact that there is no specific construct available in KDM to represent a loop. Instead, there is a Compound action element that includes entities and relations for the variables being tested and/or indexed as well as the variables and actions enclosed within the loop. It doesn't say that some of the actions are executed over and over until some other condition is met. It just says that these things are related.

This Compound element is an example of an "aggregated relation," an important concept, since it allows low-level behavior to be encapsulated within a unit, exposing only entities that you might consider to be inputs and outputs (or side effects) of the unit. By successive levels of aggregation, you can abstract the behavior higher and higher, in theory aligning with what you might have created had you been designing the software from the top down using UML. Simple in principle; less simple in practice.

While tools exist that can automatically scan the code and build up a model for the Infrastructure and Program Element layers of the metamodel, such tools don't exist for building the next layers: the Resource and Abstractions layers. For this is where we enter the realm of intent, and, thorough comments aside, intent cannot be divined from the code.

The Resource layer helps to complete the picture of how the source code runs, filling in those things that aren't contained within the code itself. There are four sub-sections to this layer to represent the Platform (run-time elements like the OS); the User Interface (presumably interactions with a separate UI); various external Events; and external persistent Data (like a database).

The Abstractions layer is where you find yourself full-on interpreting intent. There are three aspects defined for this: Conceptual (essentially, the functional model, including things like business rules); Structure (the architecture of the software system in components, layers, etc.); and the Build structure.

I got a chance to see a practical way of starting to assess intent through a brief preview by Hatha's Djenana Campara using tools designed to facilitate the process. The idea is that, when breaking a unit into sub-units, the most likely correct arrangement is one that minimizes the number of connections between units. This is because the most efficient aggregation is one that encapsulates lots of "local" interaction, keeping only a few "global" relations visible outside. Using this concept, you can start at the top and break things up into smaller subunits, layer by layer, using names and other clues, and doing so in a way that minimizes relations between entities. Because this involves judgment by a real person, it's not automated.

The most obvious place to start is with the configuration management infrastructure, which is where all the code files are. You can group directories and files in various ways, changing the arrangement around and seeing what happens to the relations between the groupings. Once satisfied with these, you can dive down another level and repeat the process.

This can be augmented by the use of pattern recognition; different software for different purposes may have characteristic patterns that help to identify such things as business rules. There is discussion of having a public pattern library that can be populated by users to better assist an analyst in plumbing the depths of an old program. It can help to direct the building of the model using something more concrete than just minimizing connections between bubbles.

The final result of all of this is a complete model of an old piece of code. You can't really guarantee that it's the exact

same model that the original architect had in mind, but you can certainly create a serviceable facsimile thereof. At the very least, it provides some insight into what's contained within the code. And, in particular, if there's something that just doesn't make sense, it allows you to investigate more closely to make sure there isn't, for example, a flight simulator buried in there somewhere.

Should you actually wish to go in and change some of the program, you now have a map of what's where and can much more readily assess the impact of any changes, with less likelihood of unintended consequences. But most importantly, it can give you a measure of confidence that the old software you are about to deploy in a new system won't bring you any nasty surprises, possibly accompanied by loss of life or limb or security.