Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Software Tracks

Diomidis Spinellis

A generous car reviewer might praise a vehicle's handling by writing that it turns as if it's running on railroad tracks. Indeed, tracks offer guidance and support. When you run on tracks you can carry more weight, you can run faster, and you can't get lost. That's why engineers, from early childhood to old age, get hooked on trains. Can we get our software to run on tracks?

There are various tools that can give our software this ability: tools that increase the accuracy and speed of software development, by forcing it to glide on a firm foundation, keeping it away from risky unexplored territory. These tools span the complete spectrum of software building: from better programming abstractions to automated processes.

Types

The main tool for guiding the code's direction is the language's type system: a trusted friend who doesn't allow us to swerve in dangerous directions. That's why programs written in languages with a powerful type system, like Haskell, often work error-free once they pass the compiler's exacting checks. In contrast, using integers to represent anything from Boolean values, to enumerations, to file descriptors, to array indices, as is typically the case in C code, is a potent source of bugs. Similarly, when we program by randomly assembling functions and procedures, as is the case in many languages that don't enforce design abstractions for code, we will run into problems once the program's size exceeds what can fit in our mind.

At the level of values, we can let the type system help us by establishing a separate type for each distinct class. The Microsoft Windows Software Development Kit defines more than 41,000 hexadecimal constants, FreeBSD (Berkeley Software Distribution) Unix almost 20,000, the GNU/Linux distribution I'm using 26,000. Consequently, I'm sure that every day some hapless programmer passes the wrong value to the wrong function, and then struggles to find out why it failed. Encapsulating these constants inside separate classes would let the type system catch these silly mistakes. Array indices, iterators, and even pointers are preferable to integers, because the alternatives carry the type of array they index, allowing the compiler to verify we're accessing the correct array.

When dealing with code, the type system can again stand by our side. Each time we specify a new class in terms of an interface or an abstract class, the compiler will ensure that we won't forget some crucial methods. A missing method will instantly trigger an error message. As an additional benefit, we minimize dependencies, because the interface's clients don't need to know about related implementations. That's why the Gang of Four advocates programming to an interface, not an implementation.

Interestingly, dynamic languages that lack sophisticated static type checking, like Ruby, Python, and Perl, can offer remarkable boosts in productivity. In their case, expressiveness leads to development speed that creates inertia, and this is another force that keeps the software on its tracks. However, this advantage holds only as long as the relevant code can fit in our head. Otherwise, the tracks are comprehensive tests that won't allow a runaway script to veer off its course.

Domain-Specific Solutions

With domain-specific languages we can efficiently express exactly what their designer intended and nothing more. I've found that the limits a domain-specific language places on its code are just as important as its expressiveness and conciseness. When the code only allows certain operations and expects some mandatory elements, it's easy to communicate this restriction to our client. This keeps frivolous requirements in check, ensures completeness, and results in a product that's easy to learn, use, and maintain. For instance, the quality of the Unix, Perl, and Java reference documentation owes a lot to the domain-specific languages used for writing it (man, POD, and javadoc).

A related approach involves implementing the domain-specific language through a code-generating wizard. In this case the language consists of our answers to each of the wizard's questions; the equivalent of a conversation with the dreaded interactive voice response systems that pass for customer support these days. However, under this alternative we can't easily review our responses or put them under version control; our specifications end up as unintelligible and unmaintainable code. As an example, the Visual Studio 2008 MFC (Microsoft Foundation Classes) Application Wizard will create an empty project of 3,700 C++ lines, 30 files, and 10 classes in seconds. Think of wizards as the fake trains that shuttle tourists around on roads in some tourist destinations: they combine the disadvantages of a car with those of a train.

Sometimes, a software train's particular direction is so straightforward that we can express its evolution simply through data. We first invest in code logic that lets us express common modifications and additions as changes to a data structure. This data can reside in an XML file, a database, or even constants initialized through code. For instance, the bulk of the Firefox user interface is written in XUL—the XML User Interface Language—while Wikipedia's stylish info boxes are written as text templates. Under this approach, all that's needed to add new functionality is to write data in the predefined format, a process typically less error-prone than programming.

Architectures

At a higher level, architectures offer us another way to guide a software's progress. A representative case involves systems designed around plug-ins. Anything that can be expressed in a plug-in is easy to add and integrate in the software's distribution. Other changes to the software, or, heaven forbid, modifications to the plug-in interface are so difficult that only a selected few will implement them. Thus, the system's evolution stays on track, propelled by the changes that are easy to implement (often without the involvement of its core team) in the direction established by the plug-in interface. Noteworthy examples include the hundreds of plug-ins that come with the GIMP raster graphics editor, the Eclipse integrated development environment, and the FindBugs static analysis error checker for Java code. In all cases, crafty developers would drool to mess with the code adding, say, a light-saber effect to GIMP or a detector for their favorite bug to FindBugs. The plug-in-based architecture of these systems ensures that the core code stays squeaky clean, while additions in the software's preferred direction of change evolve organically based on their merits without compromising the system's design or adding tricky interdependencies.

Other architectural styles that enforce a particular open-ended but well-defined interface can serve the same purpose. Examples include the blackboard, pipes and filters, representational state transfer, and rule evaluation architectures. Look at a large successful software system and beneath it you'll find an architecture that's kept its evolution on track.

Look at a large successful software system and beneath it you'll find an architecture that's kept its evolution on track.

Processes

Finally, consider the most flexible track-laying tool of all, the software development process. Because this type of track is often laid out through words, overenthusiastic managers sometimes abuse their mandate, prescribing burdensome and bureaucratic processes—seldom the way to great software. Yet, an appropriate software process can form the high-speed rail that links all the regional tracks together. It will ensure that we don't miss requirements, that our architecture binds efficiently all the software components together, that developers write high-quality code, that our artifacts are well documented, and that the software's releases and maintenance tasks run with the accuracy and speed of the Swiss railroad. A key factor for nailing down processes is tool support. Most developers hate oppressive processes, but love tools: talk to them about the configuration management activity model and they'll yawn, ask them to choose between Git and Subversion and they'll debate all night.

Fortunately, nowadays it's easy to use tools to guide and support any part of the development process. For instance, with periodic automated builds we ensure that our software is always in a consistent state, through issue-tracking systems we can monitor our progress, and we can control maintenance with remote updates and bug reports.

So when your development faces uncertainty, a lack of direction, escalating problems, and lots of pesky bugs, step away from the minutiae and look at the big picture. Think back to your first railroad set and come up with tools that can bring your project, as it were, back on track.

Diomidis Spinellis is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business. Currently he is serving as the Secretary General responsible for information systems at the Greek Ministry of Finance. Contact him at dds@aueb.gr.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**