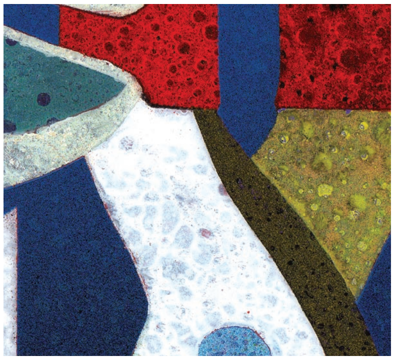
Where's My Jetpack?



Simon Helsen, *SAP*

Arthur Ryman, *IBM Rational*

Diomidis Spinellis, *Athens University of Economics and Business*

Look at the cover of a science fiction novel written 30 years ago, and you'll invariably notice that everyone has a jetpack on their back whose rockets let them fly around effortlessly wherever they choose. In our age of skyrocketing oil prices and chronic traffic jams, this vision seems like a cruel joke. Have software development tools gone through a similar hype-and-bust cycle?

As we'll see in this issue, in a sense they have. Software factory tooling, computer-aided software engineering, and model-driven development tools, to name just a few buzzwords, clearly haven't lived up to their proponents' sometimes-inflated promises. Just as with transportation, software tools' state of the art has taken a more realistic (perhaps even mundane), but not less exciting, route. Today's cars integrate sophisticated electronic steering components and satellite guidance systems. In a similar way, semiautomated software tools monitor a software product's development, evolution, quality, and maintenance throughout its entire life cycle. As with cars, comfort and economics have been the principal drivers, all rooted in the typical pragmatism of developers who must produce real software that solves real problems for real stakeholders. Rather than the predicted progression toward ever-increasing levels of abstraction, two trends have driven the evolution of software development tools: integration at the source code level and a focus on quality.

Back to the source

Source code and software development tools are uneasy bedfellows. Source code's original purpose was for writing instructions to be compiled into executable code. The editors we programmers use to write and maintain source code use heuristics, at best, to make sense of the code; at worst, they simply regard it as a plain sequence of characters. So, it's only natural that software development tool builders wanted to escape from such a lowly communication form (see "A Historical Overview of Software Development Tools"). This quest for higher levels of abstraction led to a plethora of binary and proprietary formats. Designs, models, diagrams, requirements, documentation, version control, and even source code were interpretable only by specialized tools that could display them on a glitzy GUI window.

The inevitable backlash didn't take long to materialize. It turned out that not everything could or should be displayed in graphical form; for many tasks, GUIs were unwieldy, offering mind-numbing

0740-7459/08/$25.00

A Historical Overview of Software Development Tools

Figure A displays exemplary well-known software development tools from the last four decades. The *x*-axis indicates the time when a particular tool was released or announced. The *y*-axis indicates our assessment of the abstraction level at which the tool operates. We intentionally don't define abstraction precisely because it includes several dimensions that would be difficult to define in a 2D graph. However, a higher level of abstraction implies that the tool uses a richer semantic model to manipulate or interpret its artifacts.

We don't intend the figure to be complete, as there are many more development tools. Nevertheless, even though new tools operating at lower abstraction levels are still being introduced, the abstraction level of tools clearly has risen over time.

Over these four decades of software development tools, we've identified three major eras that drove the focus of the invented tools. In the structured era, tools mostly supported structured programming. In the object-oriented era, a lot of tools were introduced to support OO design and development. Finally, the Internet era has seen the introduction of tools to support distributed Internet-based software development.

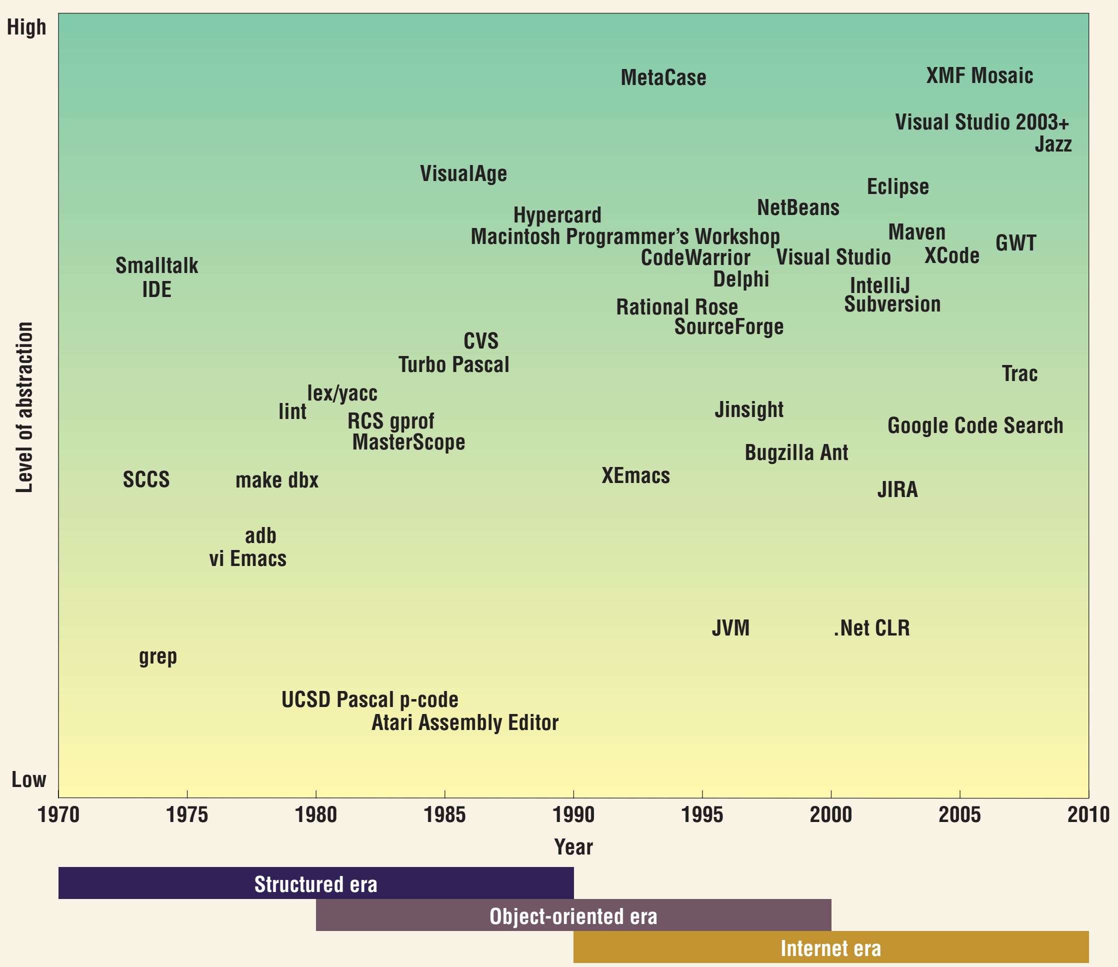


Figure A. Well-known software development tools spanning the last 40 years and three eras. The tools support higher levels of abstraction over time.

repetitiveness for what text-based tools could achieve with a few clever commands. The problem was aggravated by the lack of integration between text-based and graphical formats and their tools. Code generation from graphical artifacts made relatively trivial but extremely important activities such as debugging difficult, forcing programmers to edit the generated source code to avoid lengthy roundtrips between the binary graphical formats and the source code. Sharing code among various communities through the Internet put further nails in the coffin of proprietary binary and graphical formats.

URLs for More Information

- **Office Open XML file format:** www.iso.org/iso/pressrelease.htm?refid=Ref1123
- **Eclipse Platform:** www.eclipse.org
- **The Netbeans Platform:** www.netbeans.org
- **The IntelliJ IDEA:** www.jetbrains.com/idea
- **Java 5.0 annotations:** http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
- **C# attributes:** http://msdn.microsoft.com/en-us/library/z0w1kczw.aspx
- **JUnit testing framework:** www.junit.org
- **Declarative drawing of UML diagrams:** www.umlgraph.org
- **Enterprise JavaBeans technology:** http://java.sun.com/products/ejb
- **XML:** www.w3.org/XML
- **XML Schema:** www.w3.org/XML/Schema
- **Web Services and Description Language:** www.w3.org/TR/wsdl
- **Latest SOAP versions:** www.w3.org/TR/soap
- **Enhancing all versions of Smalltalk with XML:** http://xml.smalltalk.org

We're currently witnessing a similar move in the storage formats of office productivity applications, where binary files are becoming obsolete, replaced by standardized XML alternatives. (See the "URLs for More Information" sidebar to learn more about the tools mentioned in this article.)

Standardized text-based file formats preserve the intellectual capital invested in authoring the documents and code, allowing their communication across space and, more importantly, time. Tools that are often tied to operating systems and vendors undergo rapid evolution and often become obsolete. With the storage format decoupled from the authoring tool, however, the artifacts can live on. In the software development world, the only stable formats are programming-language source code files, and we now see an explosion of tools that work on this code. Often we can perform many software lifecycle activities such as documentation, verification, testing, and quality assurance directly on the source code. Even sophisticated integrated development environments such as Eclipse, NetBeans, and IntelliJ essentially operate on text artifacts, even though they internally construct and maintain a complete, transient model of the source code to facilitate powerful navigation and context-aware tasks.

The nature of source code has also evolved to accommodate the tools. Modern languages provide extension facilities—for example, annotations and tags in Java and attributes in C#. These mechanisms permit developers to embed metadata into the source code. This metadata can then be independently evaluated to help other aspects of the software development process. Examples include instructing a testing framework (as with JUnit 4), creating UML class diagrams (for example, UMLGraph), or guiding deployment on an application server (as with EJB 3). Furthermore, many tools increasingly rely on comments, documentation tags, coding conventions, and reflection for analyzing the source code. These mechanisms, although not perfect, are taking the role of duct tape, piecing together a complex, rich, custom-designed, organically grown process. Source code has become the bus that tools tap into for communicating with other tools. Stable language standards keep in check but also limit what tools can generate and process.

Quality time

Another interesting development in the software tools arena is the increasing focus of development tools on software quality. This is remarkable because it may well be a sign that software development, as we now practice it, is moving outside the comfort zone of what we can reliably build both as individual developers and as members of process-controlled teams. And, to make software development even more challenging, important trends such as agile development methods and globally distributed, loosely coupled teams require new project management practices and processes to control quality attributes.

The economics of software quality tell us that the earlier a problem is found, the cheaper it is to fix. Focus has therefore shifted from defect removal in the later phases to defect prevention in the earlier phases. For example, agile practices such as continuous integration, test-driven development, and customer collaboration all aim to prevent defects from escaping into delivered code. Tools help us here by detecting bugs, semiautomating test-case creation, automating refactoring tasks, monitoring the quality of produced software artifacts, increasing the ability to reuse distributable applications, and finding the reusable wheat in the tons of chaff available on the Internet. This special issue includes articles on tools that address each of these topics.

Venturing outside a comfort zone is always a risky proposition. There we can reap rich rewards and learn valuable lessons, but there also lie dragons waiting to eat us alive. In the context of software development tools, it's not clear whether in the long term the additional effort required for detecting and correcting quality defects after we write the code can scale to match software's increasing complexity. On the other hand, we might also interpret the increased number of development tools that support quality as a sign that the software development profession is maturing beyond individuals' skills. Just as an automotive engineer today employs very sophisticated tools to monitor the quality of produced parts, a modern software developer needs quality-supporting tools to develop large, multithreaded, distributed applications in a timely manner and with adequate quality. On the positive side, this might arguably indicate that software development is becoming more an engineering discipline than an art.

Challenges and outlook

As we've seen, we expect software development tools to do more with less. In many real projects, tools are the major driver for maintaining quality, and we expect them to take up this role by relying mostly on the project's source code.

Multiple attempts over the last two decades to increase the level of programming abstraction have failed but are also unlikely to stop. We managed to move from machine code to assembly language and from there all the way to modern third- and even fourth-generation programming languages. So, what's next? And what does it mean for tools? Source code is still expressed in one specific programming language and therefore bound to that language's existence and support structure. However, having the software industry standardize on one programming language is unlikely and undesirable, not only because of competitive reasons but also because we'd want to write software for many domains and specializations.

In many branches of commercial software development, XML and XML Schema are becoming the lingua franca to express content in a domain-specific manner (as expressed by the countless domain-specific schemas). Such artifacts are universally interpretable because of the widespread acceptance of XML and because XML is textual and (somewhat) human readable. Today, the integration of heterogeneous business-software landscapes is largely based on Web services using schema standards such as WSDL (Web Services Description Language) and SOAP, which are XML-based. In fact, numerous software development tools already exist for these schema standards, letting us manipulate artifacts at a higher abstraction level, namely at the domain level.

Although many elements of a software system are now encoded in XML, source programs are still expressed in their own unique syntaxes. This is because most existing general-purpose programming languages have enormous expressive power. Their domain is usually large enough that developers are still willing to learn their proprietary syntax. A notable exception to this is Smalltalk, which has a bare-bones complete syntax and is converging on an XML standard for tool integration. Yet, many language-specific frameworks require developers to express additional attributes in the form of a programming-language-independent format, usually XML. And for many of these frameworks, domain-specific tools are available to edit these additional attributes. This indicates that domain-specific XML-based elements are eating away some of the general-purpose programming language pie.

It's impossible to conclude that domain-specific languages, whether or not they're encoded in XML, will eventually replace general-purpose languages and thus permit a universal integration for development tools at a higher abstraction level than just plain text. However, it's safe to say that we'll use software development tools only if they follow economic principles and permit software projects to integrate their artifacts in an effective, transparent, and portable way. At the end of the day, we all do want jetpacks, but maybe not ones that are powered by unaffordable gasoline. And we want them to have enough safety features so that we won't crash.

Acknowledgments

The views expressed in this article are those of the authors and not necessarily of their respective employers.

About the Authors

Simon Helsen is a development architect at SAP, working on an enterprise-level metadata repository for design-time tools on top of NetWeaver's Composition platform. He has worked on the ArcStyler MDA tool while at Interactive Objects Software and has published several articles and a book chapter on model-driven development tools and techniques. His current interests range from scalable model and code-generation paradigms to high-volume repository technologies. Helsen received his PhD in computer science from Freiburg University. Contact him at shelsen@computer.org.

Arthur Ryman is a distinguished engineer in IBM's Rational Division. He works at the IBM Toronto Laboratory, where he develops tools, most recently Rational Application Developer. He was a leader of the open source Eclipse Web Tools Platform project and coauthored a book on that topic (Addison-Wesley, 2007). Ryman has a PhD in mathematics from Oxford University and is an IEEE senior member. Contact him at ryman@ca.ibm.com.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). He leads the Software Quality Observatory for Open Source Software, a European Union-funded project for establishing a suite of software quality assessment tools for open source software. Spinellis received his PhD in computer science from Imperial College London. Contact him at dds@aueb.gr.