NOVEMBER/DECEMBER 2015

IEEE Software

IEEE
IEEE computer society

WWW.COMPUTER.ORG/SOFTWARE

Keep Your Career Moving Forward

Renew Early for an Exclusive Offer!

Knowledge + Connections + Advancement = The Best Value of an IEEE Computer Society Membership

- Receive *Computer*, **the society's flagship magazine**, published 12x a year
- **Plus a second FREE digital publication** – choose from *IEEE Software*, *IEEE Security and Privacy*, *IT Professional* or *IEEE Micro*
- **Access to over 6,000 Safari Books Online and Books 24x7** – saving you thousands of dollars
- **13 Knowledge Centers** with online courses covering numerous topics such as Cisco, Oracle, IT Security plus Ethical Hacking and SCRUM
- **Stay competitive with certifications for all levels** – get the support and resources you need with unlimited access to books, videos and practice exams whenever you need it
- **Network, discuss, create future products & services** and help shape the organization's future on emerging topics by joining either a **Special Technical Community** or a **Technical Committee**

With your membership you'll also get:

- **12 FREE downloads** to the Computer Society Digital Library – a $120 value
- **3 FREE webinars** lead by Computer Society subject matter experts
- **A FREE subscription** to *ComputingEdge*, a collection of related articles from all of the Computer Society's 13 magazines
- **Deep discounts on registration fees** for more than **200 conferences** on a wide range of topics
- **A free digital copy of the IEEE CS 2022 Report** – a look at where technology is going

FEELING LUCKY?

Renew Early for Your Chance to Win 1 of 5 Apple Watches*

*Renew by 15 November 2015 for your chance to win. Winner will be randomly selected from all eligible entries.

www.computer.org/join

IEEE computer society

The Community for Technology Leaders

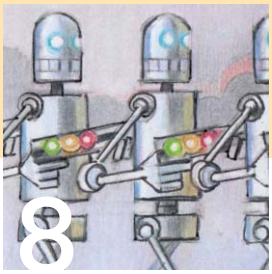
IEEE Software
TABLE OF CONTENTS
November/December 2015 Vol. 32 No. 6

FOCUS

REFACTORING

FEATURE

MISCELLANEOUS



DEPARTMENTS

See www.computer.org/software-multimedia for multimedia content related to the features in this issue.

For more information on computing topics, visit the Computer Society Digital Library at www.computer.org/csdl.

FROM THE EDITOR

Editor in Chief: **Diomidis Spinellis**
Athens University of Economics and Business, dds@computer.org

Extending Our Field's Reach

Diomidis Spinellis

I RECENTLY COLLABORATED with a digital-typography expert to create a formatting style for a publishing project. I thought we had agreed to work together through GitHub, which would let us share the current version of the manuscript and easily integrate the LaTeX style files by merging their corresponding development branch. Instead, a few weeks later, I received a .zip archive file over email.

The style was beautiful, with great attention to detail and typographical niceties. The archive's contents were also interesting. It contained two directories with similar but not identical contents: one for all projects requiring the specific style and one slightly tailored to my needs. Each directory contained tens of third-party files (some in binary format), log files, documentation, and automatically generated files. The style's source code files also contained things that might trouble a software engineer: outdated or inconsistently indented comments, copy-and-pasted and commented-out code, and a few overly long lines. To be fair, some of the style code seemed to have been written more than two decades ago, and we all know how software ages.

> Almost all software engineering processes can benefit industries that work with executable knowledge.

Still, whenever I collaborate with publishers, I'm always saddened to see the opportunities for process improvement they miss by not using the best practices and tools we've established in software engineering. For example, instead of collaboration using simple text markup over an online revision control and review system, documents in diverse incompatible binary formats are shuttled back and forth over email and, yes, FTP, with changes and comments embedded obscurely (sometimes with typographical marks scribbled on the margins of scanned paper). This creates an integration nightmare, which is only partially controlled by draconian, inflexible change-management policies and heroic efforts of all the involved parties. Once a document leaves a specific stage—say, drafting, copy-editing, or composition—there's no going back, and nobody can trace changes on an end-to-end basis. I know that some publishers use version control systems, but even there, due to the lack of build-process automation, such use often degenerates into that of a shared disk drive.

Picking on publishing would be wrong; other industries are also producing what's in effect software (ex-

IEEE Software Mission Statement: To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

ecutable knowledge) but not treating it as such. As examples, consider 3D printing, numerical control of machine tools, filmmaking, pharmaceutical laboratory automation, and workflow management. Other activities, with more abstract output, include project planning and drafting laws and regulations. The output of these activities shares considerable similarities with software: laws are constantly revised and refer to parts of each other, similarly to subroutines. These examples don't include devices in which software (the kind that runs on a CPU) is taking an ever-increasing role, such as cars, planes, phones, medical devices, and TVs. There, the problems and missed opportunities are much less severe; trained software engineers typically perform and manage the development.

Although many industries have developed their own highly effective processes over the years, software engineering maintains an essential advantage. It has developed methods and tools that let even small teams manage extremely high complexity. For example, compare the nine million LOC in the Linux kernel (which often forms only a small part of a much larger software stack) with the few tens of thousands of components in a modern car. This advantage is important because the complexity in nonsoftware activities is also increasing inexorably. Films used to have a few hundred scenes and takes, which filmmakers could easily manage by writing on a clapperboard. In contrast, modern blockbusters might depend on tens of thousands of digital artifacts. Also, processes that skilled humans executed only a few times in the past (for example, creating a car model for aerodynamic testing) can now be easily rerun (for example, by a 3D printer) with the touch of a button.

Almost all software engineering processes can benefit industries that work with executable knowledge. Requirements engineering can improve analysis, specification, validation, and traceability (why do we drill this hole at the bottom of the engine block?). Software design can be essential to control complexity through modeling, abstraction, control of coupling and cohesion, decomposition, encapsulation, and separation of concerns.

Returning to the example I presented at the beginning, the two directories I received from the publisher should have been divided into a meaningful hierarchy (decomposition) and their contents united through some application of polymorphism. Construction techniques can be used for the promotion of agility, build automation, continuous integration, reuse management, and verification. Imagine a film director being always able to use continuous integration to view the most current version of a film as it develops over the months. Software-testing techniques can reduce waste and increase efficiency, while software-inspired maintenance activities can increase a product's or process's longevity.

I believe that configuration management tools and techniques are the most productive way through which software engineering can affect other fields. The powerful tools and platforms we've developed (think of Git and GitHub) allow the effortless sharing of work over distance and time zones, the documentation and traceability of changes, the integration of issue and change management, commenting on each other's work, the organized development of separate product lines, and the

EDITORIAL STAFF

FROM THE EDITOR

pain-free merging of work items and branches. Engineers in other fields often look at these feats as alien technology. Some people, though, are recognizing the potential; consider the appearance of French civil code and German laws on GitHub (https://github.com/steeve/france.code-civil and https://github.com/bundestag/gesetze).

Exporting our hard-earned knowledge to other fields won't be easy. Each has distinct goals, competencies, values, and traditions. To communicate effectively, we must develop a shared vocabulary and way of thinking. Perhaps education is the easiest path: train practitioners from other disciplines to think and act as software engineers.

Software engineering has benefited mightily from research in fields ranging from electrical engineering and physics to mathematics and management science. It has changed our world beyond recognition by putting the artifacts it produces on billions of devices. Now, the time has come to transform our world in another way, by giving back to science and technology the knowledge software engineering has produced in the past half century.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

HOW TO REACH US

WRITERS

For detailed information on submitting articles, write for our editorial guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

MEMBERSHIP CHANGE OF ADDRESS

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES

If you are missing an issue or you received a damaged copy, contact help@computer.org.

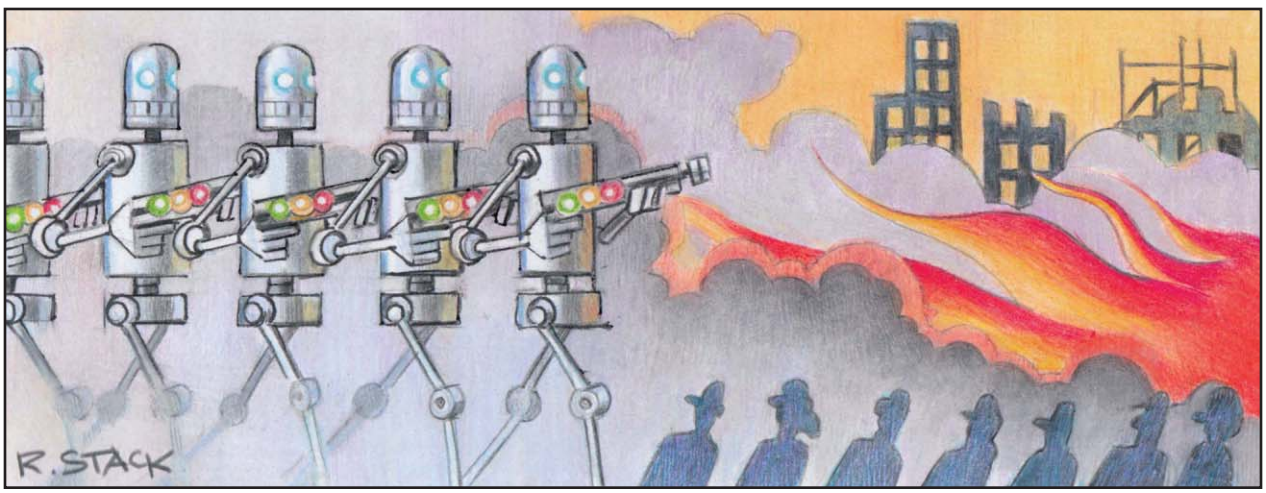
REPRINTS OF ARTICLES

For price information or to order reprints, email software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.

Editor: **Grady Booch**
IBM, grady@computingthehumanexperience.com



I, for One, Welcome Our New Computer Overlords

Grady Booch

AS IT TURNS OUT, I'm in very good company: Elon Musk, Bill Gates, Clive Sinclair, Stephen Hawking, and even Steve Wozniak all agree that the use and power of AI will increase.

But beyond that, I would argue that they're all wrong.

The thing on which we disagree is a fear of the rise of extrahuman AI, an event about which each has expressed strong concern.

Those Against

In an interview with the *Guardian*, Elon said, "I think we should be very careful about artificial intelligence. If I had to guess at what our biggest existential threat is, it's probably that."[1]

During a Reddit Ask Me Anything interview, Bill noted, "I am in the camp that is concerned about super intelligence. First, the machines will do a lot of jobs for us and not be super intelligent. That should be positive if we manage it well. A few decades after that though, the intelligence is strong enough to be a concern. I agree with Elon Musk and some others on this and don't understand why some people are not concerned."[2]

Clive Sinclair has posited, "Once you start to make machines that are rivaling and surpassing humans with intelligence, it's going to be very difficult for us to survive."[3]

In an interview with the BBC, Stephen stated, "The development of full artificial intelligence could spell the end of the human race."[4]

Woz expressed these same concerns in a wonderfully playful way, stating,

> *I agree that the future is scary and very bad for people. If we build these devices to take care of everything for us, eventually they'll think faster than us and they'll get rid of the slow humans to run companies more efficiently. Will we be the gods? Will we be the family pets? Or will we be ants that get stepped on? I don't know about that. But when I got that thinking in my head about if I'm going to be treated in the future as a pet to these smart machines, well I'm going to treat my own pet dog really nice.*[5]

Nick Bostrom, a University of Oxford philosopher, explores the existential threat of AI in his compelling, well-reasoned book *Superintelligence: Paths, Dangers, Strategies*.[6] I'll leave it to you, dear reader, to metabolize his entire work, but I'd offer that the essence of his argument lies in this statement:

> *At some point, the AI becomes better at AI design than the human programmers. Now when the AI improves itself, it improves the thing that does the improving,*

 0740-7459/15/$31.00 © 2015 IEEE

> *an intelligence explosion results—a rapid cascade of recursive self-improvement cycles causing the AI's capability to soar. … At the end of the recursive self-improvement phase, the system is strongly superintelligent.*[6]

Thus, argues Nick, some threshold exists beyond which an AI would become an existential threat to humanity; therefore, our immediate action should be to take steps to restrain, control, and otherwise limit the reach of such a superintelligent agent. This is one reason for the formation of the Future of Life Institute (http://futureoflife.org). Perhaps somewhat in the spirit of Pascal's wager, Elon has provided significant funding for the institute's research.

Those For

Before I go on, I should offer my full disclosure: I'm an early signatory to the Institute's "Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter."[7] As the letter states, "We believe that research on how to make AI systems robust and beneficial is both important and timely, and that there are concrete research directions that can be pursued today."[7]

Indeed, there's no denying that those of us who build this technology must do so intentionally and with full consideration of our work's potential consequences.

Rather than dive into a detailed analysis of Nick's work, I instead refer you to others who have taken on this task. Paul Ford offers a useful take on the matter in "Our Fear of Artificial Intelligence."[8]

I'm particularly impressed by Sebastian Benthall's deep analysis and how he questions the potential of recalcitrance.[9,10]

Rodney Brooks also has a say on the matter:

> *I say relax everybody. If we are spectacularly lucky we'll have AI over the next thirty years with the intentionality of a lizard, and robots using that AI will be useful tools. And they probably won't really be aware of us in any serious way. Worrying about AI that will be intentionally evil to us is pure fear mongering. And an immense waste of time.*[11]

Fear sells. In fact, fear fuels a great deal of contemporary news reporting and politics (if it bleeds, it leads, as the saying goes) because fear touches us on a fundamental, emotional level. It's easier to talk about what we fear than to act on that fear. This is why it's important to cut through the fear-mongering and examine why we fear the rise of superintelligence: we fear it because it calls into question what it means to be human.

> Fear touches us on a fundamental, emotional level.

To be clear, I'm not accusing Nick, Elon, Bill, Clive, Stephen, or Woz of being fearmongers: they're all speaking from their heart and their experience. Therefore, I accept that this fear of the rise of superintelligence isn't irrational.

However, it's not, as Rodney observes, an immediate fear, nor is it, in my estimation, a fear of probable things. At the worst, it's a highly misguided fear.

Rodney observes that we're a long, long way off, and I agree with him. He notes, "I think it is a mistake to be worrying about us developing malevolent AI anytime in the next few hundred years. I think the worry stems from a fundamental error in not distinguishing the difference between the very real advances in a particular aspect of AI and the enormity and complexity of building sentient volitional intelligence."[11]

These days, I'm deeply involved in creating cognitive systems. I've seen how the sausage is made. It's hard work, it's not always pretty, and although some exciting breakthroughs have occurred in perception and deep learning, we still have a long way to go.

Might a superintelligent AI emerge? In some distant future, perhaps, but we must realize that the journey of crafting such an AI changes the world along the way. So, for us to project our fears through the lens of who we are today misses the reality that when such an AI forms, we won't be the same as we are now.

What I Do Fear

That's why I suggest that we need to stop worrying and learn to love the AI, because otherwise we'll be distracted from a very clear and present

danger. I don't fear the rise of superintelligent AI as do Nick, Elon, and Bill; what I do fear is the fragile software on which society relies.

We've surrendered ourselves to computing. Software isn't just eating the world, says Marc Andreessen; it's now the foundation on which modern civilization flourishes.[12] As a computing insider, I recognize that this digital edifice is exquisitely complex and fragile. The many security breaches we encounter daily are only the visible fracture lines in this infrastructure; the real fault lines lie much deeper, in the vast amounts of legacy code that run the world.

Peter Neumann gives us an accounting of some of the risks to the public that arise from computing,[13] and I suspect his work touches only a small fraction of the tectonic forces. The loss of personal privacy, the growth of the digital divide between the haves and the have-nots, the impact of computing on employment, the disruption of industries at a speed faster than society can easily metabolize, the way that computing is changing not only social intercourse but also the way of war—these are the things that keep me awake at night, not the potential rise of a superintelligent AI of our own making.

Earlier, I quoted Clive, who said, "The development of full artificial intelligence could spell the end of the human race."[3]

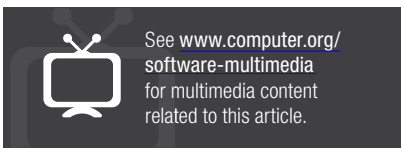
My immediate reaction to his point of view is this: Clive, you say that as if it's a bad thing. Perhaps our ultimate fate is the creation of a successor species that supersedes the human race. Why should we as humans expect a privileged place in the cosmos, simply because we're human? Instead, perhaps we should embrace the potential of the journey of what we might become in this coevolution of computing and humanity. I expect that we'll never notice a singularity because we'll slowly, irreversibly, inevitably become machines ourselves.

I do not fear our new computer overlords. Indeed, I welcome them.

References

1. S. Gibbs, "Elon Musk: Artificial Intelligence Is Our Biggest Existential Threat," *Guardian*, 27 Oct. 2014; www.theguardian.com/technology/2014/oct/27/elon-musk-artificial-intelligence-ai-biggest-existential-threat.
2. P. Holley, "Bill Gates on Dangers of Artificial Intelligence: 'I Don't Understand Why Some People Are Not Concerned'," *Washington Post*, 29 Jan. 2015; www.washingtonpost.com/news/the-switch/wp/2015/01/28/bill-gates-on-dangers-of-artificial-intelligence-dont-understand-why-some-people-are-not-concerned.
3. "Out of Control AI Will Not Kill Us, Believes Microsoft Research Chief," *BBC News*, 28 Jan. 2015; www.bbc.com/news/technology-31023741.
4. R. Cellan-Jones, "Stephen Hawking Warns Artificial Intelligence Could End Mankind," *BBC News*, 2 Dec. 2014; www.bbc.com/news/technology-30290540.
5. D. Storm, "Steve Wozniak on AI: Will We Be Pets or Mere Ants to Be Squashed Our Robot Overlords?," *Computerworld*, 25 Mar. 2015; www.computerworld.com/article/2901679/steve-wozniak-on-ai-will-we-be-pets-or-mere-ants-to-be-squashed-our-robot-overlords.html.
6. N. Bostrom, *Superintelligence: Paths, Dangers, Strategies*, Oxford Univ. Press, 2014.
7. "Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter," Future of Life Inst.; http://futureoflife.org/AI/open_letter.
8. P. Ford, "Our Fear of Artificial Intelligence," *MIT Technology Rev.*, 11 Feb. 2015; www.technologyreview.com/review/534871/our-fear-of-artificial-intelligence.
9. S. Benthall, "Bostrom's Superintelligence: Definitions and Core Argument," blog, 20 Aug. 2015; http://digifesto.com/2015/08/20/bostroms-superintelligence-definitions-and-core-argument.
10. S. Benthall, "The Recalcitrance of Prediction," blog, 28 Aug. 2015; http://digifesto.com/2015/08/28/the-recalcitrance-of-prediction.
11. R. Brooks, "Artificial Intelligence Is a Tool, Not a Threat," blog, 10 Nov. 2014; www.rethinkrobotics.com/blog/artificial-intelligence-tool-threat.
12. "Why Software Is Eating the World," *Wall Street Journal*, 20 Aug. 2011; www.wsj.com/articles/SB10001424053111903480904576512250915629460.
13. "ACM Forum on Risks to the Public in Computers and Related Systems," *Risks Digest*, vol. 28, no. 93, 3 Sept. 2015; www.csl.sri.com/users/risko/risks.txt.

GRADY BOOCH is an IBM Fellow and one of UML's original authors. He's currently developing Computing: The Human Experience, a major transmedia project for public broadcast. Contact him at grady@computingthehumanexperience.com and follow him on Twitter @grady_booch.



INSIGHTS

Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmerman**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

The Connected Car in the Cloud

A Platform for Prototyping Telematics Services

Tobias Häberle, Lambros Charissis, Christoph Fehling, Jens Nahm, and Frank Leymann

As cars turn into computers on wheels, it becomes vital to experiment with novel application scenarios and to envision suitable abstractions for integrating cars into existing enterprise information systems. This insightful project report tells the story of a prototyping platform for connected-car software and shares the project's experience with a cloud-computing pattern language that helped drive the architectural platform design.
—Cesare Pautasso and Olaf Zimmerman, department editors

CONNECTED CARS are hitting the road. Almost every automotive company has a solution for a connected car.[1] We already use many connected devices in our daily life—for example, smartphones and tablets. The connected car adds to this portfolio with its own flavor.

This article shares our experiences delivering services for the connected car. We built the Connected-Car Prototyping Platform and reusable application templates to enable prototyping of telematics services. The platform and templates simplify prototype development and reduce time-to-market. They're based on cloud principles to achieve lower initial setup costs for prototypes and better scalability.

Addressing the Challenges of Connected Cars

Providing connected services—that is, apps—for smartphones has become simple owing to a range of frameworks and back-end platforms such as Parse (https://parse.com). Such frameworks and platforms offer software development kits for many mobile platforms and programming languages. They typically provide services such as data storage, hosting, and analytics to improve the development experience.

Connected-car platforms, however, present more challenges: longer life cycles (a car typically runs 10 or more years), less IT standardization, less reliable connectivity (typically, parked cars are offline), a bigger variety of connectivity hardware, and so on. In addition, the automotive industry and the related multimedia industry are still unclear about customer demand. There's an ongoing discussion about the killer features for the next generation of connected cars.[2,3] Without knowledge of future customer demand, time-to-market and flexibility become crucial.

To address these challenges and simplify development of telematics services, we built the Connected-Car Prototyping Platform. It provides a back end for applications interacting with cars. For us, the connected car is "just another" connected device similar to smartphones, watches, or even parking lots. The platform provides an abstraction of such

0740-7459/15/$31.00 © 2015 IEEE

connected devices for developers. It also delivers services such as identity management and data storage—for example, for nonrelational data. These services are the foundation for many value-added services and applications. So, our platform offers a run-time environment for many telematics services. It also supports properties associated with cloud computing, such as self-service for developers and dynamic scalability of hosted applications. To enable such properties, we employed cloud-computing patterns[4] to design the platform and develop a reference architecture for hosted applications. Developers can focus on application functionality and don't have to deal with infrastructure or middleware.

In addition, application templates provide architecture blueprints and code snippets, on which developers can base their application prototype implementations. The templates homogenize prototype design, further accelerating application development and simplifying runtime management.

To design the platform and templates, we applied Christoph Fehling and his colleagues' design process.[5] Each step of this process considers a certain set of the cloud-computing patterns to use in the application architecture. (However, developers don't need this process to create telematics service prototypes, which are based on the templates.)

Our design process had five steps:

- *Decomposition* divides the application functionality into separate components.
- *Workload characterization* estimates component use over time.
- *Data (state) management design* identifies stateful and stateless components.
- *Component refinement* designs, in detail, functionality for user interfaces, processing, and data access.
- *Elasticity and resiliency configuration* describes run-time behavior to address workload changes and faulty components.

> To enable properties such as scalability, our design process considers cloud-computing patterns in five steps.

The patterns we used are at www.cloudcomputingpatterns.org.

The Prototyping Platform

To design our platform, we first collected high-level functional and nonfunctional requirements as a baseline. We decomposed the platform functionality according to the supported domains (see *Domain-Driven Design: Tackling Complexity in the Heart of Software*[6] for more information on domain-driven decomposition). This led to six application components (platform services): identity and access, the telematics service repository, user profiles, billing, communication, and the virtual vehicle.

Figure 1 depicts these platform services and their data-handling capabilities. The services are identified to address initial functional needs. They may then be consumed by other value-added (telematics) services, applications, or user front ends—for example, to recommend the most suitable fuel station or parking lot on the basis of the vehicle's location, fuel status, and so on.

The architecture specifically addresses the high degree of uncertainty and the need for flexibility. In particular, workloads are unpredictable. They depend on factors such as the number of vehicles and users as well as possible resource-intensive applications running on the platform. So, we designed each platform service to handle unpredictable workloads.

The platform decomposes a service into a business logic tier and a data tier, separating the services' business logic from data storage. The separation of tiers and concerns allows independent scalability of each tier.

The business logic tier is the heart of each service, ensuring correct functionality. To ease scalability, this tier is stateless. State is either handled in the storage services or provided with each request to the tier. A central load balancer handles scaling.

Data, such as vehicle positions and service intervals, is stored in the data tier, which therefore is stateful. The data can be stored independently of applications or use cases. The consistency level at this tier varies, depending on the underlying service and its data classification.

For example, we designed the identity-and-access service using the Relational Database and Strict Consistency patterns because it holds

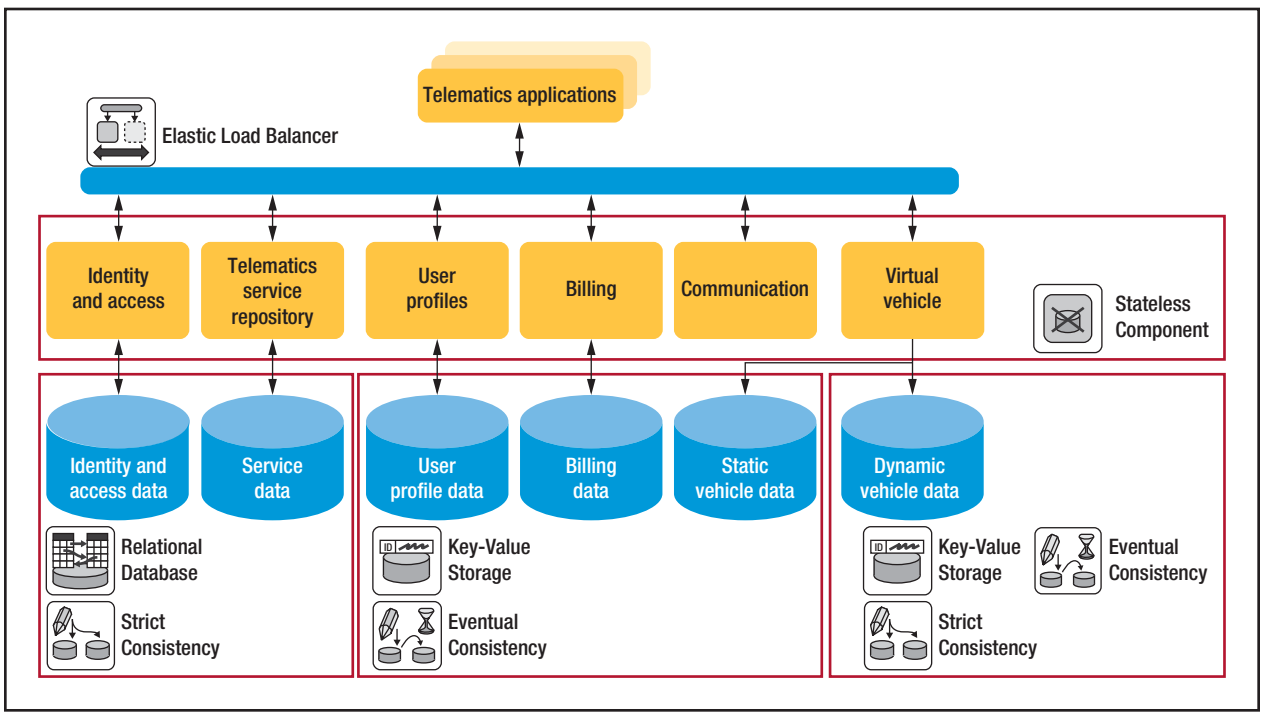


FIGURE 1. The Connected-Car Prototyping Platform architecture.[7] Telematics applications are built on top of platform services, which are stateless. The applications' data tier handles state for the platform services. Data storage depends on data characteristics (for example, key-value storage for dynamic vehicle data).

complex data relationships. The definition of access policies is based on the association of users, vehicles, and services. Changes must be reflected immediately—for example, when service access is revoked for a user. Another example is the virtual-vehicle service, which provides an abstraction of vehicles. It stores dynamic vehicle data in a key-value format. For performance and availability reasons, the platform distinguishes data with strict consistency requirements (for example, door lock status) from data with eventual-consistency characteristics (for example, a GPS position as part of a GPS data stream). In our case, a platform service provides the data tier capabilities. So, the platform can automatically scale this tier.

At the project's start, we decided to base the implementation on open-source components. We did this to

- learn more about open source's advantages and drawbacks in an enterprise environment and
- initially decrease setup costs to verify the ideas.

We implemented the platform services in Java. Each service exposes a RESTful API over HTTP[8] for interaction (REST stands for Representational State Transfer). We selected Java-Script Object Notation as the data format to support mobile applications. The business logic is deployed on a JBoss application server. We realized the data tier using HSQLDB (Hyper SQL Database) for relational data and Cassandra for nonrelational data. The platform applies Cassandra's tunable-consistency mode[9] to satisfy different consistency requirements for nonrelational data.

The Application Templates

Our platform provides fundamental functionality that each telematics application prototype can use, thereby reducing complexity for developers. However, developers will still spend significant time

- setting up and integrating the prototype's fundamental architectural components and
- implementing code that's independent of specific application functionality and could be reused the same way in every application prototype.

So, to further speed up prototyping, we introduced the application templates.

The templates provide an architecture for telematics application prototypes that are from the same

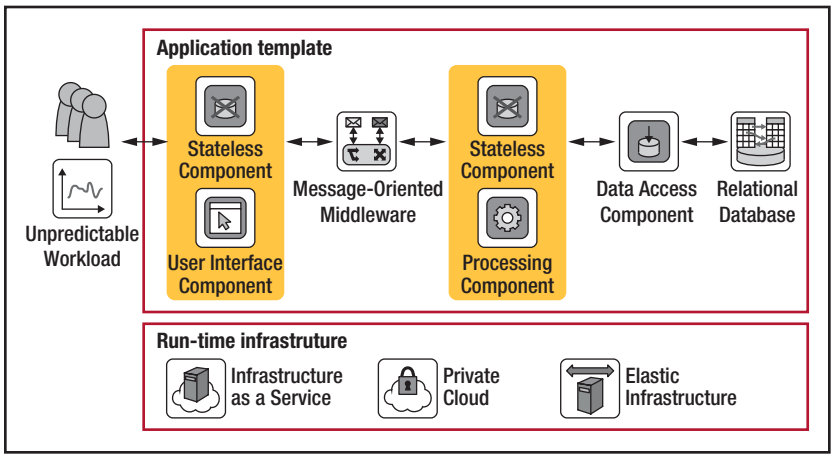


FIGURE 2. The architecture of an example application template. To provide maximum flexibility, the architecture is a modification of the Three-Tier Cloud Application pattern.

application domain and face similar requirements. An architecture by itself, however, will still require the developer to implement it. So, templates also offer the architecture's implementation. This implementation provides the required infrastructure and middleware components and is available for provisioning in one or more virtual server images.

Developers build application prototypes by provisioning templates in a prototyping environment and enriching them with individual functional code. So, they don't have to worry about finding a proper architecture and selecting, integrating, and provisioning feasible middleware products. Instead, they can focus on implementing the application functionality and generate added value faster.

The challenge when designing templates is to find an architecture that serves as a template for a possibly high number and variety of potential application prototypes. When trying to determine common requirements of application prototypes, we found we had to distinguish between two scenarios. The first scenario deals with data analysis. The vehicle frequently sends data to the connected-car platform; the data is stored persistently and then analyzed by a telematics application.

However, we focus here on the second scenario, in which telematics applications enable vehicle–user interaction. (At Daimler TSS, we frequently deal with this scenario.) With these applications, users can control a vehicle remotely, send information (such as locations) to the vehicle, or remotely view dynamic vehicle data such as the fuel state. Using this basic functionality, we can create advanced use cases that provide added value to users. For instance, by leveraging third-party APIs, users can select the most suitable gas station on the basis of the vehicle's fuel type, cruising range, current location, and destination, and the traffic volume.

To design a template for the second scenario, we first decomposed the template into three functional blocks: user interface, processing, and data access. We did this because applications in this scenario typically require user interaction and the persistent storing of application-specific data, and must handle computationally intensive tasks. When you're designing application prototypes, it's often hard to foresee how intensively they'll be used and to which user groups they'll be exposed. So, the template architecture must be flexible enough to deal with unpredictable workloads. A further design consideration is the application data state. Here, we designed the user interface component and processing component to be stateless.

The template design resulted in implementation of a modified Three-Tier Cloud Application pattern. The three tiers might be exposed to different workloads. Depending on the application prototype use case, the processing component might have to deal with computationally intensive tasks that require scaling, or it might be used less frequently than the user interface component. To achieve independent scalability of the application components, we integrated them in a loosely coupled manner, using message-oriented middleware.

We discussed integrating the processing component and data access component in a loosely coupled way as well, as proposed by the Three-Tier Cloud Application pattern. However we didn't expect the data access component to face a high workload because the applications in this scenario rely mainly on platform services for data storage. So, we used remote procedure calls to access the data access component.

We employ a private cloud offered by a VMware product suite because it provides high availability of the virtual environment by replicating the underlying physical hardware. We didn't implement

high availability at the application level because we didn't consider it necessary for prototyping. So, elasticity and resiliency configuration occurs later, when a prototype is productized. The Three-Tier Cloud Application pattern forms the basis for such extensions.

Figure 2 illustrates the template's architecture. We implemented it using Java Platform Enterprise Edition technology (see Figure 3). The user interface component implements Java Servlets and is deployed on an Apache Tomcat webserver. The Apache ActiveMQ message-oriented middleware integrates the user interface and processing components (to achieve loose coupling). The template leverages the Java Message Service (JMS) API to provide an interface with the messaging provider. The processing tier and data-handling tier use the JBoss application server. The processing component can act as a Competing Consumer pattern[10] by using Java message-driven beans. It also contains an HTTP client to utilize prototyping-platform services and access the data service. We implemented the data service as a RESTful webservice using JAX-RS (Java API for RESTful Web Services). The storage component is a MySQL database.

All templates are Maven-based Java projects stored in a central Maven repository. Ideally, more than one template is available for developers so that they can select the most suitable one for a specific application prototype scenario. For each template, corresponding virtual-machine images are available in the private cloud we mentioned earlier. These templates can be instantiated to provide the infrastructure and middleware required to deploy the template-based prototypes.

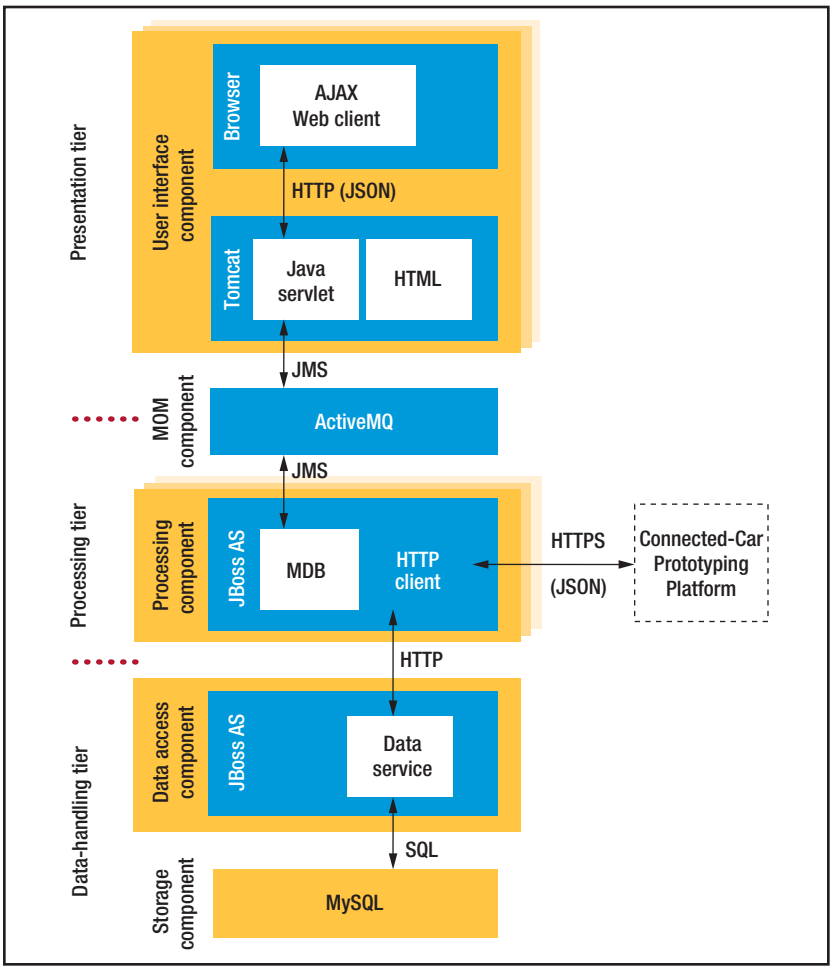


FIGURE 3. An implementation of the template in Figure 2. The implementation uses Java Platform Enterprise Edition technology. The user interface component and processing component are integrated in a loosely coupled manner using messaging, so that they can scale independently. The processing component contains the functionality to access the prototyping-platform services. AJAX is Asynchronous JavaScript and XML, JSON is JavaScript Object Notation, MOM is message-oriented middleware, JMS is Java Message Service, MDB is message-driven bean, and JBoss AS is JBoss Application Server.

We extended a framework prototype[11] to automate loading source code for templates into development environments. We further extended it to automate infrastructure and middleware provisioning and application deployment. For that, we implemented a jclouds (https://jclouds.apache.org) Maven plug-in to enable interaction with a private-cloud API. Maven plug-ins also perform middleware configuration and prototype deployment.

Discussion

We successfully used our platform to develop telematics services for a research-and-development group.

Overall, our pattern-based approach resulted in reusable functionality and guidelines that sped up development.

The current implementation is running successfully. Users can easily create services using templates. The architecture has been robust enough to accommodate new requirements without fundamental changes. For example, we added a near-real-time capability to stream data from a car to the Web for live monitoring. We extended the vehicle's connectivity unit and the platform with an offline capability that addresses cellular dead spots cars move through.

We found it's important to engage with IT stakeholders, especially IT operations staff, as early as possible to identify and avoid potential impediments. In the end, even small aspects might result in a complete showstopper owing to company policies or other corporate regulations. Some of our devices rely on non-HTTP protocols and nonstandard HTTP ports (ports other than 80 and 443) when communicating over the Internet—for example, using MQTT[12] as a transport protocol with custom ports. In an enterprise environment, the operations team will support such exceptions only if they're discussed at an early stage.

We also learned that patterns serve as a good guideline. Because they reference each other, the knowledge they provide doesn't have to be accessed in a linear form. Nevertheless, the patterns didn't address and solve all the architectural challenges we experienced. Patterns provide good hints on how to solve most challenges. However, they can still be incomplete or ambiguous with many implementation options, or they might not cover all the requirements. The following four examples illustrate such situations.

> Patterns provide hints on how to solve most challenges but can be incomplete or ambiguous.

First, in our case, the architecture serves only a little traffic at the beginning. However, it should be able to serve a rapidly growing number of vehicles and consumers—up to 5 million clients. The patterns we use enable the platform and deployed applications to scale as the number of users grows. However, additional challenges might arise—for example, if the amount of application data increases unexpectedly. So, in the future, the complete architecture might require adjustments.

The second example involves data consistency. To deal with a large amount of data (for example, a fleet of several million cars sending regular status events), the best practice is to partition the data into separate consistency levels—for example, strict versus eventual consistency. In our case, this approach requires different database technologies, which increases implementation and maintenance effort. The Data Access Component pattern reduces effort for application development and maintenance but not for database operations.

The use of different database technologies and consistency levels has led to new challenges affecting various stakeholders and supported business cases. The operations workforce must have a larger skill set to support different database technologies. Users evaluating the data must learn to acknowledge that it might not always be up to date. Finally, the business cases that such technologies support must take into account data inconsistencies. Patterns addressing these challenges seem a promising research area.

Third, the patterns cover how to use cloud offerings but not how to actually build a cloud. This is because they target application developers, not cloud providers. To design our platform, we modified the Three-Tier Cloud Application pattern, as we mentioned before. We added the central load balancer and designed the platform services as independent components. However, it would be helpful for cloud providers to have a pattern that provides initial guidelines.

The final example deals with extending the patterns to deal with wearable technology and the Internet of Things. Wearables such as smart watches are a rapidly growing market for connected services. These devices come with new interaction models and new technical characteristics. As with cars, such devices aren't always on; patterns dealing with queuing, caching, and synchronization would be beneficial. Also, one aspect of the Internet of Things is the growing number of sensors and actors. Patterns could help deal with the complexity and could address, for example, identity and con-

figuration management, or data aggregation and filtering.

Patterns don't solve all problems automatically. They can be misused if they're misunderstood. So, an experienced architect should validate the selected patterns for their applicability and perform a critical cross-check on the overall scenario.

We plan to create templates covering a variety of application scenarios and provide them to developers. Making the templates available in a central repository on the Internet will let developers select and download those that help them most in their current use case. The community will also be able to provide its own templates. Maven already proved to be a usable repository in the current setup. Its robustness and scalability in a broader setup—for example, for use by a larger group of developers—needs further investigation.

Acknowledgments

Icons used in Figures 1 and 2: http://cloudcomputingpatterns.org

References

1. "Gartner Says by 2020, a Quarter Billion Connected Vehicles Will Enable New In-Vehicle Services and Automated Driving Capabilities," Gartner, 26 Jan. 2015; www.gartner.com/newsroom/id/2970017.
2. D. Hardaware, "Buckle Up: The Connected-Car Revolution Is Almost Here," *Engadget*, 13 Jan. 2015; www.engadget.com/2015/01/13/connected-car-revolution.
3. J. Carter, "How Your Car Will Use Online Apps," *Techradar*, 24 Jan. 2013; www.techradar.com/news/car-tech/how-your-car-will-use-online-apps-1125966.
4. C. Fehling et al., *Cloud Computing Patterns*, Springer, 2014.
5. C. Fehling, F. Leymann, and R. Retter, "Your Coffee Shop Uses Cloud Computing," *IEEE Internet Computing*, vol. 18, no. 5, 2014, pp. 52–59.
6. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
7. M. Hilbert, "Architecture for a Cloud-Based Vehicle Telematics Platform," diploma thesis 3575, Univ. Stuttgart, 2013.
8. S. Allamaraju, *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*, O'Reilly, 2010.
9. "What Is Tunable Consistency?," Planet Cassandra, 2015; http://planetcassandra.org/general-faq/#0.1_data-3.
10. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2003.
11. A. Schraitle, "Provisioning of Customizable Pattern-Based Software Artifacts into Cloud Environments," diploma thesis 3468, Inst. of Architecture of Application Systems, Univ. Stuttgart, 2013.
12. *MQTT Version 3.1.1*, Oasis, 29 Oct. 2014; http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

TOBIAS HÄBERLE is a senior IT architect at Daimler TSS. Contact him at tobias.haeberle@daimler.com.

LAMBROS CHARISSIS is an IT architect at Daimler TSS. Contact him at lambros.charissis@daimler.com.

CHRISTOPH FEHLING is a research associate and PhD student at the University of Stuttgart's Institute of Architecture of Application Systems. Contact him at christoph.fehling@iaas.uni-stuttgart.de.

JENS NAHM is the manager of the Enterprise Architecture Team at Daimler TSS. Contact him at jens.nahm@daimler.com.

FRANK LEYMANN is a full professor of computer science at the University of Stuttgart and the director of the university's Institute of Architecture of Application Systems. Contact him at frank.leymann@iaas.uni-stuttgart.de.

INVITED CONTENT

Editor: **Steve Counsell**
Brunel University

Code Ownership Perspectives

The notion of a single owner or multiple owners of code has existed as long as we've been building software. Code ownership measures the extent to which a developer (or developers) "owns" a single part or multiple parts of a software system. The concept is often expressed in terms of who controls code editorship (who has modification rights). At one end of the spectrum, every contributor to a system has ownership of the code. At the other end, each contributor to a system owns only the code he or she actually wrote. The decision of which ownership model to follow (or even a midspectrum ownership model) is a topical and interesting industry issue. In this, the first instalment of this department, two prominent industry representatives discuss what code ownership means to them. —*Steve Counsell*

Code Ownership—a Quality Issue

Sigrid Eldh

CODE OWNERSHIP implies a natural sense of responsibility and pride in your work. In my experience, a collective code ownership model seems to work best when it involves two to eight people. Being a single developer and owning the code independently is too much of a risk—the danger is that you don't see your own code issues. The pair-programming approach has made that clear. A small team can easily discuss, plan, and polish the code, making the best decisions on strategies such as refactoring.

It's also worth bearing in mind that no two software systems or companies are the same. The difference between the mix of skills, experience, cultures, code "idioms," and belief systems results in individual coding styles. Some cultures tend to be more individualistic; some are more collective. So, collective code ownership might work great for one team and be lousy for another, even if we're talking about the same software product and company. Because people are different, teams and organizations differ in how they take responsibility for their quality, planning, code development, and bug corrections.

I'm a strong believer in fast, efficient feedback and correcting your own problems. This fosters the value of making a quality effort early—because it pays back. When code ownership grows to larger teams, following a collective set of common design rules must come into force—as does the notion of having "code police." This might be a necessity if the software is too large or is the result of a set of poor initial architectural decisions.

Regardless of whether you develop code individually, in small teams, or in larger teams, what matters from a quality perspective is usually the persistence to measure outcomes beyond the simple pass or fail of test cases, and to care about the quality of your own work. Equally important is to have effective ways to communicate issues to others who are directly or indirectly impacted by your code. This is where we usually fail as a development community. Follow-up on all aspects of quality improvements and code changes is of the utmost importance to achieve code quality. Leaving things half done is a recipe for failure. If we don't have time to or can't be bothered to address quality in organizations with more than one team, we soon experience code decay.

Developing thorough test suites often results in test code growing larger than

0740-7459/15/$31.00 © 2015 IEEE

the source code. Here is an area in which good architecture and common test-design-code rules could make a huge difference in efficiency. Otherwise, the test suite in an agile continuous integrated build and test quickly becomes costly, where no designer maintains test cases but adds new ones. This behavior often results in a waste of effort through overlapping test cases. Furthermore, you are led into a false sense of security through the abundance of test cases, instead of focusing on how well tested your code is. Code ownership also means ownership of the test code—and this is often forgotten.

SIGRID ELDH is a senior specialist in software test technology and researcher at Ericsson in Stockholm. Contact her at sigrid.eldh@ericsson.com.

Code Ownership—More Complex to Understand Than Research Implies

Brendan Murphy

A NUMBER of researchers, including me, have published papers on code ownership from a number of perspectives, all identifying that the greater the code ownership by individuals or organizations, the greater the code quality. Most of the research measures ownership on the basis of which developers edited the code. The logical conclusion from the studies is that a project should optimize its product development through a rigid adherence to Conway's law (an organization that designs systems is restricted in some sense by the structures of the organization itself). An underlying assumption is that only the developer who "owns" the code (understands the code's functionality and the impact of any changes on the total system) should edit the code. Another assumption is that ownership is a proxy for responsibility, whereby code owners will take greater care when altering the code because any resulting issues might reflect badly on the owner.

When I discussed these results with product teams, although they agreed on code ownership's importance, they viewed these results and assumptions as too simplistic. They had three primary issues. First, restricting the number of people who can alter specific code will create development bottlenecks. Second, restricting the developers who can alter code can result in bad coding practices, in terms of readability and development of comprehensive unit tests. Finally, several development methods organize teams around developing end-user features, rather than the product architecture, and such organization requires freedom for engineers to change any code they wish.

Further discussions indicated that ownership research should consider code complexity and other definitions of ownership. The product teams believed that any competent developer should be able to update code modules whose functionality is clearly defined and has unit tests that verify the code's functional completeness. At the other extreme, most teams would restrict the engineers who could edit an OS's memory management code. So, the importance of code ownership is probably related to either the code's complexity or its position in the overall product architecture.

Having only one person with knowledge of critical code represents a risk: if the person leaves the group or company, disruption will occur. Product teams should seek to increase the number of people who can alter the code, while managing the risk. This is often achieved through tools that manage code review. Engineers can update any code, but the code owner must agree to changes before they can be checked into the code base. So, you can identify code ownership either by determining who alters the code or identifying which engineers are responsible for signing off code changes.

Software development is complex, so a single development factor is unlikely to have a universal positive impact. Ownership research must also take into account other code and development attributes to understand ownership's overall impact on code development.

BRENDAN MURPHY is a principal researcher at Microsoft Research, Cambridge, UK. Contact him at bmurphy@microsoft.com.

VOICE OF EVIDENCE

Editor: **Rafael Prikladnicki**
Pontifica Universidade Catolica
do Rio Grande do Sul
rafael.prikladnicki@pucrs.br

Agile Compass:
A Tool for Identifying Maturity in Agile Software-Development Teams

Rafaela Mantovani Fontana, Sheila Reinehr, and Andreia Malucelli

RESEARCHERS HAVE PROPOSED many agile software-development maturity models to provide guidelines for work process improvements.[1] Most of them describe various maturity levels and the processes that teams should adopt to achieve each one.[2] A team then uses the model to assess its current level and identify the practices that would guide them to maturity.

Agile software-development methods present a challenge to using maturity models because their practices have been highly customized for specific contexts.[3] If agile teams don't usually rely on standard processes, how could they use a maturity model to identify where they are and how to move forward?

To answer this question, we researched how agile software-development teams naturally evolve (see the sidebar "Our Research Approach"). We analyzed nine teams' evolution of practices and found that the process was, as we expected, idiosyncratic. Each team adopted practices based on its circumstances and improved the practices based on the challenges it faced.

Using this research, we designed the Agile Compass, a checklist (see the Web extra at https://s3.amazonaws.com/ieeecs.cdn.csdl.public/mags/so/2015/06/extras/mso201506_AnAgile_s1.pdf) that lets agile-development teams identify where they are on the road to agile maturity without prescribing practices or adoption levels.

The Road to Maturity

We found that teams accomplished agile maturity via a dynamic evolution based on the pursuit of specific outcomes. We identified the various types of outcomes and organized them in a framework (see Figure 1).

Identifying the outcomes a team has and hasn't accomplished doesn't provide an exact roadmap for the future, as prescribed processes or maturity levels do. Instead, this process gives a general direction as to which further improvements should occur, leaving room for context-specific endeavors.

We identified seven outcome categories:

- *Practices learning* includes the learning-related outcomes teams pursue when they decide to change how they work.
- *Team conduct* describes how the team—the central role in agile maturity—evolves.
- *Pace of deliveries* describes the delivery-related outcomes the team progressively accomplishes while evolving.
- *Features disclosure* shows how the dynamics of the definition of software requirements change over time.

0740-7459/15/$31.00 © 2015 IEEE

OUR RESEARCH APPROACH

We interviewed nine agile software-development teams (see Table A). We analyzed each transcribed interview[1] to identify how agile practices evolved and are expected to evolve for each team. We then identified evidence that led to the outcomes described in our progressive-outcomes framework (see Figure 1 in the main article). For a more detailed description of the research process, see the "Agile Compass Research Approach" Web extra at https://s3.amazonaws.com/ieeecs.cdn.csdl.public/mags/so/2015/06/extras/mso201506_AnAgile_s2.png.

We developed our Agile Compass checklist based on our outcomes framework. For example, we interviewed teams and received these responses that indicated they used these approaches to develop high-level source code:

- Company A performed pair programming, refactored code, and cared about the code.
- Company B, team 2, integrated code daily.
- Company D performed pair programming, cleaned code, and refactored code.
- Company E, team 1, conducted code reviews, provided support from an architecture team, performed pair programming, defined a best-practices checklist, gave people time to study coding techniques, and encouraged them to study coding techniques.
- Company E, team 2, didn't automate deployment, refactored code with regular builds, and performed pair programming.
- Company F performed pair programming, used test-driven development, and conducted informal code reviews.

Analyzing this information and taking into account the context of each team's project and our observation of each team, we created the part of the Agile Compass (see the Web extra at https://s3.amazonaws.com/ieeecs.cdn.csdl.public/mags/so/2015/06/extras/mso201506_AnAgile_s1.pdf) pertaining to the development of high-level source code. We described this outcome by saying, "Coding is an important activity, and there are initiatives to guarantee the code is clear and robust, using best practices available." We then summed up those practices in the form of two statements to which companies could agree, if applicable: "We care about our code" and "We endeavor initiatives to guarantee our code is fine, such as reviews, refactoring, pair programming, or others."

All information we gathered that supported each outcome is in our "Map of Evidence" Web extra at https://s3.amazonaws.com/ieeecs.cdn.csdl.public/mags/so/2015/06/extras/mso201506_AnAgile_s3.png.

Reference

1. R.M. Fontana et al., "Progressive Outcomes: A Framework for Maturing in Agile Software Development," *J. Systems and Software*, vol. 102, 2015, pp. 88-108.

TABLE A

Profiles of interviewed development teams

	Team size	Company size (no. of employees)	Years since adopting agile development	Purpose for developing software
Company A	8	150	1.3	Software for the company's own use and for external customers
Company B, team 1	5	200	5	Software built on demand for external customers
Company B, team 2	7	200	6	Software packages and embedded systems
Company C	20	15,000	3	Customization or adaptation of existing software
Company D (distributed)	5	17	6	Software for the company's own use or built on demand
Company E, team 1	12	90	5	Customization or adaptation of existing software
Company E, team 2 (second site)	6	150	5	Software for the company's own use or built on demand
Company F	8	150	4	Software for the company's own use or built on demand
Company G (government organization)	11	10,000	0.4	Customization or adaptation of existing software

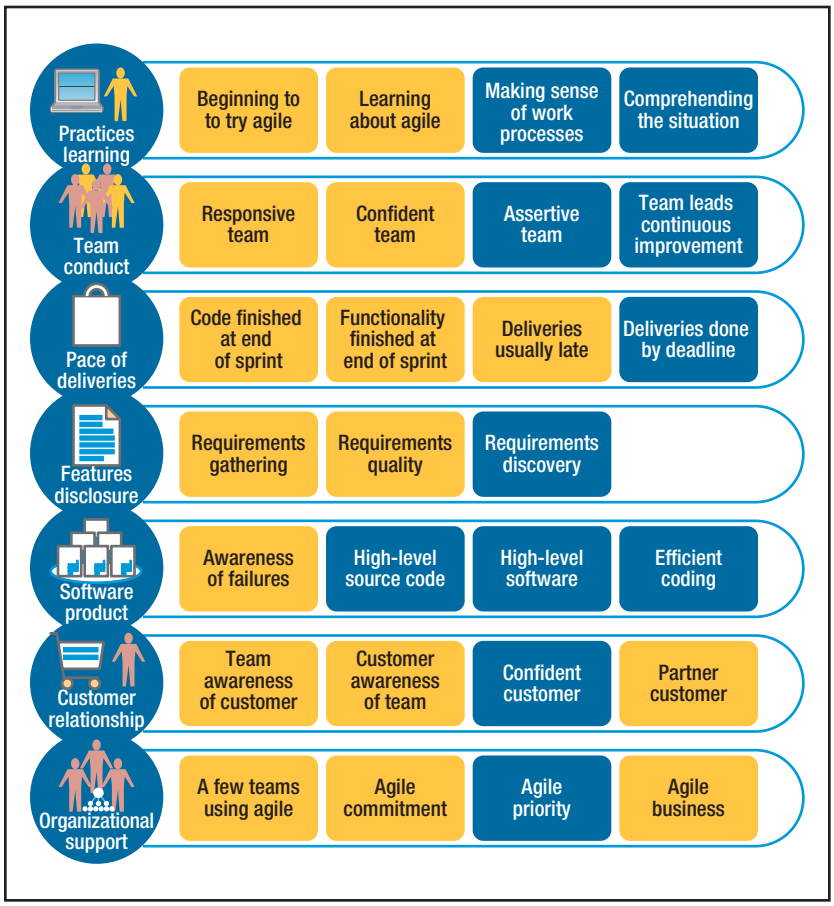


FIGURE 1. The progressive-outcomes framework. The circles represent outcome categories; the boxes are various development outcomes. The blue boxes are outcomes that indicate team maturity.

TABLE 1

Agile Compass research profile.*

Outcome category	No. of pieces of information from interviews showing development teams pursuing each outcome
Practices learning	65
Team conduct	79
Pace of deliveries	38
Features disclosure	48
Software product	83
Customer relationship	35
Organizational support	25

** The study involved nine teams (with 25 practitioners) and 27 identified outcomes.*

- *Software product* shows how the quality of the team's software evolves.
- *Customer relationship* describes the team's changing relationship with the customer.
- *Organizational support* represents how organizational support for agile development changes during the maturity process.

We lack evidence that teams either pursue all outcomes or follow a predefined sequence of outcomes in the maturing process. However, we have clear evidence of the outcomes that mature teams have achieved, which appear as the blue boxes in Figure 1. The other boxes are outcomes that teams naturally pursue during their agile evolution but that don't always characterize maturity.

Where Are We?

In the cases we studied, we identified dozens of practices that contribute to the accomplishment of outcomes (see Table 1) and used this information to develop the Agile Compass.

The Agile Compass briefly describes each outcome in an agile-development process and offers several ways to identify whether it was accomplished. Development teams can use the tool to discuss and determine which outcomes they've accomplished and which still need work, and where they must invest additional effort to improve the process.

Using the Agile Compass doesn't necessarily give a final, definitive picture of a team's progress toward maturity because the situation can change

over time. For example, when a variable—such as a team member or a technology used to build the software—changes, the team might have to start pursuing its outcomes again. In addition, once developers in a team accomplish mature outcomes, their challenge is to continuously improve work processes to sustain these outcomes in a changing business environment.

As the business environment changes, management must observe its development teams, use some simple metrics to evaluate them, and make the necessary adjustments as quickly as possible to enable continuous improvement.[4] The improvement is based not on prescribed processes but on the pursuit of outcomes. This approach places the agile team in a central role and supports the constant learning of emerging practices for specific development contexts.

References

1. M. Leppänen, "A Comparative Analysis of Agile Maturity Models," *Information Systems Development: Reflections, Challenges and New Directions*, R. Pooley et al., eds., Springer, 2013, pp. 329–343.
2. R.M. Fontana et al., "Progressive Outcomes: A Framework for Maturing in Agile Software Development," *J. Systems and Software*, vol. 102, 2015, pp. 88–108.
3. D. Bustard, G. Wilkie, and D. Greer, "The Maturation of Agile Software Development Principles and Practice: Observations on Successive Industrial Studies in 2010 and 2012," *Proc. 20th Ann. IEEE Int'l Conf. and Workshops Eng. Computer-Based Systems* (ECBS 13), 2013, pp. 139–146.
4. M. Walter et al., "From Sprints to Lean Flow: Management Strategies for Agile Improvement," *Agile Processes in Software Engineering and Extreme Programming*, C. Lassenius, T. Dingsøyr, and M. Paasivaara, eds., Springer, 2015, pp. 310–318.

RAFAELA MANTOVANI FONTANA is a doctoral student in software engineering at the Pontifical Catholic University of Paraná and a professor of software engineering at the Federal University of Paraná. Contact her at rafaela.fontana@ufpr.br.

SHEILA REINEHR is a doctoral supervisor and professor of software engineering at the Pontifical Catholic University of Paraná. Contact her at sheila.reinehr@pucpr.br.

ANDREIA MALUCELLI heads the postgraduate program in informatics at the Pontifical Catholic University of Paraná. Contact her at malu@ppgia.pucpr.br.

See www.computer.org/software-multimedia for multimedia content related to this article.

RELIABLE CODE

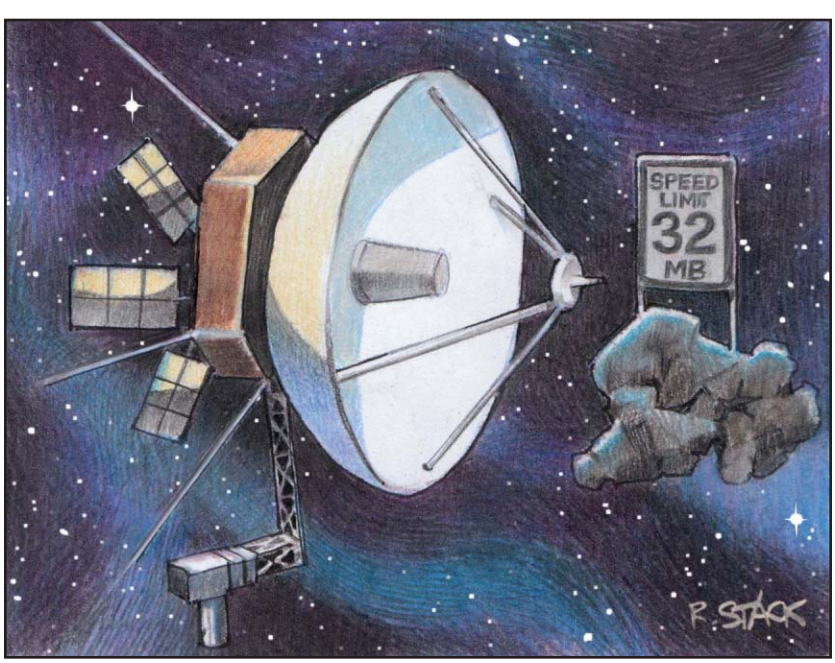
Editor: **Gerard J. Holzmann**
NASA/JPL
gholzmann@acm.org

Out of Bounds

Gerard J. Holzmann

IN SOFTWARE CERTIFICATION courses at JPL we emphasize that writing reliable code requires the development of a keen sensitivity to system bounds. In any computer only a finite amount of memory is available to perform computations, only a finite amount of time is available to do so, and every object we store and modify is necessarily finite. All resources are bounded. Stacks are bounded, queues are bounded, file system capacity is bounded, and, yes, even numbers are bounded. This makes the world of computer science very different from the world of mathematics, but too few people take this into account when they write code. Perhaps the problem is that in most cases it doesn't matter much if you're aware of the limitations of your computer because things will work correctly anyway, most of the time.

There's a great *Peanuts* strip in which Peppermint Patty explains to a schoolmate that algebra isn't really all that hard: "x is almost always eleven, and y is almost always nine," she explains patiently. In math, if x and y are both positive, then it's clear that their sum must be larger than both x and y. Not in a computer. If we use signed 32-bit integers, this property holds only if the sum is less than about two billion. Because that will be true most of the time, it's safe to say that when we increment x with a positive value, the result will "almost always" be greater than the original.



Almost Always

Assume we're maintaining a signed 32-bit counter. We start it off at zero and increment it once every second. If we keep doing that, the counter will take about 68 years to overflow. This is in fact how the number of seconds was stored in the original Unix operating system, with a clock value of zero corresponding, by convention, to 1 January 1970. This method for recording time clearly has limitations. The 32-bit Unix clock can record only a time span of about 136 years: from December 1901 (recorded as negative numbers counting seconds before January 1970) until January 2038. If we keep a Unix box running long enough, the equivalent of the Y2K problem will occur on Tuesday, 19 January 2038, when time will appear to switch back to 13 December 1901.

Fixing the Unix clock isn't hard. If we simply move the counter to a 64-bit number, the overflow won't occur for about 293 billion years. Even though this amount of time isn't unbounded, it's likely long enough, given that our solar system is expected to come to a fiery end before this counter can reach a mere five billion years.

Things change if we start counting time at a finer resolution than seconds. If, for instance, we increment our 32-bit counter once every 100 ms, a signed counter will overflow in about 6.8 years, an unsigned one in 13.6 years. If we increase the precision to one increment every 10 ms, a signed counter will overflow in 248.6 days, an unsigned one in 497.1

0740-7459/15/$31.00 © 2015 IEEE

days. Remember those numbers; we'll see them again. Table 1 summarizes them.

TABLE 1

How time resolution affects counter overflow.

If you increment a 32-bit counter once every	A signed counter will overflow in	An unsigned counter will overflow in
1 s	68.1 yrs.	136.2 yrs.
100 ms	6.8 yrs.	13.6 yrs.
10 ms	248.6 days = 0.68 yrs.	497.1 days = 1.36 yrs.

Spacecraft Time

The Deep Impact mission launched in January 2005. Its embedded controller calculated time as the number of 100-ms intervals that had elapsed since 1 January 2000. Adding 13.6 years gives us a date in August 2013, well beyond the originally intended lifetime for this mission. The spacecraft was designed to launch an impactor to collide with the comet Tempel 1 and study its composition from the resulting dust cloud. The spacecraft completed that mission on 4 July 2005.

Spacecraft often pick up additional duties after successfully completing their primary mission, provided they have sufficient resources left. On a first extended mission, Deep Impact completed a flyby of the comet Hartley 2 in November 2010. After that, it still had enough fuel for another extended mission, a flyby of asteroid 2002 GT. If all had gone well, the now-renamed spacecraft EPOXI would have reached that asteroid in 2020.

But it was not to be. The clock counter on the spacecraft overflowed on 13 August 2013, which triggered an exception. The exception tripped a reset of the spacecraft that should have put it into its safe mode. Naturally, the reset cleared everything in memory except the spacecraft clock, which meant that the spacecraft ended up in a reset cycle. Even in this desperate mode, ground controllers can do things to recover a spacecraft by issuing "hardware commands" that bypass the main CPU. However, this type of intervention must be done quickly, before the spacecraft loses track of earth and can no longer receive commands. The help didn't come in time, and the spacecraft was declared lost on 20 September 2013—killed by a 32-bit counter.

It is, alas, not the only example of integer overflow causing problems even in safety-critical systems.

Airplanes

A Boeing 787 passenger plane (the Dreamliner) has many parts, many of which have embedded controllers. Those controllers are likely to have internal clocks and count time. The embedded controllers in the Boeing's GCUs (generator control units) maintain time in 10-ms increments. The corresponding counters are stored as signed 32-bit numbers. At this point, you might want to glance back at Table 1 before reading on.

On 9 July 2015, the US Federal Aviation Administration issued an airworthiness directive (AD) to all airlines instructing them to reboot the GCUs on all Boeing 787s at least once every 248 days.[1] The directive said, "We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane." It explained, "This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs … that will overflow after 248 days of continuous power."

We already know where the 248 days came from, but it's still remarkable to see how often this type of programming flaw can occur in practice, even in systems that have passed fairly rigorous certification. The software for a commercial airplane is generally developed and tested with great care. The development process must comply with the stipulations of the DO-178B standard (and its successor DO-178C). The Boeing 787 software was certainly not written in a rush. The first 787 was shown publically on 8 July 2007, and the first flight took place on 15 December 2009. The first commercial flight was two years later. We can assume that the plane, and its control software, had been in development for a good number of years at that point. When the overflow problem was discovered in early 2015, close to 300 of the planes were in operation.

Resource Limits

It seems counterintuitive that failures in highly complex systems can have these embarrassingly simple causes. The reason might be that the really difficult design issues typically get ample attention; it's the simple stuff that can get neglected. This can explain why hitting a known resource bound can bring a system to the brink of failure. Examples are easy to find; here I describe two that were in the news fairly recently.

LightSail

On 20 May 2015, the Planetary Society launched a small spacecraft for a test flight. The mission, called LightSail, aimed to test a large solar sail. The test flight hit a snag in just two days. A mission update from 26 May described the problem: "LightSail is likely now frozen, not unlike the way a desktop computer suddenly stops responding."[2]

The cause was relatively simple. An onboard log file stored a record of all telemetry data transmitted by the craft; when that file exceeded 32 Mbytes, it crashed the flight system. Why 32 Mbytes? This wasn't explained in the press releases, but this limit matches a limit of the old FAT12 file system, which some embedded systems still use. The FAT12 design used 16-bit addresses to store sequences of 512-byte blocks in the file system, and yes, $2^{16} \times 512$ bytes comes out to 32 Mbytes.

Curiosity

Another example is much closer to home for me. The Mars Science Laboratory was designed and built at the lab where I work. It successfully landed Curiosity, a large rover, on the surface of Mars on 6 August 2012. Since then, the rover has been functioning reliably, but its software has experienced a few glitches along the way.

One of those glitches occurred almost six months after the landing, on 27 February 2013. The trigger was the sudden failure of a bank of flash memory. The rover software uses flash memory to maintain a file system for storing temporary data files, containing telemetry and images from the mission, before they're downlinked to earth. The software was designed to place the flash file system in read-only mode when something unexpected happens. So, when the bank failed, that's precisely what happened.

During normal operation of the rover, many tasks use the file system to store temporary data products. Because the flash hardware can be slow and a large amount of data must be stored, the software uses a buffering method that can temporarily store data in RAM before it's migrated to the flash file system and then downlinked to earth. All this worked perfectly, almost always.

Knowing that all resources in an embedded system are bounded, we can ask, what happens when the RAM fills up? If this system is designed properly this should happen rarely, and it shouldn't last long because the data temporarily stored in RAM will eventually drain to flash memory. So, in this rare case, the rover computer suspends the data-producing tasks until enough RAM frees up for them to resume executing.

We can also ask, what happens when the flash memory fills up? This event should be even rarer because mission managers are required to keep a substantial margin of flash memory free throughout the mission. However, if it does happen, the software is designed to start freeing flash memory by deleting low-priority files until a sufficient margin is restored. The higher-priority files that live in flash memory eventually will be transmitted to earth, and the space they occupy can be freed again.

All of that looked solid enough to pass all design reviews, code reviews, and unit tests that try to push the known resource limits. But one case wasn't tested—the one that happened on Mars in February 2013. When the flash memory is in read-only mode owing to a hardware error, no further changes can be made: it can't add or delete any more data. The RAM now slowly fills up with new data products, waiting for those data products to migrate to flash memory. At some point the RAM can't accommodate any more data. The data-producing tasks are now suspended one by one, until all activity on the rover freezes.

If this story has an upside, it's that even in this seemingly desperate case, the ground controllers recovered the spacecraft and restored normal operations within days. And our log of lessons learned grew by one more item. We can only hope that the log will be bounded, too.

Should you be worried? Programmers do, of course, know that all the resources they work with are bounded. But it can be hard to keep reminding ourselves of this sobering fact. Given the frequency with which bounds-related problems happen, it's wise to plan for them. Perform deliberate unit and system tests that reach, and try to exceed, known system limits. When using a 32-bit clock, perform tests with the clock set forward 248 days, 1.3 years, and 13.6 years, and see whether the system still works correctly. Of course you'll have a long time to think about it if you forget one of these tests, but you'll surely sleep better if you don't.

References

1. "Airworthiness Directives; the Boeing Company Airplanes," *Federal Register*, vol. 80, no. 84, 2015; http://rgl.faa.gov/Regulatory_and_Guidance_Library/rgad.nsf/0/584c7ee3b270fa3086257e38004d0f3e/$FILE/2015-09-07.pdf.
2. M. Wall, "LightSail Solar Sail Test Flight Stalled by Software Glitch," Space.com, 27 May 2015; www.space.com/29502-lightsail-solar-sail-software-glitch.html.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.

FOCUS: GUEST EDITORS' INTRODUCTION

VIDEO

Emerson Murphy-Hill, North Carolina State University

Don Roberts, University of Evansville

Peter Sommerlad, FHO / Hochschule für Technik Rapperswil

William F. Opdyke, JPMorgan Chase

This paper was prepared in William F. Opdyke's personal capacity and not at the request of JPMorgan Chase or any of its affiliates. Any views or opinions expressed herein are solely those of William F. Opdyke, and may differ from the views and opinions of JPMorgan Chase & Co., its affiliates, and its employees.

0740-7459/15/$31.00 © 2015 IEEE

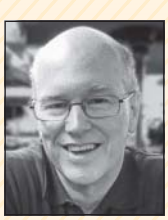
FOCUS: GUEST EDITORS' INTRODUCTION

ABOUT THE AUTHORS

EMERSON MURPHY-HILL is an associate professor in North Carolina State University's Department of Computer Science. His research interests include the intersection between human–computer interaction and software engineering. Murphy-Hill received a PhD in computer science from Portland State University. Contact him at emurph3@ncsu.edu; http://people.engr.ncsu.edu/ermurph3.

DON ROBERTS is an associate professor in the University of Evansville's Department of Electrical Engineering and Computer Science and an independent consultant. His research interests include building tools to transform source code in various ways, ranging from custom refactorings to migrating systems to a completely different language. Roberts received a PhD in computer science from the University of Illinois. Contact him at roberts@refactory.com.

PETER SOMMERLAD is the director of the Institute for Software and a professor of software engineering at FHO / Hochschule für Technik Rapperswil. His goal is to make software simpler by decremental development: refactoring software down to 10 percent of its size with better architecture, testability, and quality and functionality. To support that, his team and students created the C++ IDE Cevelop. He's a coauthor of *Pattern-Oriented Software Architecture: A System of Patterns* and *Security Patterns: Integrating Security and Systems Engineering*. Contact him at psommerl@hsr.ch.



WILLIAM F. OPDYKE is employed at JPMorgan Chase, where he focuses on agile and related technology training. His research interests include refactoring applied to agile development and legacy system evolution, and organizational and process improvements to support software innovation. Opdyke received a PhD in computer science from the University of Illinois at Urbana-Champaign; his doctoral research led to the foundational thesis in object-oriented refactoring. Contact him at opdyke@acm.org.

REFACTORING CHANGES a program's source code without changing its external behavior, typically to improve the software's design. Optimization is similar but has different goals. We've always considered refactoring akin to prime numbers; that is, any sufficiently advanced culture will discover the concept. The notion that restructuring code can improve its design can be traced back to computing's early days; Lisp practitioners and some of the Forth literature attest to that fact. However, it wasn't until 1990 that the term "refactoring" appeared in print in a research paper by William Opdyke and Ralph Johnson that described the process and identified common refactorings.[1]

Since then, the processes have been formalized (somewhat), common refactorings have received names, some operations have been automated and incorporated into development tools, and the entire concept has emerged as an academic-research area. Refactoring has been adopted as an agile practice, has been applied to help reengineer and evolve legacy code, and has been applied in other software development contexts. So, many programmers are looking for better support for these operations in their daily work.

In May 2014—nearly a quarter century since refactoring research's early days—41 people attended the Future of Refactoring seminar in Dagstuhl, Germany. Attendees came from across the spectrum: from industry to academics and from originators of the field to relative newcomers. We spent a week collaborating to deepen our understanding of what was being accomplished in refactoring and to determine the directions in which research and industry should go. One important

outcome was the idea to produce this special issue of *IEEE Software*. Another outcome was that attendees noticed that practitioners often relax the behavior-preservation aspect of refactoring. The article by Munawar Hafiz and Jeffery Overby (see the next section) examines this issue.

In This Issue

The articles we selected range from historical, exploring refactoring research's origins, to practical, exploring software developers' experiences with refactoring, to theoretical, exploring new refactoring techniques that haven't yet appeared in the wild.

"The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research," by William Griswold and William Opdyke, recounts the initial research that led to refactoring becoming a standard practice.

"Refactoring Myths," by Munawar Hafiz and Jeffrey Overby, examines popular misconceptions about tool-based refactorings.

"Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective," by Tushar Sharma and his colleagues, is an experience report about adopting refactoring techniques at Siemens Corporate Development Center India.

"Refactoring for Asynchronous Execution on Mobile Devices," by Danny Dig, describes a particular common situation: applying refactorings to convert a long-running, synchronous process into an asynchronous one.

"Refactoring—a Shot in the Dark?," by Marko Leppänen and his colleagues, analyzes interviews of 12 seasoned software architects to determine how refactoring is used in the real world.

"Database Refactoring: Lessons from the Trenches," by Gregory Vial, is an experience report of applying refactorings to restructuring a relational database in a small-to-medium enterprise.

In the Point/Counterpoint department, John Brant and Friedrich Steimann debate whether current tools are trustworthy and how much that really matters.

Whatever your interest in refactoring, there's something in this issue that will deepen your understanding of refactoring and enrich your design experience. We hope you enjoy the issue.

Reference

1. W.F. Opdyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Proc. Symp. Object-Oriented Programming Emphasizing Practical Ap-*

See www.computer.org/software-multimedia for multimedia content related to this article.

The Birth of Refactoring

A Retrospective on the Nature of High-Impact Software Engineering Research

William G. Griswold, University of California, San Diego

William F. Opdyke, JPMorgan Chase

// This article reflects on how the idea of refactoring arose and was developed in two PhD dissertations. The analysis provides useful insights for both researchers and practitioners seeking high impact in their work. //

This article was prepared in William F. Opdyke's personal capacity and not at the request of JPMorgan Chase or any of its affiliates. Any views or opinions expressed herein are solely those of William F. Opdyke and William G. Griswold, and may differ from the views and opinions of JPMorgan Chase & Co., its affiliates, and its employees.

PROGRAMMERS HAVE BEEN instinctively cleaning up, reorganizing, and restructuring their code since the first programs were written. Then, in the late 1980's, the two of us, who were computer science graduate students at the time (Bill Opdyke at the University of Illinois at Urbana-Champaign and Bill Griswold at the University of Washington), independently invented what's now called software refactoring.

Software refactoring is the systematic practice of improving application code's structure without altering its behavior. An attendee of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), where Opdyke presented the first paper to use the term "refactoring,"[1] perhaps summed it up best: Refactoring was something that he as an experienced developer naturally did to his code, without consciously thinking about it; this research made that process explicit. Our early refactoring research—and the research that followed—addressed a real-world software development challenge explicitly and systematically. Nearly 30 years later, refactoring has become a central part of software development practice. It's a core element of agile methodologies, and most professional IDEs include refactoring tools.

Here, we show that the invention of refactoring—like so many other inventions—was virtually guaranteed by the circumstances of the time. Yet, despite these enabling circumstances, the unique characteristics of the research groups at Illinois and Washington led to each focusing on unique questions regarding refactoring, resulting in complementary research results. These differences ultimately affected how each of these research contributions impacted software practice.

Although the invention of refactoring and its adoption into professional practice were practically destined, they weren't just a happy accident. Reflecting on the events surrounding the invention of refactoring, we provide advice for those who seek to create high-impact research and transition it to practice: young researchers, their mentors, and practitioners.

0740-7459/15/$31.00 © 2015 IEEE

A Brief History of Refactoring

We now each tell our story of the discovery and development of refactoring, because the details inform our insights into the nature of high-impact research. Our research paths proceeded essentially in parallel with virtually no interaction, so we present the stories as separate vignettes.

Bill Griswold's Story

Griswold began his PhD at the University of Washington in 1985 and began working with the young professor David Notkin in 1986. Notkin, a software-engineering researcher, had decided to work in the nascent subfield of software evolution.

Griswold's early research with Notkin took on the challenge that once applications were deployed, they could no longer be altered. Griswold and Notkin invented the Extension Interpreter, which let end users add functions to existing applications.[2] One thing Notkin observed about this solution was that an extended application's internal structure—its design—would inevitably be different than if it had been developed from scratch. That is, if enough function had been added, a from-scratch software design would add internal structures to accommodate this mass of functionality—perhaps new submodules, layers, or data structures.

In 1987, Tony Hoare and his colleagues' article "Laws of Programming" appeared, which introduced the idea that the semantics underlying programming languages allows programs to be treated algebraically.[3] They even showed that any nonrecursive, loop-free program could be reduced by transformation to a normal form. They also argued that skilled developers intuitively internalize these rules and apply them when they program. This suggested to Griswold that, in principle, you could algebraically transform one design of a program into any other possible design. Intrigued, Griswold began writing and transforming small mathematical programs written in the Icon programming language, and found he could make substantial structural changes systematically.

> The task of single-handedly building a meaning-preserving restructuring tool was daunting.

Griswold showed these programs to Notkin. A short time later, in summer 1988, Griswold had an internship in John Backus's Functional Programming language group at IBM Research Almaden. Over the phone, Notkin proposed this idea of restructuring programs to improve their design as a thesis topic. "You can call me back in two weeks and tell me it was your idea," he said. Griswold did.

When Griswold returned from the internship, he proposed that the restructurings should be meaning-preserving. Noktin expressed doubt, but Hoare's influence, plus the peace of mind that such a regimen would offer developers, had Griswold convinced.

The question then turned to how to build a tool to support meaning-preserving restructuring. Tools are important to software engineering research, both practically, in that they aid developers in a tedious, error-prone activity, and theoretically, in that the automation serves as a proof that the concepts have been sufficiently formalized to be executed by a computer. Automated semantic analysis is the domain of compilers. Indeed, a compiler is a kind of meaning-preserving program transformation tool.

Looking to this field for insight, Griswold talked to others in his department, such as Robert Henry, and was pointed to program dependence graphs (PDGs), a relatively new data structure for directly representing a program's semantic dependences. Griswold also learned that there had been remarkable recent progress in precise dependence analysis, notably in the presence of pointers. This gave further hope that building a meaning-preserving restructuring tool was technically feasible.

Although the nascent project itself was unfunded, Griswold had received an IBM Research graduate fellowship. However, the task of single-handedly building a meaning-preserving restructuring tool was daunting. Griswold focused on supporting the smallest, simplest programming language possible that demonstrated the essential features: side effects, pointers, arrays, records, branching, looping, and recursion. Dealing with syntax was also to be avoided. The idea, then, was to refactor a subset of imperative Scheme, a modern Lisp dialect.

As luck had it, Griswold was pointed to Jim Larus's work on the Curare parallelizing compiler, which

employed PDGs, applied (and in some cases introduced) the latest results in dependence analysis, and compiled Scheme programs.[4] Griswold contacted Larus and soon had the remarkably complete Curare compiler, written in some 25,000 lines of Common Lisp, running on the just-released DEC 3100 workstation (24 Mbytes of RAM, 16 MHz, and 11 million instructions per second). Much code would remain to be written, and some maddening debugging of sophisticated algorithms would be required. However, finding a single compiler that employed PDGs, with the latest algorithms, for Scheme was a massive stroke of good fortune.

Intellectual control of Restructure, as Griswold came to call the tool, was a primary research question, as well as a practical consideration. Reasoning about correctness was daunting, as was keeping track of the tool's numerous functionalities and data structures, from PDGs to abstract syntax trees to the user interface.

For correctness, Griswold took a page from "Laws of Programming" and formulated a set of meaning-preserving algebraic transformation rules for PDGs (for example, transitivity and distributivity).[5] The result was elegant. For one thing, a PDG is a surprisingly straightforward, essentially functional-programming representation of a program. Also, represented as PDGs, the rules were visually comprehensible graph-to-graph transformations. Even if the program representation in a restructuring tool wasn't a PDG, the rules effectively communicated the semantic requirements of meaning-preserving restructuring.

To organize Restructure's functionalities and data structures, Griswold fashioned a three-column multilayered software architecture that kept source-based and dependence-based data structures in sync.[6] This architecture also provided a powerful "virtual machine" for high-level programming of meaning-preserving transformations.

The resulting command-line tool automated a modest catalog of meaning-preserving transformations, many familiar today, such as Extract Function and Rename, and some not so familiar. The latter were necessary for carrying out a complete restructuring without intervening manual edits. They demonstrated that you can carry out a sophisticated rearchitecting of an application using just meaning-preserving transformations. Hoare's theoretical ideas indeed had practical implications.

The transformations, like Hoare's and others seen previously, were guarded, such that if all the necessary preconditions for a meaning-preserving transformation were met, the transformation could proceed. Griswold conceptualized such a transformation as a developer's local edit compensated by additional global transformations that would ensure a meaning-preserving result. However, a meaning-preserving transformation wouldn't actually be implemented that way until his WitchDoctor result more than 20 years later. Instead, a meaning-preserving transformation was a composition of local building-block transformations that together preserved meaning if the guard check succeeded. Each transformation needed to be passed at least one program element—for example, a code block or variable reference—to specify what was to be restructured.

Griswold finished his dissertation in 1991[7] and continued his research on what was now called refactoring (via Opdyke) as an assistant professor at the University of California, San Diego. One line of research was in user interface support, which initially explored form-based user interfaces like those used predominantly today. This research then moved to graphical interfaces—particularly the star diagram[8]—when it became clear that developers needed help seeing their source code's latent structure.

The other line of research concerned efficient PDG-supported restructuring, enabled by incrementally updating the underlying PDG representation using the algebraic transformation rules.[9] In 2012, Griswold published his research on WitchDoctor, in which the source code editor detects in real time that a developer is manually performing a refactoring, and offers to complete it.[10]

Bill Opdyke's Story

Opdyke worked for several years at Bell Labs before pursuing his doctorate from 1988 to 1992. As an undergraduate, his software assignments were mostly small projects that took him days or weeks, individually or as part of a small team. In contrast, the Bell Labs software development projects involved hundreds or thousands of staff, supporting products with lifetimes of 10 to 20 years or more. These differences weren't just quantitative; there were qualitative differences between evolving software and creating it from scratch.

Research that focused on how to change existing software was rare during the mid-1980's; much of it focused on "fresh start" projects: how to build software better from scratch. Nonetheless, there were several research areas that addressed software change—including knowledge-based software engineering, program transformation, and

database schema evolution—that influenced Opdyke's research.

In industry, software costs were growing dramatically. As Fred Brooks noted in his landmark 1986 article "No Silver Bullet," there was no definitive solution that had yet been invented, only incremental progress.[11] Some in the software industry felt that software process and process improvements might be a high-leverage area. Recognizing software evolution's growing importance, Brooks himself suggested "Grow—don't build—software" as a promising approach. In industrial research circles, software reuse—and application frameworks—were increasingly popular. People generally agreed that both were good, but it wasn't clear how to effectively leverage either in ways that achieved practical benefits.

Opdyke's industry experience led him to want to pursue research that was technically interesting and addressed an area of need for the kinds of large software projects he observed at Bell Labs and prior employers. When he took Ralph Johnson's course in object-oriented programming and met with Johnson, he could see areas of similar interest, although coming from differing backgrounds and perspectives. Opdyke's prior focus on software evolution and reuse was largely at the macro level, influenced by the very large-scale software systems written in C and C++ that he'd seen. In contrast, much of Johnson's research had a Smalltalk focus, influenced partly by the research team at Tektronix that included Kent Beck, Ward Cunningham, and Rebecca Wirfs-Brock. Johnson's perspective on reuse appeared to Opdyke to be at the micro level, helping an individual programmer improve the design of his or her code. But, over time, Opdyke came to appreciate how you could achieve significant functional changes and system-level design improvements by composing a series of more primitive changes.

Several months before beginning research with Johnson, Opdyke had done a literature review related to software reuse. He came upon Peter Deutsch's paper, which noted that the most valuable things to reuse were the factoring of functions and the designs of interfaces.[12] These, not code, were the costlier artifacts to "get right" up front and the most valuable to reuse downstream. How do you arrive at well-factored, reusable, and evolvable software? On multiple occasions, Johnson commented to Opdyke that "Software is not reusable until it has been reused." The key insight was that a reusable abstraction emerges from experience with multiple concrete instances in an application domain. One practical challenge of such reuse is that in the resulting code, the reusable pieces are intertwined with those specific to the application. Separating out the reusable pieces requires restructuring.

In 1989, building on Johnson's ideas that reusable application frameworks emerged from use, Opdyke investigated software change in application framework development. Several such projects were underway at the University of Illinois. For one, the Choices file system framework, PhD student Peter Madany had kept a series of snapshots of the framework encompassing several months of its development. From these snapshots, Opdyke mined several key software changes, many of which were behavior-preserving restructurings. The behavior-preserving nature of these restructurings became a key focus of Opdyke's research. Studying changes in other framework projects at Illinois extended the catalog, as did publications such as Roxanna Rochat's paper on a good Smalltalk programming style.[13]

Opdyke and Johnson decided to focus on restructuring C++ programs, partly because of the connection between C++ and Bell Labs (which sponsored Opdyke's graduate studies). In addition, the software development community would take the results of restructuring C++ programs more seriously than the results of restructuring programs in less mainstream languages.

Early on, Opdyke called his research "supporting the process of change in object-oriented programs." By mid-1990, Opdyke and Johnson authored "Refactoring: An Aid in Designing Application Frameworks and Evaluating Object-Oriented Systems."[1] This paper described their research project as the "Software Refactory Project" (a play on the then-popular term "Software Factory"). Johnson recalls hearing Tektronix researchers use the term "refactoring"; however, at the time,

> The behavior-preserving nature of these restructurings became a key focus of Opdyke's research.

they had neither defined a catalog of refactorings nor built a tool to automate them. Opdyke recalls a series of discussions with Johnson that eventually led to their selecting the term "refactoring" for their research, using the term as both the name of the restructuring process and the generic name of an individual restructuring.

On the basis of Opdyke's initial analysis, a small set of refactoring transformations emerged—several fairly broad restructurings across classes, and others fairly primitive such as renaming a variable. Two issues arose. What refactorings should you apply in a given situation? How, if at all, can you safely apply a refactoring in a given situation? Opdyke and Johnson agreed that Opdyke should focus on the second issue.

The safety issue—that a refactoring shouldn't break working code—was recognized as critical to industrial adoption. It also raised other interesting research issues. Object-oriented developers had pride in workmanship, a willingness to restructure their code after getting it to work, in ways that improved its quality and made it easier to extend. However, there were practical considerations. A refactoring that broke existing code was unlikely to be applied again, especially in a language such as C++, in which the cost to fix included costly recompilation.

The safety issue—that a refactoring shouldn't break working code—was recognized as critical to industry adoption.

What did it mean to preserve a program's behavior when it's refactored? Intuitively, it seemed obvious: the program should behave the same after refactoring as before. On closer inspection, the refactoring of object-oriented programs had to take into account a range of issues such as scoping and typing. Generally, the complexities weren't so much in changing a piece of code in isolation. Rather, they were in ensuring that all the other parts of the software that referred to the piece of code were either safeguarded from the change or changed in a compatible manner (for example, changing all the references to a variable whose name changed).

Opdyke took a computationally conservative approach to preserving program behavior, which can perhaps be best described with an example. Suppose a function has a conditional branch. Can you remove that branch without changing the program's behavior? You can if you know that the branch's condition will never test true, but generally you can't. The choice then is to either leave that branch in or remove it and then reinsert it if the code breaks as a result. Opdyke chose the former approach, allowing only a provably correct refactoring, even at the expense of disallowing a refactoring that might be safe in some situations. Although code equivalence is computationally undecidable in general, it's decidable for many common cases, so the conservative approach was conjectured to not be too limiting.

As Opdyke's research proceeded, he found that database research on schema transformation provided a model he could apply to program refactoring. Program properties that needed to remain invariant were analogous to database integrity constraints that need to be preserved. Opdyke focused on defining the set of invariant program properties and specifying for each refactoring the conditions under which it would maintain these properties. He also specified the resulting conditions after a refactoring had been applied, in ways that allowed the more primitive refactorings to be strung together to accomplish some fairly powerful meaning-preserving restructurings. The complexities were hidden from the user; the tool would check whether the requested refactoring could be performed safely and, if so, would apply it.

Opdyke completed his dissertation in 1992,[14] followed by other refactoring-related papers coauthored with Ralph Johnson in 1993 and with Brian Foote in 1994. At the 1992 ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92), Opdyke introduced Martin Fowler to his refactoring research. Opdyke and Don Roberts presented refactoring tutorials in 1995 and 1996. Related research continued at the University of Illinois, including Don Roberts and John Brant's influential work on the Smalltalk Refactoring Browser.[15]

The 1990s also saw the emergence of Java, and IDEs supporting Java and other languages began incorporating refactoring capabilities. Martin Fowler's refactoring book was published in 1999,[16] with Opdyke as a contributing author. Kent Beck's book *Extreme Programming Explained* appeared the same year.[17]

Extreme programming included refactoring as a core practice, which Beck had brought forward from his time in the Smalltalk group at Tektronix. In time, refactoring would be broadly adopted in agile practices.

Refactoring Research Today

According to Google Scholar, as of the writing of this article, at least 930 papers had been published with "refactoring" in their title, and refactoring had become a track in several major software-engineering conferences. Refactoring is a top-level menu in many popular IDEs. We estimate that refactoring-related books have collectively sold more than 100,000 copies.

Still, challenges abound in refactoring. For example, as common as refactoring is, Emerson Murphy-Hill and his colleagues showed that developers mostly refactor manually using the program editor, because refactoring-tool interfaces tend to be cumbersome and hard to learn.[18] Researchers such as Max Schäfer and Oege de Moor showed that developing traditional guarded transformations can be cumbersome at best and that sometimes it's best to check for correctness after the transformation.[19] Refactoring research's scope is growing, too, beyond the aim of improving design to improving other nonfunctional qualities such as performance, responsiveness, and usability. A second refactoring renaissance is underway.

Lessons for Researchers and Practitioners

Looking back on these two histories, we learned much about the conduct of high-impact research. The following advice applies to researchers and practitioners alike, but often in different, complementary ways.

You Won't Be Scooped

PhD students often fear they'll discover midway through their doctoral research that someone else had previously published their key idea. Trolling our libraries' shelves, microfiche, and CD-ROMs, we did find antecedents, but each with key differences from our research. Most notable, perhaps, was the research on database schema restructuring. Data representation, not code, was restructured, but it required transforming queries to work on the new schema. An added feature of some of this research, not addressed in the refactoring literature as far as we know, was restructuring existing database tables to reflect the new schema.

We were relieved, only to later discover each other's research, as well as that of Eduardo Casais,[20] striking fear into our hearts. Johnson advised Opdyke not to worry; on the contrary, when others are working on similar research, it helps to bolster the case for the research area's importance. As David Parnas observed,

> *We must not forget that the wheel is reinvented so often because it is a very good idea; I've learned to worry more about the soundness of ideas that were invented only once.*[21]

Notkin advised Griswold to look into the details of Opdyke's research because in the systems area, works that appear similar at first often end up addressing unique aspects of the problem or presenting rather different solutions. This was indeed the case. Although there were many common elements, Griswold had gone deep into the meaning-preserving aspects of refactoring, whereas Opdyke had focused on the unique opportunities in refactoring object-oriented programs, with transformations such as Push Up Method and Push Down Method. Griswold experienced this again with his WitchDoctor work, which came out in the same conference as the similar BeneFactor tool:[22] same goal, different and complementary approaches.

The lesson for young researchers is, don't be too concerned with competition.

The lesson for young researchers is, don't be too concerned with competition. Your unique perspective and abilities will lead you to unique contributions. In some cases, joining forces to collaborate might be the way to go—especially collaboration between a researcher and a practitioner—because of the unique perspectives and resources each brings to the table.

See the World as It Is, Then Plan for Luck

Although the idea of refactoring, at first blush, might appear to be a bolt out of the blue, some important preconditions allowed that idea to occur to us.

For Opdyke, the difference between the dominant research issues

of the day, which focused on fresh-start development, and the day-to-day maintenance of existing systems at AT&T was jarring. At the time, software maintenance was considered a software development phase, as though it were something unfortunate that would eventually pass, like adolescence or a gallstone. But that wasn't Opdyke's experience: the maintenance phase at AT&T and other large companies could consume more than 90 percent of a project's cost and lifetime. This contrast pointed the way for Opdyke's insights.

> Failures unequivocally convey that the model underlying the hypothesis is wrong.

Griswold's circumstances were different. He was a young graduate student who had never worked outside academia. But fortunately, Notkin had anticipated the future and had fashioned a research program in software evolution. Griswold, being relatively inexperienced, was largely unaware of the prevailing paradigm of software-engineering research and its constraining vocabulary. For him, chancing on Hoare's paper, playing with the ideas as an off-task activity, and sharing the results with his advisor put things in motion. Notkin saw the larger potential of Griswold's play, and an idea was born.

Out of these experiences, we derive four insights. First, see the world as it is, not just through the narrow lens of the prevailing paradigm. Our unique paths to seeing the world as it is teaches that more than one way exists to achieve clear seeing, but the common insight is to find a way to take off the rose-colored glasses.

Second, plan for luck. Although we felt lucky to have stumbled upon the refactoring idea, it wasn't dumb luck. We had put ourselves in (and had been put into) a position to be lucky: exposure to the "real world," reading widely, playing, and sharing our ill-considered ideas with others.

Third, from Notkin's willingness to listen to Griswold, we advise the corollary: listen to the young researchers. Often, the new and less experienced researchers have the new ideas because they aren't yet fully versed in, or vested in, the prevailing paradigm. But likewise, they might not recognize the importance of their musings.

Finally, we urge practitioners to make the real world visible to the researchers who need to experience it. Talk to them at conferences, invite them for sabbaticals, and publish case studies.

Build on—and Challenge—the Great Ideas

Griswold, knowing that Hoare was one of the great minds of computer science, had taken an interest in his laws of programming. Although the ideas were theoretical, Hoare had encouraged understanding them from a practical perspective. This led Griswold to think, "Well, if Hoare is right, then I should be able to do this on real programs."

Similarly, Johnson encouraged Opdyke to challenge big ideas concerning software reuse. Sometimes an idea pans out; sometimes it doesn't. Either way, research insights are generated. And because these investigations are tied to the greats and their ideas, the resulting research has currency in the community, whether it builds on or challenges the current paradigm.

Play, and Reflect on Play

The impetus for Griswold's ideas resulted from, essentially, fooling around. It took some trust to share his experiments with Notkin because this play wasn't part of his research. But as we often see in the stories behind high-impact research—such as the invention of Mosaic, the first Web browser—off-task behavior and play were important elements.

So, we advise researchers and practitioners to play and reflect on play. Research advisors should encourage play. Notkin employed tactics such as asking what was on Griswold's mind, asking what he'd been up to, calling him "boss," and the like.

Learn from Failure

A variant of play practiced by Opdyke when he derived a catalog of changes from an existing project leads us to recommend that you experiment and listen to the data. Don't just run an experiment to validate a system, idea, or yes-or-no question. Learn from the inevitable surprises or failures, attending to the result's qualitative aspects. Why did it come out this way? Why did it fail? Failures and deviations often imply something wrong with the prevailing paradigm, thus helping you to see the world as it is.

Indeed, a common lesson from the broader engineering literature is that failures are more valuable than successes. Successes weakly confirm

the hypothesis—the assumption is that the causality implied by the hypothesis was at play. But failures unequivocally convey that the model underlying the hypothesis is wrong. By listening to the data, you can discover deviations from the prevailing paradigm or model, pointing to new, paradigm-shifting research. Practitioners can help by giving researchers access to their real-world data.

Learn to Speak Others' Language, and Understand Their Problems

Opdyke had the unique advantage of being an experienced practitioner before beginning his PhD research. This opened the door to impact on practitioners that wasn't as readily available to Griswold. Still, when Opdyke returned to industry after completing his PhD, he often encountered unexpected resistance. He learned that because developers were working in a resource-constrained environment, they felt they couldn't afford a large spin-up cost on an uncertain outcome. Opdyke had to develop effective training, address concerns about risks, and show that refactoring could produce immediate benefits, all with little overhead.[16]

The lesson here is that you need to understand others' problems. But to do so, you must first learn to speak their language. Once you understand that someone else's problems aren't your problems, you'll appreciate their point of view. Different contexts yield different cultures, and careful listening bridges the chasm. As Opdyke's experience shows, this advice applies not only to academic researchers but also to any change agents in a corporate organization: tooling groups, process improvement organizations, managers, architects, and so on.

ABOUT THE AUTHORS

WILLIAM G. GRISWOLD is a professor of computer science and engineering at the University of California, San Diego. His research interests include software engineering, ubiquitous computing, and educational technology. Griswold was a pioneer of software refactoring. Later, he built ActiveCampus, an early mobile location-aware system. His CitiSense project is investigating technologies for low-cost ubiquitous real-time air quality sensing. Griswold received a PhD in computer science from the University of Washington. He's a member of the IEEE Computer Society and ACM and a former chair of ACM SIGSOFT. Contact him at wgg@cs.ucsd.edu.

WILLIAM F. OPDYKE works at JPMorgan Chase, where he focuses on agile and related technology training. His research interests include refactoring applied to agile development and legacy system evolution, and organizational and process improvements to support software innovation. Opdyke received a PhD in computer science from the University of Illinois at Urbana-Champaign; his doctoral research led to the foundational thesis in object-oriented refactoring. Contact him at opdyke@acm.org.

In 1987, someone wise to the reality of software evolution would have had to advise, "Design for change." But, as good as software designers might have been, they weren't omniscient about the changes that needed to be anticipated.

Today, the dictum can be stated more flexibly: "Plan for change." That is, with mature refactoring tools and software processes such as agile development, a project team can now more freely choose when and where to invest in software design. Designers are no more omniscient than before, but refactoring allows investing early only in design that will surely pay off down the road—say, an API that will be used in many subsystems—while delaying other design investments until the issues become clearer. And when those non-omniscient designers make mistakes, refactoring enables cost-effective recovery.

In short, refactoring has helped deliver on Fred Brooks's vague advice of "Grow—don't build—software." Today, others are speaking similar truths to the issues of the day, challenging us to invent the next great technology and transfer it to practice. The stories and advice relayed here provide one possible map to the territory. An overarching lesson is the importance of a wide bridge between research and practice.

Acknowledgments

We thank our PhD advisors, David Notkin (deceased) and Ralph Johnson, for their life-changing mentorship and friendship. We also thank IBM Research and Bell Labs for their financial support. Bill Griswold thanks his many master's and PhD students who contributed results in refactoring, including Robert Bowdidge, Macneil Shonle, and Stephen Foster. Bill Opdyke thanks those who encouraged and supported him when he pursued his doctoral studies after several years in

industry, and those at the University of Illinois and elsewhere who continued refactoring-related research. Finally, we thank the anonymous reviewers for their detailed comments, which significantly improved this article. US National Science Foundation grant CCF-1423517 partly supported the writing of this article.

References

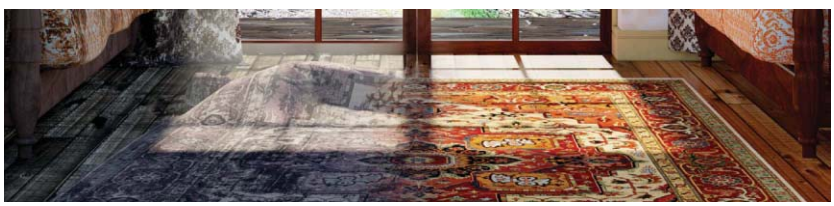
1. W.F. Opdyke and R.E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," *Proc. 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications* (SOOPPA 90), 1990, pp. 274–282.
2. D. Notkin and W.G. Griswold, "Enhancement through Extension: The Extension Interpreter," *Proc. Symp. Interpreters and Interpretive Techniques* (SIGPLAN 87), 1987, pp. 45–55.
3. C.A.R. Hoare et al., "Laws of Programming," *Comm. ACM*, vol. 30, no. 8, 1987, pp. 672–686.
4. J.R. Larus and P.N. Hilfinger, "Restructuring Lisp Programs for Concurrent Execution," *Proc. ACM/SIGPLAN Conf. Parallel Programming: Experience with Applications, Languages, and Systems* (PPEALS 88), 1988, pp. 100–110.
5. W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring," *ACM Trans. Software Eng. Methodology*, vol. 2, no. 3, 1993, pp. 228–269.
6. W.G. Griswold and D. Notkin, "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool," *IEEE Trans. Software Eng.*, vol. 21, no. 4, 1995, pp. 275–287.
7. W.G. Griswold, "Program Restructuring as an Aid to Software Maintenance," PhD diss., Dept. Computer Science and Eng., Univ. of Washington, 1991.
8. R.W. Bowdidge and W.G. Griswold, "Automated Support for Encapsulating Abstract Data Types," *Proc. 2nd ACM SIGSOFT Symp. Foundations of Software Eng.* (SIGSOFT 94), 1994, pp. 97–110.
9. W.G. Griswold, "Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool," *Proc. 1st ACM SIGSOFT Symp. Foundations of Software Eng.* (SIGSOFT 93), 1993, pp. 42–55.
10. S.R. Foster, W.G. Griswold, and S. Lerner, "WitchDoctor: IDE Support for Real-Time Auto-completion of Refactorings," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 222–232.
11. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987, pp. 10–19.
12. L.P. Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 System," *Software Reusability*, T.J. Biggerstaff and A.J. Perlis, eds., 1989, pp. 57–71.
13. R. Rochat, *In Search of Good Smalltalk Programming Style*, tech. report CR-86-19, Tektronix Laboratories Computer Research Lab, 1986.
14. W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD diss., Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.
15. D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, 1997, pp. 253–263.
16. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman, 1999.
17. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Longman, 1999.
18. E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Software Eng.*, vol. 38, no. 1, 2012, pp. 5–18.
19. M. Schäfer and O. de Moor, "Specifying and Implementing Refactorings," *Proc. 2010 ACM Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 10), 2010, pp. 286–301.
20. E. Casais, "An Incremental Class Reorganization Approach," *Proc. European Conf. Object-Oriented Programming* (ECOOP 92), Springer, 1992, pp. 114–132.
21. D.L. Parnas, "Software Aging," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), 1994, pp. 279–287.
22. X. Ge, Q.L. DuBose, and E. Murphy-Hill, "Reconciling Manual and Automatic Refactoring," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 211–221.

FOCUS: REFACTORING

Refactoring Myths

Munawar Hafiz and Jeffrey Overbey, Auburn University

// Refactoring myths are popular misconceptions about tool-based refactoring—about the tools' intent, the principles they follow, their robustness, and support for them. //



MYTHS ARE STORIES that don't convey pure truth, even if they do contain some truth. They aren't always harmful—something to be debunked. Actually, they're powerful, often thought of as serving a profound purpose by explaining the human experience. Myths let historians peek into the minds of a generation to understand their thoughts.

Refactoring is one of several groundbreaking ideas that have emerged from the software-engineering community in the last 30 years. Developers have embraced it, thanks partly to the availability of automated tools. In fact, the goals of refactoring as a design activity are often conflated with the capabilities of automated-refactoring tools. This is due to legacy: refactoring was conceived; the tools followed. During this process, ideas have become myths. Some of these arise from the original definition of refactoring, whereas others derive from the perception of how tools should be used (but not necessarily from how they are used). The myths don't reflect the reality of what refactoring tools can do, or perhaps even what they should do.

Here, we address four common myths about tool-based refactoring. We identified them from anecdotal experience, our experience working on automated-refactoring tools, and the experiences cited by other researchers and practitioners. The first myth is about the tools' intent, the second is about the principle on which they're based, the third is about their robustness, and the fourth is about the development environments supporting them.

Interestingly, none of the myths are completely false; they all contain elements of truth. We aim to illustrate the subtle ways in which they aren't true. These subtleties provide insight into the fact that how we discuss refactoring often differs markedly from how developers actually use refactoring tools. Understanding these realities can help us understand how to improve tools that support automated refactoring—and other source code transformations—for the next generation.

The Myth about Intent

This myth is that refactoring tools are useful because they help automate tedious design changes.

This statement is true, but its scope is too narrow; it doesn't accurately characterize what developers do with the tools.

Refactoring is certainly a design activity. Many developers learned refactoring from Martin Fowler's book, *Refactoring: Improving the Design of Existing Code*.[1] Note the word "design": the book points out that the most important reason to refactor is to improve the design of software. Joshua Kerievsky's *Refactoring to Patterns* focuses on applying a series of refactorings to make design-level changes.[2] Extreme Programming also employs refactoring in the context of changing design.[3]

Refactoring tools can make several design-level changes. They can move classes up and down in a class hierarchy, encapsulate data, replace classes with interfaces, and even migrate class libraries.[4] Developers make such changes with refactoring tools ... sometimes.

0740-7459/15/$31.00 © 2015 IEEE

In day-to-day development, a developer's focus is usually smaller: writing one method, class, or package. Developers use refactoring tools to rename local variables and private methods, extract functions, and extract and inline local variables. (Emerson Murphy-Hill and his colleagues confirmed that these refactorings are used more than the design-level refactorings we just mentioned.[5]) These have more to do with coding style than design.

So, refactoring as a tool capability isn't primarily about automating and making grand design changes. It's about automating tedious code changes, some of which impact design. A developer conceives of an end game—What should this code look like?—and the tool provides an expedited, reliable mechanism to achieve it. If the developer can't predict what the tool will do, or if its changes aren't what she wants, she won't use it.

Predictability is important because the developer must remain in control. When a tool's code changes are straightforward and easy to understand, this gives the developer a sense of control: the tool is simply automating changes he was going to make anyway. Although researchers will continue to advance refactoring tools' capabilities, we believe predictability will be a key factor distinguishing refactorings that are widely adopted from those that aren't.

> Predictability will be a key factor distinguising refactorings that are widely adopted from those that aren't.

Design is the developer's responsibility; the tool's benefit comes from its ability to help developers achieve a desired outcome, whether it's at the design or implementation level. So, a refactoring tool's goal should be to automate small, useful, tedious changes. Indeed, those are the most predictable changes. The developer can then compose them to create larger, design-level changes.

This brings us to another myth.

The Myth about Principle

This myth is that a refactoring tool's key benefit is its ability to guarantee behavior preservation.

Behavior preservation is indeed a critical part of the definition of a refactoring. Transformations that aren't behavior-preserving aren't refactorings.[1] Nevertheless, in terms of utility to developers, behavior preservation is secondary; developers often proceed with an automated refactoring even when it changes a program's behavior.[6]

It's difficult to give a general characterization of what it means to preserve the behavior of a program that doesn't even compile, yet Eclipse allows refactorings on programs that don't compile. For example, if an interface or a method signature has changed, but some references haven't yet been updated, it's still possible to rename identifiers or extract methods. In our experience, this is useful, especially when the semantic errors don't impact the portion of the code affected by the refactoring.

We surmise that developers think of refactorings operationally. The Rename refactoring is a kind of scope-aware find-and-replace. Encapsulate Variable is an even smarter find-and-replace. Extract Method cuts lines of code, pastes them into a new method, and replaces them with a method call.

From the developer's perspective—with this operational view of refactorings—preconditions that enforce behavior preservation are cautionary. The transformations might be useful even when the preconditions aren't met—for example, when the code contains errors. Even if the original program is correct, a refactoring can still break the program, sometimes unintentionally. Rename might break programs that use reflection. Extract Method can break (pathological) programs that access their own stack trace. Still, these refactorings are useful.

A pragmatic view is to relax the requirement of behavior preservation. A behavior-enhancing transformation is similar to a refactoring but is intended to modify the program's behavior in a small, well-defined way. For example, a Safe Library Replacement behavior-enhancing transformation replaces unsafe library functions (such as `strcpy`) in a C program with safe library functions to prevent buffer overflow vulnerabilities.[7] The modified program retains the original program's behavior except when an attack vector is present; the attack vector no longer triggers the vulnerability. Researchers have also used library replacement approaches to replace sequential libraries with parallel libraries to improve a program's performance.[8]

Eclipse's Java development tools

(JDT) provide refactorings but also provide source code transformations that don't preserve behavior. The Organize Imports refactoring adds and removes `import` statements in a Java program, on the basis of what classes are actually referenced. Another feature can generate `equals` and `hashCode` methods. Many compiler errors come with "quick fixes"; for example, if a variable is used but not declared, Eclipse can infer its type and insert a declaration. These aren't refactorings; nevertheless, they're useful. This reinforces the fact that source code transformations have utility, even without guarantees of behavior preservation.

Developers benefit from tools that predictably automate tedious code changes. Refactorings are an important type of change, but transformations that don't preserve behavior can be equally useful. As long as they're straightforward and making changes correctly, they can be a useful complement to refactorings. This correctness issue brings us to the third myth.

The Myth about Robustness

This myth is that refactoring tools are robust.

Robustness is a desirable property of software development tools—refactoring tools included. Developers won't use tools that seem unreliable. So, the widespread use of refactoring tools speaks to their apparent reliability. However, they aren't error-free. They work just well enough to be useful, and they break in relatively unimportant ways.

Traditionally, researchers have tested refactoring tools by applying them to manually or automatically generated test programs. Automated test generators have been an attractive target of testing researchers, and they've shown some success by finding tests that trigger real bugs in production IDEs such as Eclipse. However, these tools often produce tests that manifest corner cases in the programming language that developers don't care about. Fixing these bugs reduces the number of errors in such tools, but do they help make the tools more robust?

We tested the refactorings in several popular tools by applying them to large, open source codes, to replicate real refactoring scenarios.[9] We applied refactorings on many targets. For example, we applied Rename on the program's identifiers and Extract Method on arbitrary code sequences. We expected that the tools would be robust enough to handle most of the cases (these were mature refactoring tools, after all). The results surprised us.

We tested all 23 Java refactorings available in JDT; we found and reported 76 new bugs. Interestingly, the bugs didn't represent corner cases of a program; some were common features. On average, approximately 2 percent of the refactoring invocations failed, which suggests the tools are relatively reliable for Java programs but that opportunities to improve exist.

On the other hand, when we tested the five refactorings in the Eclipse C/C++ Development Tooling, they were much brittler. We reported 43 new bugs. (We performed the experiments in late 2012.[9] Some of the bugs have been fixed since then.) Even the oft-used Rename broke in approximately 6 percent of all invocations, while the less-common Toggle Function refactoring failed in approximately 30 percent of invocations. Worse, these bugs weren't Eclipse-specific. We reproduced the Java bugs in NetBeans and IntelliJ IDEA, and the C/C++ bugs in Visual Assist X, a Visual Studio extension providing C++ refactoring support.

Refaster,[10] developed at Google, provides an interesting counterpoint to the demand for refactoring-tool robustness. It's built on the assumption that compiler errors and the code's test suite are sufficient to indicate whether a transformation has broken a program. It performs simple syntactic replacement, given an example of a "before" program and the desired "after" program with the transformation applied to it. It doesn't try to check for behavior preservation. Instead, the compiler and test suite are supposed to find bugs in refactored code. This simple implementation is necessary because the tool is supposed to make many changes on millions of lines of code; scalability is desired more than robustness. (As time progresses, it will be interesting to see whether other tools adopt this approach, in which the onus of semantic correctness is on the programmer, not the tool.)

Refactoring tools aren't error-free. They work just well enough to be useful, and they break in relatively unimportant ways.

ABOUT THE AUTHORS

MUNAWAR HAFIZ is an assistant professor in Auburn University's Department of Computer Science and Software Engineering. His research focuses on applying program analysis and program transformation technologies and exploring empirical data to promote tools and methodologies that effectively improve the programming experience. Hafiz received a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at munawar.hafiz@gmail.com; www.munawarhafiz.com.

JEFFREY OVERBEY is an assistant professor in Auburn University's Department of Computer Science and Software Engineering. He has worked on refactoring tools for Fortran, Go, and C and is working on refactorings for GPU programming. Overbey received a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at joverbey@auburn.edu.

Interestingly, Refaster operates in batch mode, unlike a traditional interactive refactoring built into an IDE. This brings us to the final myth.

The Myth about Support

This myth is that integrating automated refactoring support into an IDE is the best way to attract users to automated refactoring.

The best-known refactoring tools (including IntelliJ IDEA, JDT, and ReSharper) are integrated into IDEs. IDEs are ubiquitous; they provide a project structure, a graphical interface, undo capabilities, and many other features that make them suitable for refactoring support.

When the Photran project started more than 10 years ago, we attempted to bring refactorings to Fortran programmers by integrating them into Eclipse.[11] Eclipse was popular and provided a solid foundation for building refactoring tools. Surely, Fortran programmers would like Eclipse as much as Java programmers.

There was something we didn't realize at the time.

Fortran is used mainly in high-performance computing. Programmers use SSH (Secure Shell) to connect to a remote machine, and they write code using Vim or Emacs, taking advantage of the compilers and libraries on the remote system. Scientific programmers don't care for IDEs and are wary of most software development tools; their objective is to produce science, not software.[12] So, marketing a refactoring tool to less-than-receptive Fortran programmers was going to be a challenge, but marketing an IDE to scientific programmers was a challenge in its own right.

The best advice comes from Don Roberts and his colleagues' classic paper, "A Refactoring Tool for Smalltalk."[13] "The refactoring tool must fit the way that [programmers] work. … The refactorings must be integrated into the standard development tools." In other words, refactoring must be part of the environment the programmer already uses—which might or might not be an IDE.

Although the most successful refactoring tools have been integrated into IDEs, fixating on IDEs might be dangerous. Can refactoring tools be brought to programmers who don't use IDEs? XRefactory is a refactoring browser for C/C++ programmers that integrates with Emacs, for example. In OpenRefactory/C, we integrated a refactoring engine with several common text editors: Vim, Notepad++, and Sublime Text.[14] The refactorings can also be applied using a command line interface. Perhaps the most convincing evidence comes from Google's Go programming language. Tools for Go are almost exclusively command line tools. The gofmt, gofix, and gorename tools all make source code changes, and Go programmers use them extensively. So, a future might exist for refactoring outside the IDE after all.

It's easy enough to stereotype the current refactoring landscape. Most advanced IDEs include automated refactorings that provide developers with interactive tools to improve a system's design without changing its behavior. That's true, but it doesn't fully depict reality.

Reality is a bit more complex (and mundane). Developers are increasingly adopting tools that predictably automate common source code changes. Some are design-level changes; some aren't. Some preserve behavior; some don't. Tools that are successful fit into the developer's workflow; some workflows involve

IDEs, while others don't. As refactoring tools and other source transformation tools continue to progress, we expect that the refactoring myths will begin dissipating as these realities become increasingly prominent.

Myths become a part of our perception; they're hard to dispel. In ancient Greece, myths' power and influence weakened only when logical discourse became popular. In our context, the logical discourse will come from more research challenging the refactoring myths. Only then can we advance beyond the misconceptions.

Acknowledgments

US National Science Foundation grant CCF-1217271 and a Google Faculty Research Award have supported this research.

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1997.
2. J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.
3. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
4. I. Balaban, F. Tip, and R. Fuhrer, "Refactoring Support for Class Library Migration," *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 05), 2005, pp. 265–279.
5. E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Software Eng.*, vol. 38, no. 1, 2012, pp. 5–18.
6. M. Vakilian et al., "Use, Disuse, and Misuse of Automated Refactorings," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 233–243.
7. A. Shaw, D. Doggett, and M. Hafiz, "Automatically Fixing C Buffer Overflows Using Program Transformations," *Proc. 44th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 14), 2014, pp. 124–135.
8. D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," *Proc. IEEE 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 397–407.
9. M. Gligoric et al., "Systematic Testing of Refactoring Engines on Real Software Projects," *Proc. 27th European Conf. Object-Oriented Programming* (ECOOP 13), 2013, pp. 629–653.
10. L. Wasserman, "Scalable, Example-Based Refactorings with Refaster," *Proc. 2013 ACM Workshop Refactoring Tools* (WRT 13), 2013, pp. 25–28.
11. J. Overbey et al., "Refactorings for Fortran and High-Performance Computing," *Proc. 2nd Int'l Workshop Software Eng. for High-Performance Computing System Applications* (SE-HPCS 05), 2005, pp. 37–39.
12. J. Carver et al., "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," *Proc. 29th Int'l Conf. Software Eng.* (ICSE 07), 2007, pp. 550–559.
13. D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, 1997, pp. 253–263.
14. M. Hafiz et al., "OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations," *Proc. 2013 ACM Workshop Refactoring Tools* (WRT 13), 2013, pp. 1–4.

FOCUS: REFACTORING

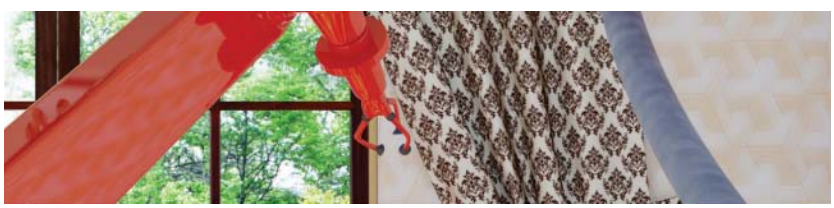
Challenges to and Solutions for Refactoring Adoption

An Industrial Perspective

Tushar Sharma and Girish Suryanarayana, Siemens Technology and Services Private Limited

Ganesh Samarthyam, independent consultant and corporate trainer

// Several practical challenges must be overcome to facilitate industry's adoption of refactoring. Results from a Siemens Corporate Development Center India survey highlight common challenges to refactoring adoption. The development center is devising and implementing ways to meet these challenges. //



INDUSTRIAL SOFTWARE systems typically have complex, evolving code bases that must be maintained for many years. It's important to ensure that such systems' design and code don't decay or accumulate technical debt.[1] Software suffering from technical debt requires significant effort to maintain and extend.

A key approach to managing technical debt is refactoring. William Opdyke defined refactoring as "behavior-preserving program transformation."[2] Martin Fowler's seminal work increased refactoring's popularity and extended its academic and industrial reach.[3] Modern software development methods such as Extreme Programming ("refactor mercilessly")[4] have adopted refactoring as an essential element.

However, our experience assessing industrial software design[5] and training software architects and developers at Siemens Corporate Development Center India (CT DC IN) has revealed numerous challenges to refactoring adoption in an industrial context. So, we surveyed CT DC IN software architects to understand these challenges. Although we knew many of the problems facing refactoring adoption, our survey gave us insight into how these challenges ranked within CT DC IN. Drawing on this insight, we outline solutions to the challenges and briefly describe key CT DC IN initiatives to encourage refactoring adoption. We hope our survey findings and refactoring-centric initiatives help move the software industry toward wider, more effective refactoring adoption.

Survey Details

CT DC IN is a core software development center for Siemens products. Its software systems pertain to different Siemens sectors (Industry, Healthcare, Infrastructure & Cities, and Energy), address diverse domains, are built on different platforms, and are in various development and maintenance stages.

CT DC IN, which has increasingly focused on improving its software's internal quality, wanted to understand the organization's status quo regarding technical debt, code and design smells, and refactoring. Furthermore, recent internal design assessments and training sessions revealed challenges to refactoring adoption. To better understand these deterrents—and thereby adopt appropriate measures to address them—we conducted our survey.

0740-7459/15/$31.00 © 2015 IEEE

RELATED WORK IN REFACTORING SURVEYS

Tom Mens and Tom Tourwe reviewed refactoring research in terms of the refactoring activities supported, the techniques and formalisms for supporting these activities, and refactoring's effect on the software process.[1] The literature also includes surveys and evaluations of refactoring tools (for example, the technical report by Jocelyn Simmonds and Tom Mens[2]), but they don't focus on challenges to adopting refactoring in an industrial context.

At Microsoft, Miryung Kim and her colleagues surveyed and interviewed 328 developers and analyzed version history data to identify refactoring benefits and challenges.[3] Respondents cited six key risk factors: regression bugs, code churns, merge conflicts, time taken from other tasks, difficulty performing code reviews after refactoring, and overengineering.

Emerson Murphy-Hill and Andrew Black surveyed 112 Agile Open Northwest conference attendees.[4] They found that professional programmers underused refactoring tools and that better, more usable, refactoring tools were needed. Our survey (see the main article) echoes these findings; we're trying to discover the specific problems architects and their teams face while using refactoring tools.

Finally, Aiko Yamashita and Leon Moonen surveyed 85 software professionals on code smells and related tooling.[5] They reported a finding similar to ours: refactoring tools should provide better support for refactoring suggestions.

References

1. T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, 2004, pp. 126–139.
2. J. Simmonds and T. Mens, *A Comparison of Software Refactoring Tools*, tech. report vub-prog-tr-02-15, Programming Technology Lab, Vrije Univ. Brussel, 2002.
3. M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," *IEEE Trans. Software Eng.*, vol. 40, no. 7, 2014, pp. 633–649.
4. E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.
5. A. Yamashita and L. Moonen, "Do Developers Care about Code Smells? An Exploratory Survey," *Proc. 20th Working Conf. Reverse Eng.* (WCRE 13), 2013, pp. 242–251.

(For a brief look at other refactoring surveys, see the related sidebar.)

This survey targeted software architects because they provide technical leadership of their projects and overall responsibility for refactoring. Our survey consisted of two question categories. The first addressed the architects' backgrounds and projects to ensure that they represented the larger organization.

The second category (questions 1 through 4 in Table 1) queried the architects about the problems they faced during refactoring and the shortcomings of existing refactoring methods and tools. We provided a list of predetermined options, and the survey respondents marked those that were relevant. We based these options on feedback from design and architecture training programs and from code and design assessments. Each question also had an "other" option, allowing respondents to describe issues or problems not on the list. Additionally, the survey had a comments field in which respondents could share their experiences and opinions about refactoring.

We distributed the survey via an organization-wide mailing list of software architects. Of the 73 architects on the list, 39 responded. The respondents' backgrounds were diverse. They represented all four Siemens sectors and had an average 11.6 years of software industry experience. Their projects represented various programming platforms and development stages (ranging from new development to long-term maintenance). Across the projects, the oldest software component's average age was 8.5 years.

Challenges to Refactoring Adoption

The challenges to refactoring adoption are well known. Our survey's value—from an organizational perspective—lies in its insights into how the architects ranked the problems. We used this ranking to prioritize and plan efforts to enhance the organization's refactoring adoption. Here, we describe the main challenges at CT DC IN.

Fear of Breaking Code

All respondents agreed that the fear of breaking working code is a major obstacle to refactoring tasks (question 1). This fear underlies the belief of many developers and managers

that "if it ain't broke, don't fix it" (where "broke" means software defects). When the code base is hard to comprehend (for example, complex legacy software for which the team can't access the original developers), the fear is even more pronounced.

Getting Management Buy-In

Question 1 responses also highlight the architects' difficulty in getting management buy-in for refactoring. More than three-fourths (76 percent) of the respondents reported not having time for refactoring tasks because management focuses on feature implementation. Furthermore, 38 percent felt unable to convince management of the need for refactoring, and 32 percent felt unable to justify to management the amount of effort required for refactoring.

In our experience, managers mainly have reservations about large-scale refactoring (refactoring that requires many person months). Legacy projects tend to be huge, stable, and lacking unit tests; large-scale refactoring in such projects is risky because it could break the working code. Development projects have considerable time-to-market and schedule pressures; so, instead of allocating time for large-scale refactoring, managers prefer to allot this time to fixing defects and adding features.

Some managers (especially those with nontechnical backgrounds) are unaware of the benefits of evolutionary design and refactoring. This unawareness is reflected in one respondent's comment that refactoring "is considered rework and has negative value. Project management expects things to be right the first time."

Lack of Awareness

Thirty-seven percent of the respondents mentioned that their team was unaware of refactoring's effect on product quality (question 1). Another 69 percent reported lack of exposure to relevant refactoring tools (question 2).

Inadequate Tool Support

Tools that help during refactoring might analyze the software and report problems as violations of predefined rules (critique tools), detect refactoring candidates, or perform refactorings.

Approximately 59 percent of the respondents were dissatisfied with the quality of available refactoring tools (question 3). Approximately 50 percent said that their critique tools generated too many false positives, or "false alarms" (question 4). As one respondent wrote, "We use two critique tools, but the number of results yielded is far too many; only some findings are useful."

A few respondents commented that existing refactoring tools can't identify relevant refactoring candidates and corresponding recommendations (question 4). Incidentally, Gustavo Pinto and Fernando Kamei reported that only a few refactoring recommendation tools identify candidate spots for refactoring using a specific refactoring technique, and these haven't matured despite refactoring recommendation being the most desired feature in refactoring tools.[6] Furthermore, 41 percent of respondents felt that refactoring tools lacked the key ability to predict refactoring's impact. Additionally, 28 percent wished refactoring tools could help them estimate the effort required for refactoring (question 4).

We've observed that tools for refactoring source code support primitive refactorings but not composite refactorings. For instance, Visual Studio supports primitive refactorings such as Rename but doesn't support useful composite refactorings such as Extract Class.[7] Furthermore, although Eclipse supports Extract Class, it doesn't execute the task flawlessly. It requires manually moving all relevant methods into a new class and doesn't automatically fix old method references.

Addressing the Challenges

Regarding software engineering practices, Leif Singer reported that simply mandating behaviors won't solve adoption problems.[8] (For a look at other research related to software engineering practice adoption, see the related sidebar.) Factors such as social influence, knowledge management, motivation, and persuasion must also be addressed. CT DC IN's approach to refactoring adoption incorporates all these elements. For example, they've instituted workshops, architects as refactoring champions, reward-and-recognition policies, and recognition and encouragement of teams that continuously refactor.

Practices

We identified five practices to address CT DC IN's refactoring adoption challenges.

Treat refactoring as integral to software development. Development teams and organizations often don't prioritize refactoring as much as other software development practices. However, integrating refactoring into software development can forestall the dedicated, multimonth efforts involved in large-scale refactoring (which would require management buy-in). Practices such as continuous refactoring ("floss refactoring"[9]) help pay off any technical debt incurred since the last refactoring. To promote development

TABLE 1

Refactoring adoption challenges: survey questions and responses*

Question	Response option	No. of respondents
1. Which of the following issues are obstacles in planning and undertaking refactoring tasks in your project? Select all that apply.	Fear of breaking working code	37
	Unable to perform refactoring tasks because of an emphasis on feature implementation	28
	Unable to predict the impact of the changes caused by refactoring	16
	Unable to convince higher management about the need for refactoring	14
	Lack of stakeholder awareness about refactoring's impact on product quality	14
	Unable to justify to higher management the amount of effort actually required for refactoring	12
	Unable to estimate the effort required for refactoring	11
	Unable to clearly show the improvement in software quality once refactoring is complete	8
	Unable to prioritize refactoring candidates (design problems)	7
	Unable to identify refactoring candidates	3
	Other	Lack of automated testing (2) Legacy code (1) Lack of support from project management (1) Calculating the improvement after refactoring is tedious (1)
2. Which of the following reasons are deterrents to adopting a refactoring tool? Select all that apply. The term "refactoring tool" includes refactoring-candidate identification tools and critique tools, as well as tools that perform refactoring.	Lack of exposure of the team to relevant refactoring tools	27
	Cost or budgetary constraints (for example, tool might be expensive)	21
	Difficult to find relevant refactoring tools	11
	We want to focus on assessing and improving only the changed or newly added parts in the source code, and refactoring tools can't run only on this differential code base	10
	Other	0
3. How satisfied are you with the refactoring tools you use in your project?	Not satisfied at all	0
	Somewhat satisfied	20
	Satisfied	11
	Very satisfied	3
	Extremely satisfied	0
4. What problems or issues have you faced while using refactoring tools? Select all that apply.	Tools generate too many "false alarms" (they generate wrong warnings)	19
	Tools can't predict the impact of the changes when refactoring would be performed	16
	Tools find too many problems, making it very difficult to process the results	16
	Lack of knowledge on tool usage	13
	Unable to estimate the effort required for refactoring	11
	Tools are difficult to customize for my project-specific needs	11
	The tool output is difficult to understand and process	10
	Tools find very few design problems, and most design problems go unreported	9
	Tools are difficult to integrate with the development life cycle or build environment	7
	Other	Refactoring tools are unable to identify relevant refactoring candidates and corresponding recommendations (3) Lack of support for composite refactorings (1)

** All 39 participants responded to questions 2 and 4. Question 1 had 37 respondents; question 3 had 34.*

RELATED WORK IN SOFTWARE ENGINEERING PRACTICE ADOPTION

Drawing on a systematic literature review of 26 studies, Mathieu Lavallée and Pierre Robillard illustrated software process improvement's effect on developers and identified seven factors that cause developers to resist adopting changes to existing practices.[1] Besides identifying challenges to the adoption of software-engineering practices, Leif Singer pinpointed specific adoption patterns.[2] Many of the challenges these authors identified are also reflected in our survey results. Furthermore, Siemens Corporate Development Center India's measures to address refactoring challenges can be mapped to Singer's adoption patterns.

In *Diffusion of Innovations*, Everett Rogers described a "preventive innovation" as an innovation that's adopted particularly slowly because the affected individuals have trouble perceiving its relative advantage.[3] We believe refactoring is a preventive innovation. Its effects aren't always directly visible to users, so considerable effort must be expended to portray its benefits to project management.

References

1. M. Lavallée and P.N. Robillard, "The Impacts of Software Process Improvement on Developers: A Systematic Review," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 113–122.
2. L. Singer, "Improving the Adoption of Software Engineering Practices through Persuasive Interventions," PhD thesis, Faculty of Electrical Eng. and Computer Science, Leibniz Univ. Hannover, 2013.
3. E.M. Rogers, *Diffusion of Innovations*, Simon & Schuster, 2010.

teams' continuous refactoring, CT DC IN has adopted a multipronged approach that includes training programs and workshops to spread awareness of continuous refactoring's benefits.

Additionally, CT DC IN has developed design, code, and test guidelines for high-quality software.[10] The guidelines dictate that developers employ static-analysis tools on the source code to generate metrics. These metrics, which indicate the software's quality, must be within predefined thresholds; otherwise, the code must be refactored. Currently, CT CD IN only enforces these guidelines before a release; however, to encourage continuous refactoring, the organization plans to enforce their use throughout development. Specifically, developers will run the static-analysis tools and then generate and submit reports while checking the code in the version-control system.

Augment tool use with expert input. Our experience suggests that manual intervention can somewhat address refactoring tools' shortcomings. For example, experienced team members can prioritize violations reported by critique tools on the basis of their impact on quality attributes, violation of design principles, or violation of constraints. During this process, the team members can identify and discard false positives.

In some cases, expert judgment can complement automated tools for effective refactoring. For instance, critique tools primarily identify refactoring candidates but don't specify corresponding refactorings. Experts can carefully analyze and annotate the violations with refactoring suggestions. The expert-based code and design assessments already rolled out at CT CD IN have been effective in this regard.[5,10]

Foster the right team culture. Team and organization culture influences software development activities and the quality of the artifacts produced. Andrew Hunt and David Thomas applied the "broken-window theory" to programming.[11] This theory proposes that, when a broken window in a building isn't repaired, the inhabitants will feel a sense of abandonment and will be more likely to litter or to damage the structure. Similarly, if most of a programming team doesn't refactor regularly, the remaining members tend to be demotivated and cease to refactor. However, if team members not only implement best practices but also perform them with rigor and professionalism, other members—whether old or new—will likely follow suit.

CT DC IN is exploring ways to trigger an attitude change toward refactoring, such as adopting a reward-and-recognition policy for teams that continuously refactor and maintain high design quality. The organization also plans to focus its internal architecture and design training programs so that architects will be well trained and will help usher in a culture of refactoring.

Refactor incrementally. Ideally, regular refactoring keeps technical debt manageable. However, when technical debt is already huge and unmanageable, we advise incremental refactoring rather than a huge lump-sum effort ("root canal refactoring"[12]). Such a strategy involves identifying, prioritizing (for instance, on the basis of files to be changed owing to a bug fix or feature addition), and executing refactoring candidates, along with planned fixes and enhancements. Managing and repaying technical debt in such situations depends on factors such as the remaining product life, debt severity and its impact on future development activities, and current business-related constraints.

To increase organizational awareness, CT DC IN recently conducted a workshop on technical-debt management. About 60 participants attended the event, which featured sessions on foundational concepts, Siemens case studies, and refactoring tools and techniques. Attendee feedback indicated that they gained a better understanding of technical debt and pragmatic refactoring practices and tools.

Use automated tests. Automated tests are considered a safety net for refactoring activities. In many situations, development teams must refactor a code base that has either no or poor-quality automated tests. This creates a chicken-and-egg problem: although writing automated tests without refactoring is difficult (owing to low testability), refactoring without automated tests is also difficult. Some of our survey respondents also mentioned the lack of automated tests as a refactoring deterrent (question 1).

Ideally, the development team strives for good-quality automated tests with high test coverage; otherwise, over the years, they accumulate legacy code base with test debt.[1] To deal with this, whenever developers fix or enhance legacy code, they should write a set of automated tests for that piece of code. This strategy allows a comprehensive set of automated tests to emerge over time for the software, thereby providing a safety net for future refactoring activities.[13] To increase awareness of these practices, CT DC IN has integrated specific sessions into advanced training on software testing for development teams.

> We hope our findings and refactoring initiatives help move software development toward more effective refactoring adoption.

Education

Our survey results show that many development teams aren't sufficiently aware of either the benefits of refactoring or relevant refactoring tools. So, education in these areas is key.

Dedicated training on refactoring. Hands-on training and workshops can help developers better understand refactoring concepts and techniques and expose them to relevant refactoring tools. CT DC IN is in the process of launching a three-day refactoring training. This training aims to cultivate software developers' spirit of craftsmanship[14] by covering design principles, code and design smells, refactoring for design smells, and design patterns' role in refactoring.

A refactoring-tool demonstration. Tool-focused sessions can expose development teams to new and better tools as well as to previously overlooked features of their existing tools. During the CT DC IN technical-debt workshop, we conducted an hour-long demonstration of refactoring tools to increase awareness of the kinds of tools available and their effective application.

Tools

On the basis of on our survey results (question 4) and our experience with real-world projects, we recommend the following four approaches to improve refactoring tools.

Identify genuine refactoring candidates. Critique tools' detection approach must be refined to significantly reduce the number of false positives. Furthermore, many critique tools don't categorize reported violations, necessitating a manual effort to sift through and prioritize these violations. So, tools must categorize the violations according to factors such as the severity or impact on quality attributes so that development teams can address high-priority violations first. Finally, most critique tools fail to identify several design issues that manual reviews can identify, highlighting the need for more comprehensive critique tools.

Provide refactoring recommendations. Refactoring recommendations not only identify refactoring candidates but also provide guidance on specific

ABOUT THE AUTHORS

TUSHAR SHARMA is a technical expert at the Corporate Research and Technology Center, Siemens Technology and Services Private Limited, India. His research interests include software design and architecture, code and design quality, and refactoring. Sharma received an MS in computer science from the Indian Institute of Technology-Madras. He's an IEEE senior member. Contact him at tusharsharma@ieee.org

GIRISH SURYANARAYANA is a senior research scientist at the Corporate Research and Technologies Center, Siemens Technology and Services Private Limited, India. His research interests include software architecture and design, design smells and refactoring, and software analytics. Suryanarayana received a PhD in information and computer science from the University of California, Irvine. He's a member of the *IEEE Software* advisory board. Contact him at girish.suryanarayana@siemens.com.

GANESH SAMARTHYAM is an independent consultant and a corporate trainer. His research interests include software architecture, software design and refactoring, and programming languages. He received an MCA (master of computer applications) degree from the PSG College of Technology. Contact him at ganesh.samarthyam@gmail.com.

refactoring techniques. As we mentioned before, few refactoring tools support recommendations, and those that do support only a few refactorings. CT DC IN is adopting an internally developed refactoring tool that provides recommendations.[15]

Support impact analysis and effort estimation. Our survey results highlight two features missing in existing refactoring tools (similar to findings by Miryung Kim and her colleagues[16]). The first is comprehensive change impact analysis, which predicts source code entities that refactoring could affect. This feature can boost developers' confidence about the planned refactoring changes. The second feature is effort estimation, which predicts the effort required to refactor and tests the change's correctness.

Support nontrivial composite refactorings. IDEs and their extensions need to reliably support and realize nontrivial composite refactorings to encourage refactoring adoption.

Limitations

This work has a few limitations. First, the small sample of architects might limit the survey results' validity. This is due partly to organizational policies requiring surveys to be voluntary. Furthermore, a refactoring survey should include the software developers executing the actual refactoring. So, we're planning a more comprehensive survey to include all relevant stakeholders. Nevertheless, we believe that because the architects are responsible for a project's technical leadership, their responses largely reflect the development team's. Hence, our initial survey's results still provide useful insights.

A second limitation is that it's difficult to evaluate the effectiveness and usefulness of training programs, workshops, and seminars, particularly in an organizational context. At CT DC IN, we've addressed this limitation by eliciting workshop and training participants' feedback on these educational activities' perceived usefulness. (We don't include those survey results here because of space constraints.)

In this article, we outlined CT DC IN initiatives to address its refactoring adoption challenges. We plan to conduct a detailed study on these initiatives' effectiveness in promoting refactoring at CT DC IN.

Despite refactoring's popularity, the large body of refactoring research, and the emergence of various refactoring tools, many practical challenges remain to the wider adoption of refactoring in industrial contexts. Organizations must approach such challenges holistically, integrating refactoring best practices and educational approaches and techniques into the organizational framework. Furthermore, academia, industry, and tool vendors must collaborate to create a new generation of more useable, effective refactoring tools.

References

1. G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design*

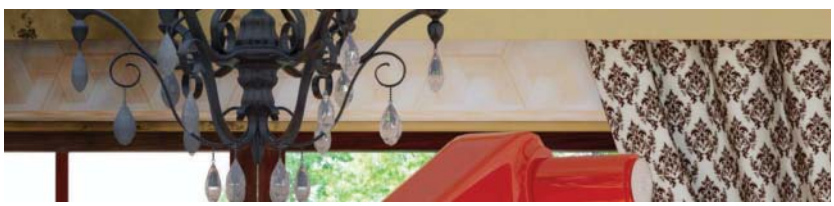
Smells: Managing Technical Debt, Morgan Kaufmann, 2014.
2. W.F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD thesis, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1992.
3. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2004.
5. G. Samarthyam et al., "MIDAS: A Design Quality Assessment Method for Industrial Software," *Proc. 2013 Int'l Conf. Software Eng.* (ICSE 13), 2013, pp. 911–920.
6. G.H. Pinto and F. Kamei, "What Programmers Say about Refactoring Tools? An Empirical Investigation of Stack Overflow," *Proc. 2013 ACM Workshop Refactoring Tools* (WRT 13), 2013, pp. 33–36.
7. "Welcome to Visual Studio 2015," Microsoft, Apr. 2015; https://msdn.microsoft.com/en-us/library/dd831853.aspx.
8. L. Singer, "Improving the Adoption of Software Engineering Practices through Persuasive Interventions," PhD thesis, Faculty of Electrical Eng. and Computer Science, Leibniz Univ. Hannover, 2013.
9. E. Murphy-Hill and A.P. Black, "Why Don't People Use Refactoring Tools?," *Proc. 1st ECOOP Workshop Refactoring Tools*, 2007, pp. 60–61.
10. S. Gupta et al., "SCQAM: A Scalable Structured Code Quality Assessment Method for Industrial Software," *Proc. 22nd Int'l Conf. Program Comprehension* (ICPC 14), 2014, pp. 244–252.
11. A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.
12. E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.
13. M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004.
14. S. Mancuso, *The Software Craftsman: Professionalism, Pragmatism, Pride*, Prentice Hall, 2014.
15. T. Sharma, "Identifying Extract-Method Refactoring Candidates Automatically," *Proc. 5th Workshop Refactoring Tools* (WRT 12), 2012, pp. 50–53.
16. M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," *IEEE Trans. Software Eng.*, vol. 40, no. 7, 2014, pp. 633–649.

FOCUS: REFACTORING

Refactoring for Asynchronous Execution on Mobile Devices

Danny Dig, Oregon State University

// Refactoring that changes synchronous code to asynchronous code can improve mobile apps' responsiveness. However, using asynchrony presents obstacles. Researchers have developed educational resources and tools that can help. //



ASYNCHRONOUS PROGRAMMING is in demand. Asynchrony is essential for long-running CPU activities (for example, image encoding or decoding) or I/O activities that are potentially blocking (for example, accessing the Web or file system). Asynchrony helps an application stay responsive because it can continue with other work.

Asynchrony is especially valuable for applications that access the UI thread. Today's UI frameworks are usually designed around a single UI event thread: every operation that modifies UI state executes atomically as an event on that thread.[1] The UI freezes when it can't respond to input or can't be redrawn; because it seems nonresponsive, users might become frustrated. It's recommended that long-running CPU-bound or blocking-I/O operations execute asynchronously so that the application continues to respond to UI events.

Other times, asynchrony is the natural programming model. For example, event-driven programming models arise naturally to match the asynchronicity of input streams coming from diverse sources such as touchscreens, accelerometers, microphones, and other sensors on smartphones. Similarly, modern Web applications extensively use asynchrony, through AJAX (asynchronous JavaScript and XML) requests, and use asynchronous code loading to reduce the perceived page load time.

This article focuses on mobile apps because they provide many exemplars of asynchronous programming, given that responsiveness is critical. They can easily be unresponsive because mobile devices have limited resources and high latency (excessive network accesses).[2] With the immediacy of touch-based UIs, even small hiccups in responsiveness are more obvious and jarring than with a mouse or keyboard. Some sluggishness might motivate users to uninstall an app and submit negative comments in the app store. Here, I show how refactoring synchronous code into asynchronous code can improve mobile apps' responsiveness and avoid these problems.

The Problems with Asynchrony

When programmers work with asynchrony, they encounter two key problems.

The Lack of Methods and Tools

The first problem is the lack of methods and tools for converting synchronous code to asynchronous code. In formative studies of hundreds of open source apps, my colleagues and I found that half of the asynchrony usages weren't introduced from scratch but had been converted from preexisting synchronous code. This step is a source-to-source transformation that must preserve the application's current functionality (while

0740-7459/15/$31.00 © 2015 IEEE

improving some nonfunctional property such as responsiveness). So, it's a refactoring.[3]

This refactoring requires complex code transformations that invert the control flow and introduce callbacks to notify the caller when an asynchronous operation finishes. Also, it requires reasoning about asynchronous operations' noninterference with the main thread of execution; otherwise, the asynchrony can lead to nondeterministic data races.

Currently, developers solve such problems and perform such nontrivial transformations manually. However, interactive refactoring tools can help.

The Lack of Knowledge

The second problem is the lack of clear knowledge of how to convert synchronous code into asynchronous code. There's extensive programmer documentation (for example, Android's *Best Practices for Performance*[1]) and tools that detect blocking-I/O operations (for example, StrictMode for Android[4]). But they focus on designing asynchronous programs from scratch. So, many programs suffer from poor responsiveness.

A recent paper found that 75 percent of performance bugs in Android apps arise because of missed opportunities to introduce asynchrony in UI code.[2] Our research discovered hundreds of places in UI code that should have used asynchrony.[5,6] This shows the need for educational resources on how to retrofit asynchrony.

Dealing with the Problems

My colleagues and I have addressed some of these challenges by releasing educational resources on our online portal, http://learnasync.net. We've also developed tools that automatically refactor synchronous code into asynchronous code. We discuss our portal and tools in more detail later. You can learn more about the tools and download them at http://refactoring.info/tools.

Refactorings for Asynchrony

Examples from GUI programming on mobile devices illustrate refactoring flavors and the refactorings my colleagues and I have developed. Although the code examples use C# and .NET APIs, similar constructs exist or are planned for Java and Android.

Most GUI frameworks use an event-driven model. Events in mobile apps include life-cycle events (for example, GUI screen creation), user actions (for example, button click and menu selection), and sensor inputs (for example, GPS and screen orientation change), and so on. Developers define event handlers to respond to these events.

To handle app events, the operating system creates a main thread—the UI event thread. This thread dispatches UI events to appropriate widgets or life-cycle events to screen pages. It puts events into an event queue, dequeues events, and executes the corresponding event handlers.

When a GUI event handler executes a synchronous long-running CPU-bound or blocking-I/O operation, the UI freezes because the UI event thread can't respond to events. Figure 1a shows a handler that responds to a button click by downloading and displaying the IEEE Computer Society homepage. The main thread invokes a blocking, potentially long-running operation, `getResponse`, which downloads the webpage, and the UI becomes unresponsive during the download. Figure 2a shows this code's execution flow, highlighting when the UI freezes.

To avoid unresponsiveness, programmers exploit asynchrony by encapsulating and running long-running CPU or blocking-I/O computations in the background.

Next, I present the two prevalent asynchronous-programming styles using examples from .NET:

- the framework-based, callback-based style, which uses the Asynchronous Programming Model (APM) from .NET 1.0, and
- the new pause-and-play style based on the asynchronous language constructs `async` and `await` (called `async/await` in the rest of this article) from the most recent version of .NET

Our refactoring tools target both styles.

Refactoring to Callback-Based Asynchrony

Figure 1b shows the refactored, asynchronous version of the code in Figure 1a. A `Begin` method (`BeginGetResponse`) invocation starts APM asynchronous operations; an `End` method (`EndGetResponse`) invocation obtains the results.

Figure 2b shows the execution flow for Figure 1b. `BeginGetResponse` initiates an asynchronous HTTP `GET` request. The .NET framework starts the I/O operation in the background (in this case, sending the request to the remote webserver). Control returns to the calling method, in this case the UI event handler, which can then continue to do something else; thus, it is responsive. When the server responds, the .NET

```
1  void Button_Click(...) {
2     var request = WebRequest.Create("computer.org");
3     var response = request.GetResponse();
4     var stream = response.GetResponseStream();
5     textBox.Text = stream.ReadAsString();
6  }
7
8
9
10
11
12
13
14 .
```

(a)

```
1  void Button_Click(...) {
2     var request = WebRequest.Create("computer.org");
3     request.BeginGetResponse(Callback, request);
4  }
5
6  void Callback(IAsyncResult aResult) {
7     var request = (WebRequest)aResult.AsyncState;
8     var response = request.EndGetResponse(aResult);
9     var stream = response.getResponseStream();
10    var content = stream.ReadAsString();
11    Dispatcher.BeginInvoke(() => {
12       textBox.Text = content;
13    });
14 }
```

(b)

```
1  async void Button_Click(...){
2     var request = WebRequest.Create("computer.org");
3     var response = await request.GetResponseAsync();
4     var stream = response.GetResponseStream();
5     textBox.Text = stream.ReadAsString();
6  }
7
8
9
10
11
12
13
14 .
```

(c)

FIGURE 1. Three versions of the same code for reading text from the Web. (a) The original synchronous code. (b) The asynchronous refactoring using callback-based constructs. (c) The **async/await** version. In the asynchronous versions, the UI can process other events during the I/O operation.

framework "calls back" to the application to notify it that the response is ready. The callback code then uses **EndGetResponse** to retrieve the operation's results.

An important difference exists between the synchronous example in Figures 1a and 2a and the asynchronous, callback-based example in Figures 1b and 2b. In the synchronous example, the **Button_Click** method contains the UI update (setting the download result as the text box's contents). In the asynchronous example, the final callback contains an invocation of **Dispatcher.BeginInvoke(...)** to change the context from the thread pool to the UI event thread and to update the display. (Updating GUI elements from outside the UI thread causes data races.)

Callback-based asynchronous programming has many variations. Several languages (such as Java, C#, and Scala) support lightweight *tasks*. A task can represent an asynchronous operation and its future result. Developers still must encapsulate any long-running operations or blocking I/O as tasks. For example, Android developers must encapsulate asynchronous operations in **AsyncTask** or **IntentService**. (We discuss these constructs in more detail later.) .NET developers can use the Task Asynchronous Programming (TAP) model.

Refactoring to async/await

So far, the dominant models for asynchronous APIs (for example, Android and C# versions before 5.0) rely on programmers reasoning about callbacks. However, callbacks invert the control flow, are awkward, and obfuscate the original synchronous code's intent.[7,8]

Let's revisit the example in Figure 1b. Using the APM style has two main drawbacks. First, the code

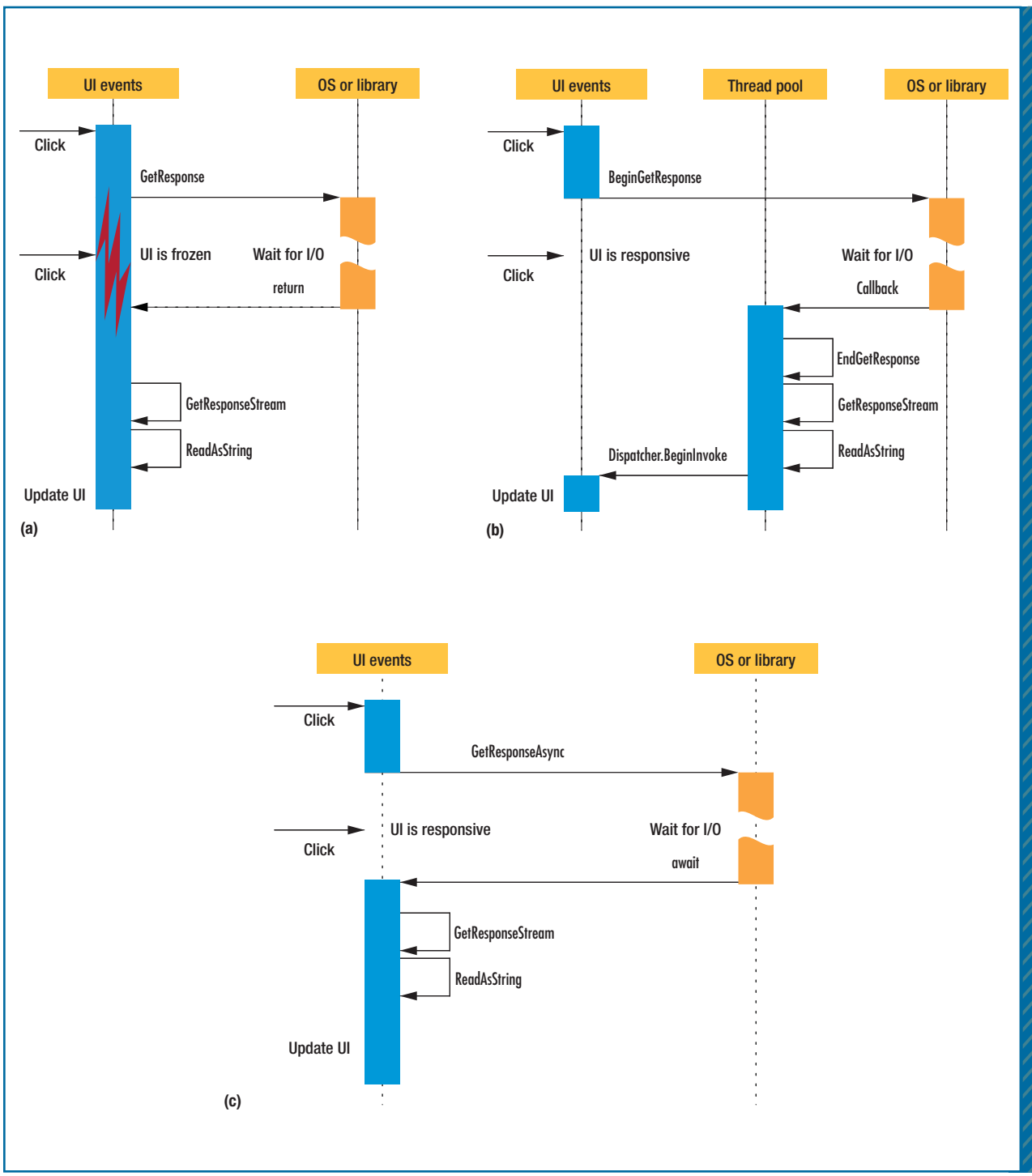


FIGURE 2. Run-time execution flow of the code in Figure 1. The time flows from top to bottom. (a) The execution flow of the code in Figure 1a. The UI freezes while waiting for the I/O operation. (b) The execution flow for the asynchronous code in Figure 1b. (c) The execution flow for the **async/await**-based code in Figure 1c. In Figures 2b and 2c, the UI can process other events during the I/O operation.

that must execute after the asynchronous operation finishes must be passed explicitly to the **Begin** method invocation. Even more scaffolding is required. The **End** method must be called, which usually requires the explicit passing and casting of an **AsyncState object** instance (see Figure 1b, lines 7 and 8).

Second, even though the **Begin** method might be called from the UI event thread, the callback code executes on a thread pool thread. To update the UI after the asynchronous operation from that thread completes, an event must be sent to the UI event thread explicitly (see Figure 1b, lines 11 to 13).

Recently, major programming languages (C# and Visual Basic,[9] F#,[7] and Scala[10]) introduced asynchronous constructs that resemble the straightforward coding style of traditional synchronous code. In the rest of this section, I use the variant introduced by C# 5.0.

This variant is based on the **async** and **await** keywords. When a method has the **async** keyword modifier in its signature, the **await** keyword can be used to define pausing points. When a task is awaited in an **await** expression, the current method pauses and control is returned to the caller. When the **await**ed background operation finishes, the method resumes from right after the **await** expression.

Figure 1c shows the **async/await**-based equivalent of Figure 1b. The code following the **await** expression can be considered a continuation of the method, exactly like the callback that must be supplied explicitly when using APM. Figure 2c shows the execution flow for Figure 1c.

An important difference exists between APM callback continuations and **async/await** continuations: APM always executes the callback on a thread pool thread. In **async/await** continuations, the **await** keyword, by default, captures information about the thread in which it executes. This captured context is used to schedule execution of the rest of the method in the same context as when the asynchronous operation was called.

In my example, because the **await** keyword is encountered in the UI event thread, once the background operation finishes, the continuation of the rest of the method is scheduled back onto the UI event thread. This behavior lets developers write asynchronous code that resembles the original synchronous code (compare Figures 1a and 1c).

With the advent of **async/await** constructs, the asynchronous code looks deceptively like the synchronous code. Although I applaud such engineering feats from the language designers and compiler writers, app developers must be aware of the execution models' fundamental differences (see Figure 2).

> We found misuses as real antipatterns, which hurt performance and caused serious problems such as deadlocks.

Formative Studies

It's easy for academic research to become disconnected from the software practice and for researchers to build tools that don't solve immediate real-world problems. To avoid this, our refactoring tools are grounded on empirical studies of how developers use, misuse, or underuse asynchrony.

Specifically, to obtain a deep understanding of the problems with asynchronous programming in Android and Windows Phone apps, my colleagues and I conducted the following two studies.

Asynchrony in Android

We conducted this study to understand how Android developers use asynchrony to improve responsiveness.[5] We analyzed the 104 most popular Android apps in GitHub, comprising 1.34M SLOC, produced by 1,139 developers. We found that 48 percent of those projects used asynchrony in hundreds of places, and half of them retrofitted asynchrony through manual refactoring.

We also found that 251 places in 51 projects executed long-running operations in the UI event thread and should have used asynchrony. We submitted refactoring patches for six apps. Developers of four apps replied and accepted 10 of our refactorings.

Asynchrony on Windows Phones

This study helped us understand how programmers use asynchrony through **async/await**.[9] (The next major version of Java plans to support similar constructs.) We analyzed 1,378 open source Windows Phone apps, comprising 12M SLOC, produced by 3,376 developers.[6]

In 76 percent of cases, developers used callback-based asynchronous idioms. However, Microsoft officially no longer recommends these

idioms.[11] Because refactoring these idioms to **async/await** idioms is nontrivial, the need exists for refactoring tools.

Refactoring Obstacles

On the basis of the formative studies, here I present the top 10 questions app developers must answer competently before refactoring synchronous code into asynchronous code. Answering these questions wrongly or not at all has severe consequences for

- correctness,[5] resulting in hard-to-debug data races;
- performance,[6] resulting in significant slowdowns or even deadlocks;
- maintainability,[6,12] resulting in obsolete code; and
- usability, resulting in confusing interfaces.

These questions fall into the following four categories related to the obstacles to asynchronous programming.

Obstacle 1: Concurrency

Four questions are concurrency related:

- What other code may run in parallel with the asynchronous code? Besides the main thread, are there other event handlers, or background threads spawned by the app, that could run in parallel?
- Do dependencies exist between the code to be executed asynchronously and the other code that may run in parallel?
- Is the code to be refactored called from within the UI event thread or another thread?
- After getting back the result from the asynchronous code, does the continuing code update the UI?

Our study of Android apps found that for 13 apps containing manual refactorings, the asynchronous code accessed objects in a way that wasn't thread-safe. We discovered 77 data races on GUI widgets. Developers had already fixed 53 of those races in later versions, and we discovered and reported 24 new races along with the patch to fix them. The developers acknowledged and accepted our patch.

Inspired by these developer needs, we released *Asynchronizer*, a tool that helps Android programmers safely refactor synchronous code into asynchronous code. As is typical in the refactoring community, to ensure our automated refactorings' safety, each refactoring is guarded by preconditions—criteria the input code must satisfy before it can be safely refactored.

Obstacle 2: Performance

Three questions are performance related:

- After I encapsulate work within a task, does the code launch it asynchronously?
- Is any other long-running or blocking-I/O code still in the UI thread?
- Does my code follow the vendor-specific performance recommendations for the target platform?

We found misuses as real antipatterns, which hurt performance and caused serious problems such as deadlocks. Our Android app study found that in 4 percent of the cases, the code launched an asynchronous task and immediately blocked to wait for the task's result. So, the code appeared to have asynchronous syntax but ran synchronously instead of asynchronously.

> Developers almost always unnecessarily captured UI context, hurting performance.

Our previous studies on concurrent libraries in C# discovered similar problems.[6,13] We found that 14 percent of methods that used (the expensive) **async/await** keywords did this unnecessarily. The awaited statement was the last one in an **async** method, so the method would pause unnecessarily because it would be awaited anyway at its call site.

In the following example, the last **await** is unnecessary because the task returned by **SendPutRequestAsync** and then **ChangeTemperatureAsync** will be awaited anyway at the call site of **ChangeTemperatureAsync** (not shown here):

```
public async Task<?> ChangeTempera-
tureAsync(...) {
...
return await SendPutRequestAsync(url,
requestString);
}
```

In addition, our Windows Phone study discovered that 19 percent of methods didn't follow the important practice that an **async** method should be awaitable unless it's the

top-level event handler.[14] When an **async** method returned **void** instead of a **Task**, the code fired an **async** method that couldn't be awaited, thus wasting resources. We call this idiom "fire and forget."

Moreover, we found that one of five apps missed opportunities in **async** methods to increase asynchrony.

We also found that developers almost always unnecessarily captured UI context, hurting performance. Recall the example from Figure 1c where the remainder of the **async** method executes on the UI thread. As asynchronous GUI apps grow larger, many small parts of **async** methods might use the UI event thread as their context. This can cause sluggishness as responsiveness suffers "death by a thousand paper cuts." The asynchronous code should run in parallel with the UI event thread instead of constantly badgering it with bits of work to do.

So much misuse of **async/await** suggests the need for transformation tools that find and fix performance antipatterns. So, we developed *AsyncFixer*, a tool that detects several **async/await** antipatterns and recommends fixes for them.

Obstacle 3: Continuous, Sometimes Aggressive, API Evolution

Like any useful API, the asynchronous APIs constantly evolve to support better constructs, hardware improvements that enable compiler optimizations, and so on. Just because developers have used asynchrony on their code doesn't mean they won't ever change that asynchronous code. App developers must answer questions related to programming-model evolution—for example,

> *Am I willing to learn a new programming model and change my already asynchronous code to support it?*

Consider the evolution of asynchronous constructs in the Android platform. Android provides three major asynchronous constructs: **AsyncTask**, **IntentService**, and **AsyncTaskLoader**. **AsyncTask** is for encapsulating short-running tasks; the other two are good for long-running tasks. However, as our most recent study on a corpus of 611 Android apps showed, **AsyncTask** is the most widely used construct, dominating by a factor of 3× over the other two choices combined.[12]

Using **AsyncTask** excessively for long-running tasks causes memory leaks, lost results, and wasted energy.[12] So, developers might consider refactoring **AsyncTask** into **IntentService**. However, this refactoring is nontrivial owing to drastic changes in communication with the GUI. This is a challenge because developers must transform shared-memory-based communication (through access to the shared variables) into distributed communication (through marshaling objects on special channels). Fortunately, semiautomated refactoring tools can handle this challenge.

Inspired by these obstacles, we released two tools that modernize existing asynchronous code. *AsyncDroid* lets Android programmers convert **AsyncTask** to **IntentService**. *Asyncifier* lets .NET programmers upgrade APM callback-style code to modern **async/await**.

Obstacle 4: Changing the UI paradigm

As a result of retrofitting asynchrony, app developers might have to change the UI paradigm too. Consider again the example from Figure 1. With the UI for the synchronous model of Figure 1a, users can only press the button once and then must wait for the download to finish before executing other UI actions. With the asynchronous models of Figures 1b and 1c, users can click the download button multiple times consecutively. So, multiple requests for downloading the same page might be in progress simultaneously. This can frustrate users and lead to inefficient use of resources.

App developers must answer questions such as these:

- If an **async** operation is already in progress, should the triggering UI widget be disabled?
- How can I group my UI widgets to balance responsiveness (making the user feel in control) and efficient use of resources (to avoid wasted subsequent requests that override each other's results)?

Answering such questions might require developers to rethink the UI workflow model and layout. For example, developers might disable a UI widget while an asynchronous operation is in progress or have the

app display a progress report during long-running operations.

Our Research's Practical Impact

Our research has had a practical impact in the following areas.

Our Online Portal

In our studies, we've seen extensive underuse and misuse of asynchronous constructs. Why is the misuse so extensive? Are developers unaware of the risks or performance characteristics of `async/await`?

To significantly improve education, our online portal provides resources for .NET programmers who want to learn about asynchronous constructs. Developers learn new programming constructs through both positive and negative examples. So, our portal provides thousands of real-world examples of all asynchronous idioms. Because developers might need to inspect the whole source file or project to understand an example, our portal links to highlighted source files on GitHub so that developers see the asynchronous code in its context.

Since we launched the portal two years ago, it has had more than 36,000 visitors.

Our Refactoring Tools

Here, we look at three of our tools: Asynchronizer, for Android, and Asyncifier and AsyncFixer, for .NET.

Asynchronizer. This tool lets app developers extract long-running operations into `AsyncTask`. We used it to refactor 135 places in 19 open source Android projects. We evaluated its usefulness from five angles. First, because 95 percent of the cases met refactoring preconditions, refactoring was highly applicable. Second, in 99 percent of cases, Asynchronizer applied changes similar to those that open source developers applied manually; thus, our transformation was accurate. Third, Asynchronizer changed 2,394 LOC in 62 files in just a few seconds per refactoring. Fourth, using Asynchronizer, we discovered and reported 169 data races in 10 apps. Developers of five apps replied and confirmed 62 races. Fifth, we submitted patches for 58 refactorings in six apps. Developers of four apps replied and accepted 10 refactorings.

Asyncifier and AsyncFixer. Our previous research showed that Asyncifier and AsyncFixer are highly applicable and efficient.[6] Developers find our transformations useful.

Using Asyncifier, we applied and reported refactorings in 10 apps. Nine of the developers replied and accepted our 28 refactorings. PhoneGuitarTab's developer said that he had "been thinking about replacing all asynchronous calls [with] new `async/await`-style code."

Using AsyncFixer, we found and reported misuses in 19 apps. All the developers replied and accepted our 286 patches. After applying our patch, Softbuilddata's developer experienced performance improvements: "Response time has been improved to 28 milliseconds from 49 milliseconds."

A subset of AsyncFixer will ship with the official release of Visual Studio at the end of 2015.

Individual and Ecosystem Outreach

Researchers can move their research into practice by connecting with individual developers and companies or with whole communities of developers. We call the former the "downstream" approach because the researcher integrates the research at the end of an existing pipeline of tools that developers already use. We call the latter the "upstream" approach because the researcher packages the research into an ecosystem of tools that gets supplied to a large community of developers. We've been pursuing both approaches.

The downstream approach. This approach provides unique points of interaction with developers, early feedback from users, deep insights into problems that industry faces, and the ability to replicate open source experiments on industrial code bases.

ABOUT THE AUTHOR

DANNY DIG is an assistant professor in Oregon State University's Electrical Engineering and Computer Science department and an adjunct assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests are automated refactoring, interactive program analysis and transformation, and design and architectural patterns. Dig received a PhD in computer science from the University of Illinois at Urbana-Champaign. His dissertation on API refactoring won the University of Illinois David Kuck Outstanding PhD Thesis Award and First Prize at the ACM Student Research Competition Grand Finals. Contact him at digd@eecs.oregonstate.edu.

RELATED WORK IN PROGRAM PERFORMANCE AND REFACTORING

Empirical studies of performance bug patterns in Android apps revealed that lack of responsiveness was the main culprit.[1,2] Testing researchers have used machine learning,[3] concolic testing,[4] and random testing[5,6] to generate test inputs for mobile apps. My research is complementary; I explore how programmers can use refactoring to eliminate performance issues detected by testing.

Although refactoring has usually been associated with improving code design, the refactoring community has been taking a similar approach to improve other nonfunctional requirements—for example, performance. In 2009 my colleagues and I published the research paper that opened the field of interactive refactorings for parallelism.[7] We've continued to publish extensively on this topic, including a summary paper that describes related work in this field.[8]

Whereas our formative studies identified misuse and underuse of refactoring in the context of responsiveness in mobile apps,[9–11] other researchers have described such problems in general software. Miryung Kim and her colleagues presented a field study at Microsoft on refactoring challenges and benefits.[12] Emerson Murphy-Hill and his colleagues reported on the state of the practice in refactoring usage.[13] Other researchers have studied the general use, disuse, and misuse of refactoring.[14,15]

References

1. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1013–1024.
2. S. Yang, D. Yan, and A. Rountev, "Testing for Poor Responsiveness in Android Applications," *Proc. 1st Int'l Workshop Eng. of Mobile-Enabled Systems* (MOBS 13), 2013, pp. 1–6.
3. W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," *Proc. 28th ACM SIGPLAN Int'l Conf. Object Oriented Programming Systems Languages & Applications* (OOPSLA 13), 2013, pp. 623–640.
4. S. Anand et al., "Automated Concolic Testing of Smartphone Apps," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 59.
5. C.S. Jensen, M.R. Prasad, and A. Møller, "Automated Testing with Targeted Event Sequence Generation," *Proc. 13th Int'l Symp. Software Testing and Analysis* (ISSTA 13), 2013, pp. 67–77.
6. C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," *Proc. 6th Int'l Workshop Automation of Software Test* (AST 11), 2011, pp. 77–83.
7. D. Dig, J. Marrero, and M.D. Ernst, "Refactoring Sequential Java Code for Concurrency via Concurrent Libraries," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 397–407.
8. D. Dig, "A Refactoring Approach to Parallelism," *IEEE Software*, vol. 28, no. 1, 2011, pp. 17–22.
9. Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications through Refactoring," *Proc. ACM SIGSOFT 22nd Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 341–352.
10. S. Okur et al., "A Study and Toolkit for Asynchronous Programming in C#," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1117–1127.
11. Y. Lin, S. Okur, and D. Dig, "Study and Refactoring of Android Asynchronous Programming," tech. report, School of Electrical Eng. and Computer Science, Oregon State Univ., 2015; http://hdl.handle.net/1957/56106.
12. M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 50.
13. E.R. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 287–297.
14. M. Vakilian et al., "Use, Disuse, and Misuse of Automated Refactorings," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 233–243.
15. S. Negara et al., "A Comparative Study of Manual and Automated Refactorings," *Proc. 27th European Conf. Object-Oriented Programming* (ECOOP 13), 2013, pp. 552–576.

As part of this approach, we're submitting refactoring patches to hundreds of open source projects. Moreover, we recently replicated our open source study of asynchronous constructs[6] on the proprietary code base of an industrial partner from the Pacific Northwest. The partner obtained a clear sense of how its code base compared to the open source projects in terms of the use, misuse, and underuse of `async` constructs.

This was clearly a win–win situation. We expanded our research on case studies that otherwise would have been unavailable to us; the company gained both a deeper understanding of their asynchronous-code practices and a list of actionable items.

The upstream approach. This approach can have a massive impact. As part of this approach, we're working with IDE developers (Visual Studio, Eclipse, and NetBeans) by contributing our research as plug-ins that ship with the official IDE version. We'll explore this even more.

We're also working with Google to deploy our refactoring and transformation tools as analyzers for the Shipshape static-analysis platform (https://github.com/google/shipshape). The vision is for Shipshape to become widely used. Developers who want to check code quality—for example, before submitting an app to Google Play—would run Shipshape on their code base. Shipshape lets custom analyzers, such as our `async` analysis and transformations, plug in through a common interface. Shipshape generates analysis results along with transformation patches that developers can accept on their code. We expect that by contributing new `async`

analyzers to Shipshape, we'll enable millions of app developers to execute our analyses and transformations on their code.

With a rich array of sensors, camera, and GPS, all integrated in a convenient form factor, mobile devices can offer new, exciting applications and services that previously weren't possible on consumer devices. But these can be harnessed only if mobile applications leverage asynchrony to ensure responsive behavior.

We hope our educational resources show developers asynchronous constructs' potential and pitfalls. We're constantly looking for industrial partners we can help to discover their asynchronous practices and refactor their code to improve responsiveness.

For a look at other research on program performance and refactoring, see the sidebar.

Acknowledgments

I thank Yu Lin, Semih Okur, Michael Hilton, Sruti Srinivasa, Lily Mast, and the anonymous reviewers for insightful comments on earlier drafts of this article. US National Science Foundation grants CCF-1439957 and CCF-1442157, a Microsoft Software Engineering Innovation Foundation award, and an Intel gift grant partly funded this research.

References

1. *Best Practices for Performance*, Android Open Source Project; http://developer.android.com/training/best-performance.html.
2. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1013–1024.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. "StrictMode," Android Open Source Project; http://developer.android.com/reference/android/os/StrictMode.html.
5. Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications through Refactoring," *Proc. 2014 ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 341–352.
6. S. Okur et al., "A Study and Toolkit for Asynchronous Programming in C#," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 1117–1127.
7. D. Syme, T. Petricek, and D. Lomov, "The F# Asynchronous Programming Model," *Proc. 2011 Int'l Conf. Practical Aspects of Declarative Languages* (PADL 11), 2011, pp. 175–189.
8. G. Salvaneschi et al., "An Empirical Study on Program Comprehension with Reactive Programming," *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 14), 2014, pp. 564–575.
9. G. Bierman et al., "Pause 'n' Play: Formalizing Asynchronous C#," *ECOOP 2012—Object-Oriented Programming*, LNCS 7313, 2012, pp. 233–257.
10. P. Haller and J. Zaugg, "SIP-22—Async," EPFL, 2013; http://docs.scala-lang.org/sips/pending/async.html.
11. "Asynchronous Programming Patterns," Microsoft; http://msdn.microsoft.com/en-us/library/jj152938.aspx.
12. Y. Lin, S. Okur, and D. Dig, "Study and Refactoring of Android Asynchronous Programming," to be published in *Proc. 2015 Int'l Conf. Automated Software Eng.* (ASE 15), 2015.
13. S. Okur and D. Dig, "How Do Developers Use Parallel Libraries?," *Proc. 20th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, pp. 54–65.
14. S. Cleary, "Best Practices in Asynchronous Programming," Microsoft, 2015; http://msdn.microsoft.com/en-us/magazine/jj991977.aspx.

FOCUS: REFACTORING

Refactoring—a Shot in the Dark?

Marko Leppänen, Tampere University of Technology

Simo Mäkinen, University of Helsinki

Samuel Lahtinen and Outi Sievi-Korte, Tampere University of Technology

Antti-Pekka Tuovinen and Tomi Männistö, University of Helsinki

// An interviewed group of software architects and developers considered refactoring to be valuable but had difficulty explaining and justifying it to management and customers. They didn't use measurements to quantify the need for or impact of refactoring. //

REFACTORING IS CONSIDERED an implicit part of agile software development. When working on code, developers are expected to constantly improve its quality—for example, maintainability. However, developers are often pressured to use their work time to add features rather than refactor the code. The increasing pace of business requires quickly delivering a steady stream of new functionalities.

To study the state of refactoring in the industry, we interviewed software architects and developers at prominent Finnish software companies. We aimed to find out what they considered to be refactoring, what benefits and challenges they saw in it, and how they incorporated refactoring into their development workflows. We also wanted to learn whether the companies' decision to employ refactoring was based on facts or just a shot in the dark.

Study Description

We looked for information-rich cases of software in companies in varied industry domains and with varied process maturity. We conducted semi-structured, themed interviews in nine companies with 12 senior software architects or developers in 10 sessions. The interviewees had a minimum of 10 years' experience.

We carried out the interviews in Finnish in 2015, with one or two interviewers and interviewees present. Audio was recorded and transcribed. We analyzed the data qualitatively following Per Runeson and Martin Höst's case study guidelines for identifying common themes from the transcripts and annotating the respective sections of text with specific keywords.[1]

Table 1 presents the companies' characteristics. The industry domains ranged from digital-services production in application areas such as weather information, public transportation, and Web services, to embedded industrial automation and machine control. The companies' sizes differed, but the interviewees generally worked in small development teams of fewer than 10.

Software development teams performed product development internally, performed it externally for customers and users, or contracted work to other parties, which can have implications for accepted refactoring practices.

Regarding development practice maturity, we roughly mapped the involved companies according to the Stairway to Heaven model.[2] The model conceptualizes organizational software development capabilities as five steps:

- A—traditional development,
- B—agile R&D organization,
- C—continuous integration,
- D—continuous deployment, and
- E—R&D as an innovation system.

0740-7459/15/$31.00 © 2015 IEEE

TABLE 1

The companies in the study and their refactoring practices.

Company	Domain	Product	Stairway to Heaven level*	Test automation	Refactoring/development ratio estimate	Refactoring frequency	Deployment frequency
A	Machine control	Internal, outsourced	A	Some (some builds could be broken for weeks)	50/50	Weekly	Monthly
B	Medical monitoring	Contracted	A	Yes	95/5	Almost daily	Every two years
C	Web services	Contracted	B	No	10/90 to 15/85	Daily	Once per project
D	Weather information	External	C	Yes	20/80 to 30/70	Daily	Weekly to biweekly
E	Web services	Contracted	C	Yes	33/67	Daily	Once a month
F	Web services	Contracted	C	Yes	5/95 to 15/85	Incorporated in feature development	Once a year
G	Transportion service on demand	External	C	Yes	15/85	Daily	Bimonthly
H	Industrial automation	Internal	D	Yes	10/90 to 20/80	Daily to weekly	Several years
I	Content management system product	External	D	Yes	20/80	Daily	10 times a day
J	Web services	Contracted	E	Yes	20/80	Every other day	Daily

*A = traditional development, B = agile R&D organization, C = continuous integration, D = continuous deployment, and E = R&D as an innovation system.

We used the Stairway to Heaven level to get a more representative sample of the companies regarding the adoption of development practices supporting continuous delivery and deployment. Seven companies were at step C or higher.

Most companies employed test automation, which provides the essential safety net for refactoring. Although internal development cycles could be fast, production releases were far rarer, so that the deployment frequency could be up to several years, depending on the domain. Also, Table 1 presents the interviewees' impressions regarding the refactoring effort in their recent projects. Refactoring was common; typically, the interviewees spent a fifth of their working time on it.

Our research protocol helped establish a solid chain of evidence for the conclusions made in this article. Although we tried to gather rich experience-based data from a variety of companies with divergent domains and process maturity, the sampling of companies was based mostly on convenience sampling and the availability of cases identified in the network of familiar companies. In the companies, we looked for cases with a suitable history in terms of refactoring and individuals who had been involved in software development for a long time with field experience in refactoring. We acknowledge that single case studies, such as this one, are limited to describing the situation in the cases, whereas theory creation or confirmation of hypotheses requires more studies. So, we concentrate on reporting insights from the cases and avoid

quantitative aggregates because of their lack of clear representativeness.

Understanding Refactoring and Its Causes

The interviewees generally understood refactoring as changing the code's structure without altering the program's perceived behavior. This resonates with the idea originally presented by Martin Fowler.[3] However, the perceptions of the magnitude and nature of structural changes representing refactoring were equivocal. Interviewees usually reserved the word "refactoring" for restructuring and redesigning the system, as in preparatory or planned refactoring.[4] (For more on these types of refactoring, see the sidebar.) They didn't consider daily small changes—for example, method renaming and moving the code around a bit—as refactoring. As example cases, the interviewees described tasks whose work amount would equal several days. So, the practical use of the term didn't fully align with Fowler's original meaning. Thus, you should be aware that the use of the term "refactoring" in this article reflects our interviewees' understanding of it.

The interviewees nominated the constant rush as the leading cause of refactoring because there wasn't enough time to create good code. Developers reported knowingly taking shortcuts and acknowledged that the choices might come back to haunt them.

Refactoring needs also arise because people learn. While engaged in daily software development, developers gain new insights and see existing code and structures in a new light. The interviewed developers felt that initially, they lacked the technical skills or understanding about the domain necessary to make good decisions. In addition, as a developer at a start-up mentioned, refactoring lets you try to learn whether a certain way of coding would be more appropriate than the current implementation, providing the opportunity for personal growth.

Architectural constraints might still be vague at the outset, leading to refactoring. Several interviewees replied that sometimes they had a less than perfect picture of what was to be built because the information between developers and customers didn't flow smoothly. The interviewee from company A, whose development is heavily outsourced, said that distributed teams, possibly separated by a language barrier, contributed to the miscommunications. The design constraints also changed because the world evolved, resulting in subsequent changes in requirements.

> Developers reported knowingly taking shortcuts and acknowledged that the choices might come back to haunt them.

In contrast with what we just discussed, refactoring can be a philosophy of developing software. A developer mentioned that he promoted and practiced a culture in which developers gradually constructed the code better and better. Refactoring in this sense became a part of the normal development flow.

Are Refactoring Decisions Rational?

A tug of war seemed to exist between feature development and refactoring. New features generate revenue, whereas refactoring is an investment in the developers' experience. However, the interviewees clearly stated that time spent on refactoring usually paid itself back in future development. New features can be rapidly implemented if the code quality is already good. However, the need for refactoring seemed to arise only from the developers' tacit knowledge, and no good measures for quantifying the return on investment existed. Making purely rational decisions was difficult.

Overall, the interviewees had difficulty estimating the workload needed for refactoring when developing new features. If the code quality was bad, unexpected things could bog down the development even in tasks perceived as easy and small. In company B, a relatively small change, adding a battery charge level to a display, took three and a half weeks instead of the expected few hours. In relation to this, a few companies tracked the team velocity as a proxy metric for code quality because velocity tends to decrease quickly when quality problems occur in the code.

Table 1 presents the ratio of the time spent on refactoring to the time spent developing new features. This could serve to roughly estimate code quality because the time spent on refactoring was more pronounced in systems in which developers felt the overall quality was low. Of course,

REFACTORING

Refactoring is basically a straightforward technique. However, researchers have introduced additional concepts that paint a more complex picture of refactoring in development workflows.

Martin Fowler defined refactoring as making behavior-preserving design improvements right after adding and testing new functionality.[1] But developers can also refactor code opportunistically—recurrent "upkeep refactoring" has been called "floss refactoring," "litter-pickup refactoring," or "comprehension refactoring."[2–4] Adding new functionality might trigger "preparatory refactoring" that makes the changes easier.[4]

Without floss refactoring, developers might need to apply "root canal refactoring" or "planned refactoring"—that is, change large parts of the code, which requires specific backlog items.[3,4] Fowler considered a recurring need for planned refactoring as a bad smell.[4] He suggested that architectural changes could be done as "long-term refactoring," in which developers gradually and knowingly migrate the system to a new architecture by applying opportunistic changes. Michael Stal recommended recurrent refactoring of the architectural design directly.[5] Although everyday work can easily include upkeep refactoring, large-scale or architectural refactoring is difficult to justify to stakeholders, especially customers, who might not understand the nature of software.[2,4,5]

The views on refactoring have been mixed. Some people see refactoring as beneficial, even a success factor, whereas others still strongly advocate the "if it ain't broke, don't fix it" mentality.[6] Developers feel that refactorings' benefits are difficult to measure and sell to management.[7] Developers don't care to make quality measurements; they want to tackle the concrete problems in code, rather than trying to improve internal code quality metrics.[8] There are conflicting results if refactoring actually improves code metrics.[9]

Refactoring poses risks, and experienced architects might even aim to minimize it.[10] Risks and difficulties increase as the system size increases,[9,11] and careless refactoring might introduce a significant number of faults.[12]

References

1. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
2. J. Chen et al., "Refactoring Planning and Practice in Agile Software Development: An Empirical Study," *Proc. 2014 Int'l Conf. Software and System Process* (ICSSP 14), 2014, pp. 55–64.
3. E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.
4. M. Fowler, "Workflows of Refactoring," 2014; http://martinfowler.com/articles/workflowsOfRefactoring.
5. M. Stal, "Refactoring Software Architectures," *Agile Software Architecture: Aligning Agile Processes and Software Architectures*, M.A. Babar, A.W. Brown, and I. Mistrik, eds., Morgan Kaufmann, 2014, pp. 63–82.
6. M. Lindvall et al., "Agile Software Development in Large Organizations," *Computer*, vol. 37, no. 12, 2004, pp. 26–34.
7. M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study on Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 50.
8. G. Szöke et al., "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?," *Proc. 14th IEEE Int'l Working Conf. Source Code Analysis and Manipulation* (SCAM 14), 2014, pp. 95–104.
9. R. Moser et al., "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," *Balancing Agility and Formalism in Software Eng.*, LNCS 5082, Springer, 2008, pp. 252–266.
10. F. Buschmann, "Gardening Your Architecture, Part 1: Refactoring," *IEEE Software*, vol. 28, no. 4, 2011, pp. 92–94.
11. B. Boehm, "Get Ready for Agile Methods, with Care," *Computer*, vol. 35, no. 1, 2002, pp. 54–59.
12. G. Bavota et al., "When Does a Refactoring Induce Bugs? An Empirical Study," *Proc. IEEE 12th Int'l Working Conf. Source Code Analysis and Manipulation* (SCAM 12), 2012, pp. 104–113.

this ratio also reflects what counted as refactoring. If all code polishing was counted, this figure could be significantly higher, as several interviewees affirmed. The developers were usually satisfied with the code quality if refactoring took less than 20 to 30 percent of their time. Strikingly, in company B's project, refactoring took 95 percent of the time. Consequently, that company's interviewee mused that it would have been more efficient had the project been scrapped and restarted from scratch.

The interviewees primarily considered refactoring as an internal issue of the development team. Most interviewees included refactoring in the effort estimate of a task. However, it was a different story when the need for a larger restructuring or redesign arose. In those cases, most interviewees asked management for permission. Such a task

was usually perceived to take more than one day.

In customer projects, refactoring didn't "bring cash to the house," so convincing customers to pay for it was sometimes difficult. We speculate that if you have to communicate the need for refactoring to the customer, it becomes a trust issue. Even though agile methodologies build on mutual trust, they also promote visibility with frequent demos. Because refactoring is by definition something that doesn't change the code's behavior, its benefits are difficult to show to customers.

> If you have to communicate the need for refactoring to the customer, it becomes a trust issue.

However, several interviewees said that the more technical background the customer's representative had, the easier it was to convince him or her to take quality issues into account. One interviewee said that refactoring had to be demystified because the concept wasn't self-evident to customers. In committing stakeholders to their way of working, developers can introduce refactoring as a significant tool for continuous quality.

If the development was for in-house purposes only or there wasn't so much hurry to produce new features, refactoring was more common. However, even in those cases, it didn't pay off to change something that remained static for the foreseeable future. So, refactoring was usually paired with an actual functional change. One attribute affecting the decision making was that if the software's expected development lifecycle was short, it didn't pay to refactor the code. It seemed that monolithic software underwent more refactoring than limited-life microservices, which tended to eventually become static.

Why Bother?

Refactoring doesn't bring visible changes for users in terms of added functionality, so what does refactoring improve? Our interviewees pointed out several benefits.

Regardless of the results for users, the code level changes are visible to the developers, who have to read, understand, and modify the code. Almost all interviewees agreed that by refactoring, they were trying to make future development easier—not only improving the code's maintainability but also easing the addition of features. A software architect at company J said,

> *Then another thing is to make future development possible altogether, so that we don't end up in a situation in which maintenance or further development would be impossible.*

Refactoring is obviously done by developers for developers. Most interviewees saw that refactoring can improve how developers read and understand code. This is comprehension refactoring as Martin Fowler defined it.[4] One developer mentioned that you write code not only for yourself or the compiler but also for other developers who might need to take care of the code later and adopt the style the code was written in. For example, company D's interviewee stated that code was more readable and easier to understand when the method names were clear and development followed the single-responsibility principle.

Although the developers—similarly to a large survey study at Microsoft[5]—wanted to keep the code maintainable and readable through refactoring, they didn't seem to have a good way to tell whether they had succeeded. The interviewees didn't rate code metrics high in terms of utility. So, actions were driven more by intuition than metrics. Readability and understandability could also be subjective; for one architect, that meant keeping things simple by reducing the amount of code.

Several developers said that increasing generalization improved code reusability. They saved effort because they needed less added code for each new case. One developer explained that template functions in C++ were a good example of this. According to another developer, splitting classes made the code more modular, enabling more flexible work distribution among teams and work sites, compared with one gigantic class.

Algorithmic performance could trigger refactoring, too. For company H in the industrial-automation domain, optimizing program execution speed was a key driver for refactoring. In addition, the company wanted refactoring to increase reliability and robustness. The refactoring needs were thus potentially domain- and application-specific, and the desired quality attributes weren't always the same.

One architect stated that refactoring was required when developers felt uneasy about extending the code. The code's perceived internal complexity could increase gradually over time, and refactoring could help fight it. Software decay emerged as an issue because it was challenging to foresee in which direction the software would evolve and how to program the code properly when starting development.

Refactoring could also keep the developers' spirits high. Several interviewees noted that giving developers the chance to refactor increased their motivation and satisfaction. Continuous improvement could create an atmosphere in which developers felt generally happier.

What Could Possibly Go Wrong?

The interviewees' primary fear was that refactoring might break the code somewhere else. Despite good intentions, refactoring-related changes might introduce defects that go unnoticed no matter how complete the test suites are; a concern that had been shared earlier at Microsoft.[5]

An architect we interviewed mentioned that modifying interfaces and method signatures could be particularly challenging when external services depended on the interfaces. Refactoring such code might break the other end and cause harm for the external provider. So, developers should think twice before refactoring; this motivates the need for clear interface deprecation policies that are communicated to all parties. This is in line with previous arguments that underlined the difficulty of changing core services' fundamental characteristics.[6]

Several interviewees acknowledged that success wasn't always guaranteed; code structures might not have improved at all after refactoring, or the perceived internal code quality might have actually degraded. An architect from company J stated,

> *One thing could be starting a huge refactoring, if we feel there's a lot of technical debt here. We then start refactoring, which is done for weeks and months, only to eventually notice that it's never going to get fixed, and we have to give up at some point. We realize that we just did a month's work for nothing because we weren't able to fix anything sensible.*

Although restricted to structural and other behavior-preserving changes, refactoring is still programming. Code is added, modified, and deleted, which all takes time. According to the interviewees, it wasn't easy to see beforehand whether this effort would later provide added value in some form. A software architect from company F stated fittingly,

> *It's like a wish that when I refactor, I'll save some time in the future. It's sort of an investment in the future that might never realize.*

The challenge is to improve the code and not spend time on changes from which few people or no one will benefit. When making the code extendable and easy to read, understand, and maintain, developers should somehow be able to predict future needs. This might be why several companies tried to tie refactoring with a functional change.

Refactoring Tools and Metrics

We also asked about tool use. We anticipated the tools to fall into two categories: ones that help refactoring and others that indicate the need for refactoring. However, most of the companies had no tools in the latter category.

Several interviewees said that the essential toolset for refactoring was a good version control system and a good set of automated tests on a continuous-integration server. The version control system ensured that refactoring-related changes didn't interfere with the rest of the team. Good automated tests gave some certainty that the changes didn't affect the code's functionality. So, step C (continuous improvement) seemed to be the watershed level. However, some interviewees weren't satisfied with the tests' quality or coverage.

> Giving developers the chance to refactor increased their motivation and satisfaction.

Although continuous-improvement practices support effective refactoring, interviewees were slightly concerned about fitting bigger changes into the practice of committing often to a working codebase. A large nonincremental restructuring of the code could break down the system. However, branching in version control

helped achieve such changes. Still, developers usually postponed huge restructurings until the software had stabilized so that no more significant changes were required. At this point, a stable version of the system could be available all the time, and refactoring could be done so that no new versions of the system were published.

> It's often easier to write a new implementation instead of modifying an existing one.

The interviewees agreed that no tool could actually help in the refactoring activity itself. Even though developers found refactoring tools in certain IDEs, such as Eclipse, useful, they didn't find them necessary for development. They saw the tools at best as a way to save some time and avoid errors introduced by manual work, but they had little trust in automation.

No good tools for quantifying refactoring needs seemed to exist; the interviewees usually deemed metrics unessential because they appeared to measure nothing useful. One interviewee mentioned that he and his colleagues tried to find suitable metrics but usually ended up in a worse situation because "nothing else mattered [to their managers] than the one number the tool produced." However, the companies involved in safety-related software (A, B, and H) used static code analysis to find problematic parts of the code. Nevertheless, this didn't give them any metrics, just a list of things to fix as soon as possible. This resonates with Gábor Szöke and his colleagues' finding that developers tend to target concrete problems, not improve metrics or remove antipatterns.[7]

However, some interviewees felt that tools or metrics for measuring technical debt and the need for refactoring could be useful. In customer projects, metrics for internal quality could help prove the need for refactoring to a reluctant customer. Internal projects might benefit from a mechanism to grade the code and to avoid trivial errors in code quality.

One developer mentioned that the Code Climate tool had been useful. Another interviewee told us he used a code grader for his hobby project and perceived no significant quality increase after refactoring the code from a low grade to the highest one. Still, no interviewee said that his or her current project used some method to grade code quality.

Tips and Tricks Learned

The interviewees found the following practices and techniques useful in their everyday work.

Establish a safety net. Sensible version control practices and a meaningful test suite are essential to comfortably make changes to working code.

Use revision control to log refactoring operations. This will reveal your refactoring steps, and you can track the rationale of refactoring. If a new feature requires refactoring, do refactoring first, commit it separately, and add the feature later.

Check and revise unit tests before refactoring. Refactoring often invalidates unit tests, hindering testing.

Use code reviews for major refactoring operations.

Break it and fix it. It's often easier to write a new implementation instead of modifying an existing one. Remove the implementation fragments you're going to refactor and write new ones. You need a reliable test suite to test that the functionality matches the previous implementation.

Do small refactoring tasks whenever needed. You can perform minor refactoring on the fly while fixing bugs or adding features. This helps maintain the code base's quality, decreases the slow deterioration of the code, and even improves the code base in long run.

Include small refactoring tasks in your task estimations. This lets you maintain the code base's quality and decreases technical debt.

Schedule your refactoring. Don't perform major refactoring tasks right before a release date. If you're using continuous deployment, don't do larger refactoring tasks at the end of the week, to avoid emergency bug fixes on weekends.

Have a dedicated refactoring day each week. Especially during early development, exact requirements are rare, and the optimal solutions for the final product are unknown. Use the first sensible solution, "code with the flow," and move forward, instead of trying to figure out the optimal solution. Spend your refactoring day to go through the code, tidy it up, and refactor the parts that turn out to be poor.

Involve technology-oriented people from the customer organization.

This makes it easier to communicate your refactoring needs.

Have small teams working closely together so that communication is easy and refactoring can occur without a major hassle. Exclaiming "Don't touch this component for a couple of hours; I'm doing a major rewrite on it" is sufficient.

To measure refactoring needs, consult the programmers. They live with the code every day; they know its good and bad parts.

Benefits	Risks
• Easier future development • Understandability • Reuse • Improving quality attributes such as performance • Boosting morale and motivation	• Breaking something • Causing externally visible changes • Worsening the code quality • Wasting time and effort

FIGURE 1. Refactoring's benefits and risks, according to the interviewees.

The interviewees seemed to be well aware of refactoring. However, as we mentioned before, they seemed to have reserved the word for quite large restructurings, and they saw opportunistic refactoring as an integral part of development, not a separate programming mode.

The companies were committed to delivering features, and refactoring was a second-class citizen. The cost of work and pressing deadlines often required postponing or reducing refactoring, much to the developers' dismay. One contributing factor was the difficulty of selling refactoring, especially to business-oriented customer representatives. One interviewee declared, "It is a part of the customer representative's, product owner's, or product manager's competence to buy refactoring." To make matters worse, there was no easy way to give refactoring a return-on-investment metric.

As a whole, the developers didn't use maintainability metrics because they felt they knew the refactoring needs without measurements. They usually knew how to write decent code, so checking coding conventions automatically was unnecessary. If used, code reviews helped catch the most obvious problems.

Most interviewees said that understandability was among refactoring's top three benefits (see Figure 1), but in the end, it seemed that refactoring mostly helped maintain the interviewees' sanity. The developers acknowledged that they planned to add features to the code or maintain it anyway, so it paid off to produce code that was as good as possible in the first place.

Surprisingly, the software's application domain had little effect on refactoring practices. Deadline pressure seemed to weigh more. Even process maturity wasn't a huge determiner of refactoring. The biggest enablers for refactoring were that continuous-improvement practices and automated tests were in place.

In the end, refactoring wasn't a shot in the dark. The developers had an intuition of the code structures that needed fixing, and they appreciated the potential benefits. However, although they had a pretty good understanding about what they were aiming at, there were obstacles that could block the shot. Also, it could be difficult to tell whether the shot was a hit or a miss.

Acknowledgments

TEKES (the Finnish Funding Agency for Innovation) supported this article as part of the N4S (Need for Speed) Program of DIGILE (the Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

References

1. P. Runeson and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," *Empirical Software Eng.*, vol. 14, no. 2, 2009, pp. 131–164.
2. H.H. Olsson, H. Alahyari, and J. Bosch, "Climbing the 'Stairway to Heaven'—a Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software," *Proc. 38th EUROMICRO Conf. Software Eng. and Advanced Applications* (SEAA 12), 2012, pp. 392–399.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. M. Fowler, "Workflows of Refactoring," 2014; http://martinfowler.com//articles/workflowsOfRefactoring.
5. M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study on Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 50.

6. D. Turk, R. France, and B. Rumpe, "Limitations of Agile Software Processes," *Proc. 3rd Int'l Conf. Extreme Programming and Flexible Processes in Software Eng.* (XP 02), 2002, pp. 43–46.
7. G. Szöke et al., "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?," *Proc. 14th IEEE Int'l Working Conf. Source Code Analysis and Manipulation* (SCAM 14), 2014, pp. 95–104.

ABOUT THE AUTHORS

MARKO LEPPÄNEN is a doctoral student in the Tampere University of Technology's Department of Pervasive Computing. His research focuses on software architectures, agile methodologies, and continuous delivery. Leppänen received an MSc in electrical engineering from the Tampere University of Technology. Contact him at marko.leppanen@tut.fi.

SIMO MÄKINEN is a doctoral student in the University of Helsinki's Department of Computer Science. His research interests include software development methodologies, software quality assurance practices and tools, and software architectures. Mäkinen received an MSc in computer science from the University of Helsinki. Contact him at simo.v.makinen@helsinki.fi.

SAMUEL LAHTINEN is an assistant professor in the Tampere University of Technology's Department of Pervasive Computing. His research interests include software architecture, software design and development, and software engineering education. Lahtinen received a PhD in software engineering from the Tampere University of Technology. Contact him at samuel.lahtinen@tut.fi.

OUTI SIEVI-KORTE is an assistant professor in the Tampere University of Technology's Department of Pervasive Computing and a postdoctoral researcher sponsored by the Academy of Finland. Her current research project focuses on global software development and software architecture, project planning and management, and the application of search-based mechanisms in that context. Sievi-Korte received a PhD from the University of Tampere. Contact her at outi.sievi-korte@tut.fi.

ANTTI-PEKKA TUOVINEN is an associate professor of software engineering in the University of Helsinki's Department of Computer Science. His research interests include software architecture, programming languages, and software quality. Tuovinen received a PhD in computer science from the University of Helsinki. Contact him at antti-pekka.tuovinen@helsinki.fi.

TOMI MÄNNISTÖ is a professor of software engineering in the University of Helsinki's Department of Computer Science. His research interests include software architectures; variability modeling, management, and evolution; configuration knowledge; and flexible requirements engineering. Männistö received a PhD in computer science from the Helsinki University of Technology (now Aalto University). He's a member of the IFIP TC2 Working Group 2.10 Software Architecture, the IEEE Computer Society, and ACM. Contact him at tomi.mannisto@helsinki.fi.

FOCUS: REFACTORING

Database Refactoring

Lessons from the Trenches

Gregory Vial, HEC Montreal

// A software development firm refactored a database for a logistics application. The project employed clear database development conventions and packaged the documentation in the database itself. The authors discuss key lessons, including the development of a tool for common refactoring operations. //

REFACTORING IS INTEGRAL to software development and has been prescribed in methods such as Extreme Programming.[1,2] Of the many available refactoring patterns, the most frequently used ones are embedded in IDEs to streamline development.[3] Although refactoring originally focused on object-oriented programming,[4] researchers have advocated its use in other software development areas.

Here, I focus on refactoring relational databases.[5] (For a look at why research is lacking on that topic, see the sidebar.) To help showcase refactoring's relevance and benefits, I provide an experiential report from a small North American software development firm. Although this report is anecdotal and the results are specific to this case study, I hope that outlining the key lessons learned stirs further interest on this topic.

Case Description

Logicompo (not the company's real name) has 20 employees, including five developers. Since the late 1990s, Logicompo has been developing and selling Optilogis (not the product's real name), a highly configurable three-tier application that helps clients optimize logistics processes. It's built on legacy components; server components interact with the database, exposed as COM (Component Object Model) and COM+ interfaces used by client applications. Its continued use by several Fortune 500 companies as part of their mission-critical applications is a testament to its quality and robustness.

The average Optilogis database uses approximately 60 Gbytes. Developers modify the schema by writing SQL scripts that they commit into Logicompo's source control repository. Because of Logicompo customers' heterogeneous environments, the company supports multiple concurrent versions of its application on multiple versions of Microsoft SQL Server (MSSQL). It develops new versions primarily in response to customers' requests, as well as other features it puts on its product roadmap.

In the late 2000s, Logicompo included the standardization of Optilogis's database schema in the product roadmap. Three main factors motivated this decision:

- A Logicompo developer volunteered to take charge of Optilogis's database development best practices.
- Customers often requested documentation for the database.
- The schema lacked coherence and readability, which at times reflected negatively on the application's quality.

Regarding the last point, because database objects aren't packaged into binaries like application code, their content—unless encrypted—is visible. So, Logicompo designed a set of database standards and guidelines to rename and reorganize years of

0740-7459/15/$31.00 © 2015 IEEE

RELATIONAL DATABASES AND REFACTORING

Relational databases are critical pieces of software that support many applications. However, little research has focused on database refactoring related to application development. For instance, a search in early 2015 for "database refactoring" in IEEE Xplore and the ACM Digital Library yielded 16 articles, including reviews of the most popular work on the topic, *Refactoring Databases: Evolutionary Database Design*.[1]

This paucity of research has two causes. First, database evolution typically follows software evolution, rather than the other way around, and is therefore considered part of software refactoring.[2] Second, databases aren't compiled or built like applications and are only loosely coupled with the applications they support. This renders database refactoring difficult and automation hard to achieve, unlike in IDEs. For example, the Microsoft SQL Server (MSSQL) client tools don't support refactoring, and code completion has been available only since the 2008 edition of MSSQL. In contrast, refactoring has been available for more than 20 years in Visual Studio.[1]

References

1. S.W. Ambler and P.J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Pearson Education, 2006.
2. C.A. Curino, H.J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: The Prism Workbench," *Proc. VLDB Endowment*, vol. 1, no. 1, 2008, pp. 761–772.

accumulated SQL objects. This included more than 170 tables.

Database refactoring can involve modifying an application's code base[5] (an opportunity that benefited Logicompo). However, some researchers advocate using frameworks to handle database evolution without requiring source code modifications.[6] Logicompo employees acknowledged the risks associated with this project given their contractual obligations. They decided to perform refactoring during off-peak times without a specific target version. To create a robust foundation, they elicited and approved two key principles: define standards and document within the database.

Define Standards

This principle focuses on studying the application's domain and its use of various types of SQL objects to determine a series of conventions forming a pattern applicable across the schema. The idea is to design a standard to provide enough structure and improve the schema's readability without constraining developers. The naming conventions selected (see Table 1) are simple yet comprehensive enough to cover a range of scenarios and are still in place at Logicompo.

Document within the Database

Previously, developers had handled clients' requests to view documentation for the Optilogis database by using Microsoft Visio to produce an entity–relationship diagram. However, the developers often stored documentation for tables, columns, and other SQL objects in outdated spreadsheets.

We decided to use MSSQL's extended properties as a storage space for metadata, such as object descriptions and documentation.[7] Logicompo's policy is simple: every SQL object in the database must have at least one extended property documenting its purpose. A few key patterns facilitate creating this documentation (for example, attributes that are part of a primary key should indicate so in their extended property). The objective is to ensure that retrieving and maintaining the database documentation are easy.

Although this principle initially seemed sound to developers and project management staff, its implementation proved problematic. This is because quickly viewing and manipulating multiple extended properties is cumbersome, whether you use MSSQL's client tools or Transact-SQL (T-SQL) statements (see Figure 1).

Five Key Lessons

Dealing with this project's challenges led to the following insights.

Automate Work

Although developers approved the two aforementioned principles, their knowledge of T-SQL and the queries used to refactor SQL objects and manipulate extended properties varied greatly. So, developing features that required modifying the database schema (such as creating new tables) often yielded inconsistencies with the previously designed guidelines. This made initial refactoring

TABLE 1

Naming conventions.

Convention	Example
Use camel case to improve readability.	SITEWAREHOUSE → **SiteWarehouse**
Table names should be always singular.	**Warehouse**
Tables should have a synonym (five letters maximum) for each reference to that table.	Warehouse → **WHSE** Site → **SITE**
Keys should use the table's synonym.	Warehouse → **WHSE_ID, SITE_ID (foreign key referencing the Site table)**
Constraints and objects should follow a simple naming pattern.	Foreign key from the warehouse table to the site table: **FK_WHSE_SITE** Index on the warehouse table's name and country attributes: **IX_WHSE_Name_Country** Stored procedure: **sp_SimpleProcedure**
Database maintenance tables should use a special prefix.	Database maintenance table: **sys_DatabaseMaintenance**

efforts painstaking and error-prone. To alleviate such issues, we created two main tools to support database refactoring and evolution.

The most important tool is Refactor, a proprietary tool that handles refactoring and Optilogis's routine database schema evolution. Refactor is a Win32 application that developers run on their own environment. It connects to their Optilogis database, queries its catalog, and lets them perform a variety of refactoring operations, such as renaming SQL objects. They can also create or alter tables and columns, add indexes and constraints, and generally implement new functionalities or fix bugs. They perform these operations within a single GUI rather than a series of modal forms, such as those offered by MSSQL's client tools.

When developing Refactor, we considered using MSSQL's client API. However, it inefficiently retrieves bulk information (such as extended properties) because it requires multiple calls and iterations over .NET assemblies that can hinder the application's responsiveness.

```
/* Creating a Table 'Test' in the database */
CREATE TABLE dbo.Test(ID [int]);

/* Adding an extended property on the table */
EXEC sp_addextendedproperty
   @name = 'Description',
   @value = 'Test table description.',
   @level0type = 'schema', @level0name = 'dbo',
   @level1type = 'table', @level1name = Test,
   @level2type = NULL, @level2name = NULL;

/* Modifying the extended property on the table */
EXEC sp_updateextendedproperty
   @name = 'Description',
   @value = 'New description.',
   @level0type = 'schema', @level0name = 'dbo',
   @level1type = 'table', @level1name = Test,
   @level2type = NULL, @level2name = NULL;

/* Retrieving the extended property on the table */
/* Other methods exist since MSSQL 2005/2008 */
SELECT objtype, objname, name, value
   FROM fn_listextendedproperty('Description', 'schema', 'dbo', 'table', 'Test', default, default);

/* Note: In MSSQL's client tools, no way exists to view all extended properties of all the
columns of a given table. Those must be viewed one by one in separate modal forms */
```

FIGURE 1. Manipulating extended properties on database objects in Transact-SQL (T-SQL). This process can quickly become cumbersome.

```
/* Assume a user has decided to change a table's synonym from */
/* ABC to XYZ using Refactor. Refactor executes the following */
/* statement to retrieve ABC and its dependencies */
SELECT name, type_desc FROM sys.objects
   WHERE schema_id = SCHEMA_ID('dbo')
   AND name LIKE '%ABC%'
   ORDER BY type;
```

FIGURE 2. A sample lookup of all semantic dependencies for object renaming in T-SQL as used by Refactor to identify dependencies affected by the renaming of an object.

Instead, Refactor executes custom queries using an ActiveX Data Objects (ADO) connection. Refactor embeds Logicompo's standards and guidelines and simply generates SQL scripts based on user input. For example, once a user chooses a synonym for a new table, we can name keys and constraints accordingly, per the standards in Table 1. Refactor also supports automatically creating and documenting common database patterns (such as soft-delete maintenance columns), saving time and helping developers focus on software code.

Developers then can review the Refactor-generated scripts and apply them to their development database for testing. Integration with Logicompo's source control repository lets Refactor save scripts with the proper naming convention and sequence number for the build system. Refactor has proven invaluable because it makes some refactoring work transparent. It also ensures that the database schema's evolution is tightly integrated within the source control system, much like the rest of the application code.

The second tool is a custom stored procedure that validates the structure of SQL objects against Logicompo's guidelines and standards. It scans a database and reports, either interactively or in a dedicated table, any violations of those standards. Although this procedure doesn't perform refactoring per se, it does report on object-renaming opportunities and other refactoring patterns that could be applied. Embedded in a build system, calls to this procedure can generate automated reports that are emailed to developers for corrections.

These tools have facilitated our initial refactoring project's progress and helped minimize database debt. With Refactor, developers abide by database standards without having to refer to them, making it easier for them to use refactoring daily (much like software refactoring is embedded in IDEs). Refactor also removes the need for external database documentation. Database documentation and guidelines live in the database and are enforced by warnings raised by Refactor and the validation procedure when users violate them. Refactor's integration with Logicompo's source control system lets Logicompo treat database evolution as a series of sequential patches applied during an upgrade. Optilogis databases use extended properties and maintenance tables to keep track of current versions and scripts executed in the past, to allow fast forwarding to any given version.

Use the Catalog

In databases, the catalog (often implemented as `INFORMATION_SCHEMA` views) generally contains the information needed to find objects and perform simple refactoring operations, such as renaming objects. For instance, it provides a simple way to explore the schema and find object dependencies, whether using referential integrity constraints or pure semantics.

Logicompo uses the catalog to explore those dependencies during refactoring. For example, in MSSQL, the stored procedure `sp_rename` is often used to rename objects such as tables. As per Logicompo's guidelines, renaming a table or column requires changing the names of its associated constraints and keys (see Table 1), something `sp_rename` doesn't handle.

Refactor lets developers rename a table and select a new synonym for it. To generate a comprehensive update script, Refactor executes a query similar to the one outlined in Figure 2. This query looks for objects that share the old synonym. On the basis of the query results, Refactor adds separate `DROP` or `CREATE` statements or calls `sp_rename` to the script to rename those objects.

The catalog also provides a toolbox to extract the database schema's structure and generate documentation for developers, technical staff, and clients. We coded simple wrapper functions and stored procedures to extract the database's structure and objects' extended properties (see Figure 3). These calls provide a much-needed level of abstraction for those unfamiliar with the actual execution of queries against the catalog.

They also help return the output in a format that's easy to view in MSSQL's client tools. Logicompo calls those objects in reports installed with Optilogis to let clients' technical staff view and navigate inside the live database schema's structure. This effectively pushes the database schema to the forefront of Logicompo's technical documentation and saves time in bids with potential customers. In comparison, viewing the same information in MSSQL's client tools or retrieving it via SQL Server Management Objects (SMOs) adds an unnecessary layer of complexity.[8]

```
/* Retrieving all column descriptions in one query */
CREATE FUNCTION dbo.fn_getTableColumnsDescription(@aTableName sysname)
RETURNS TABLE AS
RETURN
(
  SELECT
    COL.name, PROP.value
    FROM sys.extended_properties PROP
    INNER JOIN sys.columns COL ON
      COL.object_id = PROP.major_id AND COL.column_id = PROP.minor_id
    WHERE major_id = OBJECT_ID(@aTableName)
    AND minor_id <> 0
    AND class_desc = 'OBJECT_OR_COLUMN'
);
GO
```

FIGURE 3. A wrapper function to retrieve columns' descriptions in tables. This function provides a level of abstraction and helps return the output in an easy-to-view format.

Proceed in Increments, Package Changes Individually

Although ways exist to perform database refactoring without altering the application code, we decided to update Optilogis's code base. This decreased costs for maintaining another layer of database objects (for example, views that translate between old and new object names) and promoted Optilogis's refactoring.

During the initial refactoring project, we created a task in the product backlog for each high-level object (such as a table or stored procedure). This helped us evaluate and implement refactoring on that object. This piecemeal approach provided an easy way to track progress while incorporating this effort into the application's main development. We preferred this to logging hundreds of hours for a single task. It also let nontechnical stakeholders (such as project managers) see our progress. We started with small tables and other objects seldom used by Optilogis to test and refine our new database standards while minimizing rework. We left large tables that are central to Optilogis until the refactoring practices were well understood and considered routine.

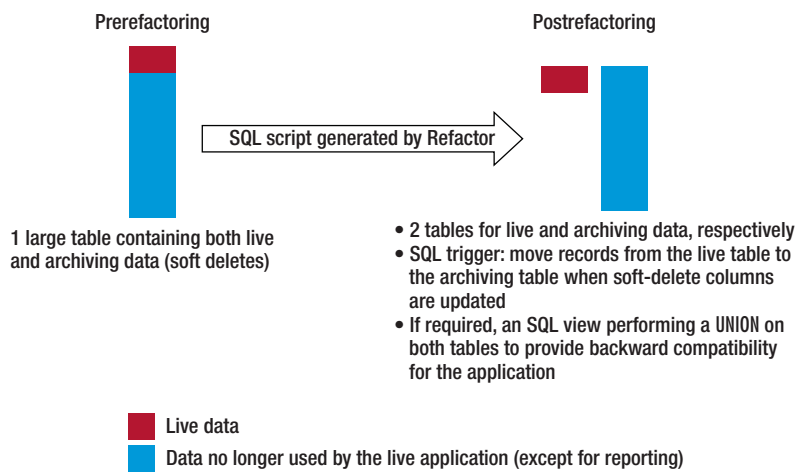
Small commits linked to each task help developers reduce the risk of merge conflicts and minimize rework. When a developer commits an SQL script, others pull it onto their local environment and use Refactor to fast-forward their development environment to the latest version. To ensure that refactoring doesn't break Optilogis, developers inventory all the calls that refer to an object before refactoring it.

Although in some cases starting up Optilogis sufficiently tests refactoring, reporting queries are more complex because of the numerous parameters and filters they contain. As these parameters and filters change on the basis of user input, so does a given query's pattern. Logicompo's testing strategy includes running Optilogis in debug mode, with breakpoints at specific places taken during the initial inventory. It also includes running SQL traces to monitor specific queries made to those objects. In complex scenarios, developers use test-driven development by building small unit tests, which they can automate on their workstations.[9] These strategies, combined with frequent commits, let other developers and automated build systems help test changes.

Increase Performance and Streamline Application Code

Because users rely on Optilogis to perform hundreds or thousands of operations daily, Optilogis's database inevitably grows. Although this situation isn't problematic for online transaction processing (OLTP), it eventually creates issues in other scenarios. For example, full backups take longer to perform and ship to Logicompo for testing and debugging.

In addition, read-only queries used for reporting can cause cache and resource-contention issues that hinder clients' operations



(a)

```
/* Live Table */
CREATE TABLE dbo.Test_Live(ID int, IsDeleted bit);
/* Archiving Table */
CREATE TABLE dbo.Test_Archiving(ID int, IsDeleted bit);
GO
/* Archiving Trigger */
CREATE TRIGGER dbo.TG_Test_Archive ON Test_Live
   AFTER UPDATE
AS BEGIN
   IF UPDATE(IsDeleted) BEGIN
      INSERT INTO Test_Archiving
         SELECT ID, IsDeleted FROM INSERTED WHERE IsDeleted = 1;
      DELETE Live FROM Test_Live Live
         JOIN INSERTED INS ON INS.ID = Live.ID AND INS.IsDeleted = 1;
   END
END
GO

/* View joining both live and archiving tables */
CREATE VIEW vw_Test AS
   SELECT ID, IsDeleted FROM Test_Live
   UNION ALL
   SELECT ID, IsDeleted FROM Test_Archiving
GO
```

(b)

FIGURE 4. Refactoring large tables for live and archiving data. (a) The basic process. (b) The underlying code. When developers modify a live table's structure, Refactor recognizes that table as archivable and modifies the archiving table's structure.

and the application's responsiveness. For example, while refactoring the database, we noted that using soft deletes (which are crucial for debugging and reporting) caused problems for some of Optilogis's OLTP tables. As much as 95 percent of some tables' records would be flagged as deleted, while Optilogis used only a few thousand records for operations purposes.

To alleviate this problem, we modified Refactor to let developers flag tables as archivable. When a table has this option turned on, Refactor sets a flag as an extended property on that table. Then Refactor creates and maintains two tables: a live table and an archiving table. When records are flagged as deleted in the live table, Optilogis no longer owns them, although reports might use them. A database trigger generated by Refactor moves deleted records from the live table to the archiving table. When developers modify the live table's structure, Refactor recognizes that table as archivable and modifies the archiving table's structure too. This process, once coded in Refactor, is transparent to developers and Optilogis because the database engine handles it exclusively. Figure 4 illustrates this pattern's overall design.

This simple pattern has decreased Optilogis's startup time from dozens of seconds to less than 10 seconds for large clients. It also improves the read and write performance and the maintenance of indexes on live tables. Because it's embedded in Refactor, it costs no effort and opens the door to implementing different storage strategies for live and archiving data in the same database. For example, we could have a primary file group for live data and a secondary file group for archiving data.

During the initial refactoring, we uncovered three other types of optimizations for Optilogis. First, we deduplicated some queries by merging methods and adding parameters where applicable. Second, we quickly grasped the need to use parameterized queries to improve query performance and readability. When Optilogis was originally implemented, support for parameterized queries was lacking and seldom used in applications. Refactoring provides the opportunity to revisit those early choices in light of the new affordances made by technological innovations. Third, we wrote stored procedures and functions to bundle calls and save bandwidth while improving execution time where applicable. All these improvements are marginal on a per-transaction basis but yield a significant impact overall.

Finally, database refactoring offers the opportunity to eliminate technical debt in an application. For example, sometimes Optilogis table names aren't defined as constants or are defined in multiple places. While making an initial inventory of the places affected by a refactoring operation, we can find and fix those anomalies. In one instance, we found about 1,000 lines of broken legacy code to support other database management systems (such as Oracle and DB2). Seeing this code provided the opportunity to challenge its relevance to business stakeholders, who eventually let us remove it and improve the code's readability.

Be Sensible

Refactoring a database can have profound implications on the application it supports, its code base, and its performance. Our experience shows that this can't be handled by staff who are dedicated solely to database development. Rather, this requires a combination of database and programming skills to identify changes induced across layers and apply them safely. Working across these two layers requires an excellent understanding of both the database and the application (including usage patterns, number of requests, and so on).

In certain cases, because of Optilogis's architecture, some refactoring patterns didn't yield the benefits we envisioned. For instance, the database contains some level of data duplication to improve query performance and remove the need to perform exhaustive joins across tables. Although we could use numerous technologies (for example, caching and asynchronous patterns) to deal with such issues, these technologies aren't always applicable. This is especially true in legacy architectures such as COM and COM+ and Windows client applications. So, the developers devised a simple rule of thumb: duplication is possible only when retrieval of a value involves more than three join levels. In those cases, the database incurs a certain level of debt, but it's done conscientiously and documented.

Refactor doesn't integrate database refactoring with software refactoring. It also doesn't automate all refactoring patterns. Developers must identify database smells, recognize when to apply refactoring, and acknowledge Refactor's limitations in performing some of those operations. They also must take into account any necessary changes to Optilogis's source code.

> Database refactoring offers the opportunity to eliminate technical debt in an application.

Compared to the refactoring features offered by IDEs, Refactor probably scores weakly. Yet, it still saves dozens of hours of development every month.

What started as a simple database standardization project eventually became a comprehensive database refactoring effort. This approach is ingrained now in how developers maintain the Optilogis persistence layer. Rename, the most commonly used refactoring pattern, was only the first step of what led to a number of changes.[10] These changes improved the readability, maintainability, and performance of Optilogis and its persistence layer.

The most important aspect of this effort was the development of a tool to handle common refactoring operations, which commercial tools lacked at the time. Database engines provide many facilities (such as the catalog) that help implement refactoring patterns and reduce the risk of technical debt. In line with agile practices, an incremental approach lets Logicompo reflect on its practices and design refactoring patterns based on its environment's

TABLE 2

Database-refactoring categories used by Logicompo.

Refactoring category[5]	Example	Supported by Refactor	Use at Logicompo	No. of occurrences during initial refactoring
Structural: changes to definitions of tables or views	Moving a column	Yes	Frequent	>50
Data quality: changes that improve the database information's quality	Adding default values to columns	Yes	Frequent	>150
Referential: changes that ensure that referenced rows exist or are removed on the basis of their need to be in the database	Implementing cascading deletes between parent–child tables	No	Infrequent	<10
Architectural: changes that improve how external programs interact with the database	Moving some application code in the database to let other applications use that code	No	Infrequent	<50
Method: changes to methods (such as functions or stored procedures) in which refactoring patterns also apply	Renaming a stored procedure	Yes	Occasional	>50
Nonrefactoring transformation: changes to the schema that alter the database semantics.	Adding a new column to a table	Yes	Very frequent	>250

specificities.[2,11] Database refactoring forces developers to question design choices and consider the database schema as an integral part of Optilogis, rather than a mere storage engine.

In hindsight, much of our experience follows the categories that Scott Ambler and Pramod Sadalage described (see Table 2).[5] Whereas the initial need to refactor the entire schema called for "root canal" refactoring, our pace was more akin to "floss" refactoring.[12] Although Optilogis's developers now all perform database refactoring, it began with one developer spearheading the effort. We accomplished these tasks over the better part of three years, with about 300 hours of work.

Logicompo's experience with database refactoring has paid off in the development of other applications that also use Refactor. Developers built these applications with database refactoring in mind from day one. Besides helping developers, this approach facilitates project staff's ad hoc queries to investigate issues. Being able to provide current and prospective customers with comprehensive documentation—that they can view and interact with at their fingertips—also has been a major selling point.

Acknowledgments

I thank the case study company employees for sharing their experience and the anonymous reviewers for their insightful comments. The Fonds de Recherche du Québec—Société et Culture (FRQSC) and the HEC Montréal Chair in Strategic Management of Information Technology supported this research.

References

1. T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, 2004, pp. 126–139.
2. K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, 2004.
3. M. Fowler, *Catalog of Refactorings*, 2013; http://refactoring.com/catalog.
4. W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1992; http://hdl.handle.net/2142/72072.
5. S.W. Ambler and P.J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Pearson Education, 2006.
6. C.A. Curino, H.J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: The Prism Workbench," *Proc. VLDB Endowment*, vol. 1, no. 1, 2008, pp. 761–772.
7. "Using Extended Properties on Database Objects," Microsoft, 2015; https://technet.microsoft.com/en-us/library/ms190243(v=sql.105).aspx.
8. *SQL Server White Paper: SQL Server 2008 Compliance Guide*, white paper, Microsoft, 13 Nov. 2013; www.microsoft.com/en-us/download/details.aspx?id=6808.
9. S.W. Ambler, "Test-Driven Development of Relational Databases," *IEEE Software*, vol. 24, no. 3, 2007, pp. 37–43.
10. E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know

It," *IEEE. Trans. Software Eng.*, vol. 38, no. 1, 2012, pp. 5–18.
11. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1997.
12. E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, 2008, pp. 38–44.

ABOUT THE AUTHOR

GREGORY VIAL is a PhD candidate in HEC Montreal's Department of Information Technology. His research interests are outsourcing, systems development practices and methodologies, and relational and dimensional databases. Vial received an MSc in information technology from HEC Montreal. Contact him at gregory.vial@hec.ca.

POINT

Refactoring Tools Are Trustworthy Enough

John Brant

Refactoring tools don't have to guarantee correctness to be useful. Sometimes imperfect tools can be particularly helpful.

A COMMON DEFINITION of refactoring is "a behavior-preserving transformation that improves the overall code quality." Code quality is subjective, and a particular refactoring in a sequence of refactorings often might temporarily make the code worse. So, the code-quality-improvement part of the definition is often omitted, which leaves that refactorings are simply behavior-preserving transformations.

From that definition, the most important part of tool-supported refactorings appears to be correctness in behavior preservation. However, from a developer's viewpoint, the most important part is the refactoring's usefulness: can it help developers get their job done better and faster? Although absolute correctness is a great feature to have, it's neither a necessary nor sufficient condition for developers to use an automated refactoring tool.

Consider an imperfect refactoring tool. If a developer needs to perform a refactoring that the tool provides, he or she has two options. The developer can either use the tool and fix the bugs it introduced or perform manual refactoring and fix the bugs the manual changes introduced. If the time spent using the tool and fixing the bugs is less than the time doing it manually, the tool is useful. Furthermore, if the tool supports preview and undo, it can be more useful. With previewing, the developer can double-check that the changes look correct before they're saved; with undo, the developer can quickly revert the changes if they introduced any bugs.

Often, even a buggy refactoring tool is more useful than an automated refactoring tool that never introduces bugs. For example, automated tools often can't check all the preconditions for a refactoring. The preconditions might be undecidable, or no efficient algorithm exists for checking them. In this case, the buggy tool might check as much as it can and proceed with the refactoring, whereas the correct version sees that it can't check everything it needs and aborts the refactoring, leaving the developer to perform it manually. Depending on the buggy tool's defect rate and the developer's abilities, the buggy tool might introduce fewer errors than the correct tool paired with manual refactoring.

Even when a refactoring can be implemented without bugs, it can be beneficial to relax some preconditions to allow non-behavior-preserving transformations. For example, after implementing Extract Method in the Smalltalk Refactoring Browser, my colleagues and I received an email requesting that we allow the extracted method to override

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

COUNTERPOINT

Trust Must Be Earned

Friedrich Steimann

Creating bug-free refactoring tools is a real challenge. However, tool developers will have to meet this challenge for their tools to be truly accepted.

WHEN I ASK people about the progress of their programming projects, I often get answers like "I got it to work—now I need to do some refactoring!" What they mean is that they managed to tweak their code so that it appears to do what it's supposed to do, but knowing the process, they realize all too well that its result won't pass even the lightest code review. In the following refactoring phase, whether it's manual or tool supported, minor or even larger behavior changes go unnoticed, are tolerated, or are even welcomed (because refactoring the code has revealed logical errors). I assume that this conception of refactoring is by far the most common, and I have no objections to it (other than, perhaps, that I would question such a software process per se).

Now imagine a scenario in which code has undergone extensive (and expensive) certification. If this code is touched in multiple locations, chances are that the entire certification must be repeated. Pervasive changes typically become necessary if the functional requirements change and the code's current design can't accommodate the new requirements in a form that would allow isolated certification of the changed code. If, however, we had refactoring tools that have been certified to preserve behavior, we might be able to refactor the code so that the necessary functional changes remain local and don't require global recertification of the software. Unfortunately, we don't have such tools.

There's also a third perspective—the one I care about most. As an engineer, and even more so as a researcher, I want to do things that are state-of-the-art. Where the state-of-the-art leaves something to be desired, I want to push it further. If that's impossible, I want to know why, and I want people to understand why so that they can adjust their expectations. Refactoring-tool users will more easily accept limitations if these limitations are inherent in the nature of the matter and aren't engineering shortcomings.

What we have today is the common sentiment that "if only the tool people had enough resources, they would fix the refactoring bugs," suggesting that no fundamental obstacles to fixing them exist. This of course has the corollary that the bugs aren't troubling enough to be fixed (because otherwise, the necessary resources would be made available). For this corollary, two explanations are common: "Hardly anyone uses refactoring tools anyway, so who cares about the bugs?" and "The bugs aren't a real problem; my compiler and test suite will catch them as I go." I reject both expla-

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

Point continued from page 80

an inherited method. Although the person requesting the change knew that this wouldn't preserve behavior, he also knew that the Extract Method transformation could be much quicker and more reliable with the tool than by hand. So, we promised to warn the developer about the issue but perform the transformation anyhow if the developer agreed.

Refactoring tools can also become more useful by relaxing the definition of "behavior." Under a strict definition, a program that executes more slowly than the original has changed its behavior. For some programs, such a change would be unacceptable. However, for most programs, executing a few milliseconds slower is acceptable. So, most refactoring tools omit execution time from the preserved behaviors.

Reflection is another area in which the behavior preservation requirement is usually relaxed. For example, if you employ reflection to use strings to find classes by name, any Rename Class refactoring will break your program. By allowing refactoring tools to ignore certain behaviors, we can build more useful tools. Consider replacing a set of radio buttons with a drop-down list. Such a change obviously isn't behavior preserving because the user will interact with the application differently. However, if we look at the behavior on the basis of what's saved to the database or what other widgets get enabled or disabled when users make a selection, it could be considered a refactoring.

Many people believe that the most important part of automated refactoring tools is correctness. They feel that without correctness, the tools won't be trusted, and without trust, they won't be used. However, I believe that helping developers work more efficiently is much more important than the dogma of behavior preservation. If such a tool can help developers, they'll use it, even though they can't trust that it will always be correct.

JOHN BRANT is an independent consultant and the coauthor of the Smalltalk Refactoring Browser. Contact him at brant@refactoryworkers.com.

Counterpoint continued from page 81

nations—the first because it denies refactoring the status of a relevant problem requiring tool support, the second because it implies a dependence on testing that refactoring-tool users might find unacceptable. Besides, coming up with excuses for ignoring bugs, rather than doing our best to fix them, won't increase trust in refactoring tools.

After working on refactoring tools for more than seven years, I've concluded that ridding them of their bugs is actually much harder than most tool users would believe. With respect to maintaining well-formedness (that is, the tool doesn't introduce compilation errors), static checking (as implemented by the compiler, which is basically a decision problem) must be extended to the much harder problem of computing the additional changes required to maintain a refactored program's well-formedness (basically a search problem). The idea of implementing a refactoring as a sequence of steps ("mechanics") grossly underrates the technical effort required to do this.

With respect to preserving behavior, the problems are even harder. Here, the boundaries are basically set by the precision of available static analyses. Surely, some programming languages are more amenable to such analyses than others, but I doubt whether programmers will ever adopt a programming language because of its "safe refactorability." So, we must accept that guarantees regarding behavior preservation can be given in only fairly limited cases (which might nevertheless be worthy of refactoring-tool support).

So, from my experience, in terms of reliability, current refactoring tools don't play in the same league as other programming tools, notably compilers, debuggers, or version control systems. This doesn't make them useless; having less-than-perfect refactoring tools is better than having no refactoring tools. Yet, to deserve users' trust, refactoring-tool builders can't be satisfied with the status quo but must continuously demonstrate a desire to build correct tools.

FRIEDRICH STEIMANN is full professor and chair of Programming Systems at Fernuniversität in Hagen. Contact him at steimann@acm.org.

STEIMANN RESPONDS

Software is soft because it's quickly changed. Refactoring tools make changing software even quicker. When I use a refactoring tool and the refactoring affects more than, say, a dozen distinct locations in the code, I usually look at the first couple of changes in the preview. I then find that I'll sacrifice much of the promised speedup if I try to manually check the correctness of all the scheduled changes. So, I accept all changes and wait for what the compiler has to say. If it says everything is okay, I usually don't worry about correctness and happily proceed with my work.

However, with this extra speed, which I certainly enjoy ("Wow, am I productive today!"), I tend to lose control of my code. If a bug pops up sometime later, I'm not sure who (me or the tool) introduced it when or where—the speed of development has overrun me. True, manual refactoring is slower and likely introduces bugs too; however, it leaves me more conscious of what I actually changed. Only if refactoring tools are correct is this consciousness never needed.

BRANT RESPONDS

First, I believe that refactoring tools and optimizing compilers do play in the same league. Both try to change code while preserving behavior. Optimizing compilers have a few more decades of research, so they're a little further along than some refactoring tools. However, they still have their issues. Many have command line switches that let users disable optimizations when they aren't working.

Second, although it's good to research what code analysis can and can't do, we can also do state-of-the-art research to determine what refactoring is needed most or create refactoring frameworks that support making new refactorings quickly.

Finally, I'm not against having refactoring tools that have been certified to preserve behavior. However, given that few compilers have such certification and that few projects get certified, I believe that time would be better spent researching other issues that affect more people.

FEATURE: REQUIREMENTS ENGINEERING

Coping with Quality Requirements in Large, Contract-Based Projects

Maya Daneva, University of Twente

Andrea Herrmann, Herrmann & Ehrlich

Luigi Buglione, Engineering Ingegneria Informatica

// Contracts for delivering large software systems must address issues such as system quality, timelines, delivery cost and effort, and service-level agreements. A study with 20 software architects revealed how they coped with quality requirements in this context. //

IN TODAY'S market, client organizations usually contract IT service vendors to deliver large software systems. In a contract to deliver a large software system, the client and vendor agree to undertake, or refrain from undertaking, certain actions in the course of delivering the system. The contact regulates the client–vendor relationship by defining each party's rights, liability, and expectations. It addresses a host of related issues such as system quality, timelines, delivery cost and effort, penalties for mistakes or missed deadlines, and service-level agreements (SLAs).[1–4]

How these issues are settled in a contract often creates various incentives for the participating parties.[4] Compared to in-house development, in contract-based projects the pressure to align incentives across parties is stronger, and the need to align the parties' understandings of system quality attributes is much greater. So, monitoring the project artifacts' quality is more complex.[4] Starting a conversation on quality requirements (QRs) as early as possible is therefore crucial. Software architects act as mediators between business analysts and clients on one side and developers on the other, and can constructively influence QRs engineering.

With this in mind, we studied architects' day-to-day coping strategies concerning QRs in large-scale, contract-based systems delivery projects. Here, we present some insights and surprising lessons that challenge the notion of software architects as "technical heroes" and redefine our understanding of QRs engineering from the architects' perspective (for more on this, see the sidebar). Most of our lessons signal that for software architects, moving from in-house to contract-based projects means embracing a different mentality, which might require some preparation to smooth the transition.

The Study

To determine a possible range of views on how architects cope with QRs, we ran a qualitative study in which we interviewed 20 software architects. These participants were from 14 companies in the Netherlands, Belgium, Finland, and Germany.

0740-7459/15/$31.00 © 2015 IEEE

RELATED WORK IN ARCHITECTS' PERCEPTIONS OF QUALITY REQUIREMENTS

Requirements engineering (RE) considers quality requirements (QRs) and the software architect's perspective important. However, only recently has RE research yielded the first five empirical studies of software architects' perceptions of QRs.[1–5] The consensus of these studies was that software architects' and RE specialists' perspectives on QRs differ. The software architects' experiences came mostly from small and mid-sized projects. Our research (see the main article) complements these studies with findings from large, contract-based projects.

As in those five studies, we found that software architects feel it's important to gain a deep understanding of the QRs and use this to deliver good architecture design. Our study revealed that all the software architects were actively involved in QRs refinement. This agrees with Eltjo Poort and his colleagues' findings[4,5] but contradicts those of David Ameller and his colleagues.[3]

However, regarding most aspects studied, we found significant differences due to

- the different architects' profiles,
- the project organizations' sizes,
- possible incentives in play, and
- the fact that large, contract-based projects are managed differently and take place in more regulated and standardized contexts than smaller projects.

In particular, we found the following differences from previously published results.

In Ameller and his colleagues' study, the software architects took on diverse technical roles and tasks (for example, coding). In contrast, our architects defined their role as a bridge connecting clients' QRs to the architecture design, with social interaction being key. Also, Ameller and his colleagues indicated a broad terminological gap between software architects and RE staff. We found no terminology-related issues because our case study's projects took place in regulated organizations in which standards defined terminologies explicitly. We also found that software architects mostly used standardized forms or templates plus natural language. This differed from Ameller and his colleagues' study, in which the software architects couldn't agree on a systematic way to document QRs.

Our results match Poort and his colleagues' findings regarding the close attention given to QRs quantification. However, our architects didn't indicate that they were preoccupied with searching for new or better quantification techniques. As in a previous study,[5] our architects emphasized the pitfalls of premature quantification. Such quantification might be based on too many assumptions about the solution, so a vendor might find itself in a precarious situation if those assumptions are unrealistic. Regarding quantification, our architects suggested either using a standard or engaging an expert in a specific type of QRs (for example, scalability).

We found no study investigating how contracts shape the way in which RE professionals or software architects handle QRs. So, we investigated this to some degree in our study, and we consider it an interesting line of future research.

References

1. R. Capilla et al., "Quality Requirements Engineering for Systems and Software Architecting: Methods, Approaches, and Tools," *Requirements Eng. J.*, vol. 17, no. 4, 2012, pp. 255–258.
2. U. van Heesch and P. Avgeriou, "Mature Architecting—a Survey about the Reasoning Process of Professional Architects," *Proc. 9th Working IEEE/IFIP Conf. Software Architecture*, 2011, pp. 260–269.
3. D. Ameller et al., "How Do Software Architects Consider Non-functional Requirements: An Exploratory Study," *Proc. 20th IEEE Requirements Eng. Conf.*, 2012, pp. 41–50.
4. E.R. Poort et al., "How Architects See Non-functional Requirements: Beware of Modifiability," *Requirements Engineering: Foundation for Software Quality*, LNCS 7195, Springer, 2012, pp. 37–51.
5. E.R. Poort et al., "Issues Dealing with Non-functional Requirements across the Contractual Divide," *Proc. 2012 Joint Working IEEE/IFIP Conf. Software Architecture and European Conf. Software Architecture*, 2012, pp. 315–319.

Each participant served in projects in which parties established contracts in two steps.[3] First, the parties agreed on a contract to get the requirements documented in sufficient detail such that an architect could use them to work on the architecture design. Then, a second contract addressed the system delivery itself. The pricing agreements varied across the companies: fixed-price, variable, or a combination of the two.

Also, each architect worked in large, contract-based projects running in at least three development locations in one country, and had clients in more than two countries. Finally, each participant had at least 10 years' experience in large systems.

Table 1 lists the projects' details. The contracts' total number of pages (see the rightmost column) included high-level business requirements, a statement of work, key performance

TABLE 1

Study participant details.*

Participant ID	Years of experience	Business	System description	No. of people on team	Project duration (mo.)	Pricing arrangement	Total no. of pages
P1	13	Large IT vendor	ERP package implementation (Oracle)	35	18	FP + V	350
P2	10.5	Large IT vendor	ERP package implementation (SAP)	60	15	FP + V	300
P3	15	Large IT vendor	ERP package implementation (SAP)	75	18	V	250
P4	18	Large IT vendor	ERP package implementation (SAP)	41	12	V	300
P5	10	Large IT vendor	ERP package implementation (SAP)	51	12	V	350
P6	12	Large IT vendor	ERP package implementation (Oracle)	45	12	V	350
P7	13	IT vendor	ERP package implementation (SAP)	40	18	V	330
P8	11	Software producer	Online learning environment	22	12	FP	180
P9	12	Software producer	Sensor system for in-building navigation	35	12	FP	220
P10	14	Software producer	Online ticket-booking application	15	12	FP + V	300
P11	13	Oil and gas	Logistics-planning application	21	12	FP + V	350
P12	10	Insurance	Web application for client self-service	61	24	FP	350
P13	11	Insurance	Client claim management and reimbursement application	53	16	FP	350
P14	12	Real estate	Web application for rental-contract handling	42	18	FP	300
P15	11	Air carrier	Web application for processing passenger feedback	11	14	V	350
P16	13	Video streaming	Viewer recommendation management system	18	18	FP + V	200
P17	10	Video streaming	Viewer complaint management system	45	9	FP + V	250
P18	16	Online bookstore	Order-processing system	15	10	FP + V	300
P19	14	Online-game producer	Gaming system	81	21	FP + V	220
P20	13	Online travel agency	Room deal identification system	45	12	V	300

** ERP = enterprise resource planning; F = fixed-price and V = variable (time and material contract).*

indicators (KPIs), SLAs, agreements regarding compliance with applicable laws, and the post-services schedule.

We asked each participant to think of his or her most recent project and provide insights into three areas concerning software architects' coping strategies for QRs:

- interaction with stakeholders (for example, how they negotiated QRs with stakeholders),
- engineering (how they documented QRs), and
- contract compliance (what role the agreements played in how they handled QRs).

The technical aspects of our research process appear elsewhere.[5]

Here, we distill takeaway messages for software professionals transitioning to contract-based system delivery models or searching for practices to adopt or to adapt for their own contract-based QRs engineering. Where applicable, we illustrate the messages with quotes from the participants.

Assume a Mediator's Role

Thirteen participants thought of their role as a bridge between clients' QRs and the underlying technology. As participant P10 put it,

> *You are there to connect people, to bridge gaps between business "patrons" and developers, to make sure technology catches up with their business demands and that they are aware of the possible choices.*

The other seven participants regarded their role as a review gatekeeper because they served most of their time reviewing QRs, giving feedback, and evaluating contract compliance.

Whom did architects speak with most of the time concerning QRs? Seventeen participants went to business analysts (requirements-engineering staff) for clarification on QRs. Two considered the vendor relationship manager responsible for their projects. That was because the vendor relationship manager had the authority to make decisions and negotiate with external parties (vendors) about expected quality levels, and ensured that project execution occurred as per the contract. One participant identified her project manager as the most frequently contacted person.

No participant dealt directly with the user community on an ongoing basis. However, the key business users knew the participants personally because the participants served as mediators in translating technology choices in business terms to non-technical project stakeholders.

Perhaps our most surprising finding is that, although the participants interacted with a diverse audience, they didn't struggle with communication breakdowns due to various QRs interpretations. They attributed this observation to the domain knowledge they had accumulated in their many years of experience in their respective business sectors. For example, when working with less-experienced business analysts, they found their domain knowledge instrumental to spot missing or incomplete requirements. P15, who worked on an online system for processing feedback from an air carrier's clients, said that in this application, scalability was usually regarded as the highest-priority QR. If a specification mentioned nothing about it, he would red-flag that specification and follow up with business analysts and collect details on it.[5]

Use Standards to Ease Communication

The participants who worked in ISO-certified organizations suggested that, next to domain knowledge, knowledge of the adopted ISO standards and mandatory ISO training for everyone in their organization helped them and their business analysts understand each other. As P11 said,

> *We all are in the habit of looking back to what the ISO-compliant Quality Manual says.*

We found that the companies invoked two streams of ISO standards to keep a common interpretation of QRs terminology:

- management systems (for example, ISO standard 9001-27001-20000-1) that are about requirements to be met and
- technical standards (such as 14143-x, 9126-x, and so on) that describe processes and how to do things.

A common concern of the participants, though, was that both streams used QRs terms interchangeably, and it wasn't clear which QR followed the terminology of which stream. So, the participants often coped by making the tacit knowledge explicit by establishing cross-references between the terms traceable to the various standards.

Discover QRs through Refinement

The study participants felt that QRs elicitation included refining any QRs deemed important for the project and mentioned explicitly in the contract. They reported that they took the lead in this process. Fourteen participants working in regulated business domains (such as insurance) championed the

use of checklists to refine QRs. As a basis for the checklists, eight used ISO standards in the SQuaRE (System and Software Quality Requirements and Evaluation) series—for example, 25045:2010, 25010:2011, 25041:2012, and 25060:2010. Four participants used architecture frameworks that were specific to their company, the business sector, or the client organization. Two used stakeholder engagement standards such as AA1000SES. Ten of them created the checklists themselves.

Six participants employed more creative techniques (for example, in video streaming and game design). For example, some used storytelling techniques to gather knowledge in the form of stories. Others employed serious-game-based techniques (used in logistics information systems projects), in which users played a scheduling game by using an early prototype to detect the system's performance and availability requirements.

Use Predefined QRs Templates

Most participants preferred to use predefined templates for QRs documentation. Fifteen participants used templates based on

- the ISO standard,
- vendor-specific standards (for example, in SAP and Oracle projects), or
- Quality Function Deployment (QFD), which translates user-recognizable quality attributes into features for implementation in the system.[6]

The other five participants used plain text to define QRs, plus information on the end user to perform acceptance tests and demonstrate that the system met the QRs.

We assume these results are due to the regulated nature of contract-based environments and the use of standards. So, the continual monitoring of contracts (with well-specified SLAs and explicitly defined measures and metrics) forces IT professionals to adopt a sound template-based documentation flow throughout the project.[2]

Regardless of the document format, the participants agreed on the importance of ensuring that QRs get reviewed and updated after each milestone in the contract. As P11 stated,

> *You must update [the QRs] and ensure your commitments to quality levels are still realistic. Otherwise, you open a door for big issues with your clients.*

A Business Case Helps to Prioritize QRs

All the participants felt that their project's business case was the key driver in prioritizing QRs. This finding is in line with Frank Buschmann and his colleagues' business value perspective on system quality.[7] The participants felt responsible for making other stakeholders aware of QRs' tradeoffs. So, they perceived QRs prioritization as an iterative learning experience in which the architects first learn about the QRs needed for the business, along with how urgently the company needs them. Then, the business analysts and stakeholders learn about the technical and cost limitations.

We found four prioritization criteria for making QRs tradeoffs, on the basis of the business case:

- *Cost and benefits* might be estimated quantitatively (for example, the person-months spent to implement specific QRs).
- *Perceived risk* is subsumed in the cost category; that is, risks to QRs are translated into costs.
- *Affordability* determines whether a QR's estimated cost is in accord with the resources specified in the contract and with the client's long-term contract spending.
- *Willingness to pay* is the client's readiness to pay extra for a perceived benefit of implementing or increasing a specific QR.

Although cost, benefits, and risk are well known,[3] no studies we know of have covered affordability and willingness to pay. We assume that the choice of these criteria is due to the contract-based nature of the participants' projects. In those projects, the parties had to gain a clear understanding as early as possible of the scope, the project duration, how they organized their work processes, and the penalties for deviation.

Who ultimately decides on QRs priorities? Thirteen participants named the project's steering committee because it linked prioritization to a business driver or a project's KPIs. Seven participants considered themselves the key decision makers on QRs priorities. For example, P13 said,

> *I must be resolute and very clear about what we can call "must-haves."*

P14 said,

> *I make the decision on the premise that I have a good justification for each choice.*

Nineteen participants suggested no explicit use of any method for QRs prioritization other than classifying QRs as essential, marginal, or optional. For three of the 19 (P8,

P19, and P20), naming the top two or three most important QRs wasn't an issue because those QRs were obvious to the client. For example, in the computer game sector, it's a given that the most important QRs are scalability and the user experience.

However, the participants deemed difficult the prioritization of those QRs that were less prominent (from the user's perspective), yet important from the vendor's perspective. For example, regarding maintainability, P13 said,

> *These requirements matter to you, not to the client. Even [if] you do a brilliant job on maintainability, nobody will pat you on the back and say thank you for this. It's very difficult, I'd say, almost political, to prioritize this kind of QR.*

If Quantifying QRs, Start with the Contract

Contract-based projects seem to prompt project organizations to express their QRs quantitatively. All the participants agreed on this, but there was no common approach to quantifying QRs.

Our data suggests that architects either use a size-estimation standard (such as the International Function Point Users Group's nonfunctional-assessment method[8]) or engage an expert in quantifying a specific type of QR (security or scalability, for example). The first approach ensures that a project explicitly accounts for all QRs implementation tasks; the second allows for deeper analysis of a single quality attribute and its interplay with others.[9] The participants deemed both approaches important to prevent early and poorly conceived commitments. They indicated that if a contract explicitly mentioned a size estimation standard or an independent expert's evaluation, then it was considered legally binding and they had to apply it.

A common starting point was the contract's prespecified quantitative definitions. For example, a contract might state explicitly that the system should scale up to serve hundreds of thousands of subscribers. However, eight participants experienced that contracts often confused QRs with design-level requirements.[3] For instance, instead of QRs, one contract contained detailed feature specifications of how to achieve QRs. This example involved specifying a proprietary influence metric to include in the ranking algorithms of a recommender system that's part of an online video-streaming system. In another case, the contract specified a particular algorithm rather than a required quality attribute value and criteria for verifying compliance. This example involved coding a specific search algorithm for finding hotel deals. The participants considered this issue critical because it signalled a misaligned understanding of what was really quantified and what measures were used.

Use Walk-Throughs to Validate QRs

For our participants, validation ensures that QRs are aligned with the client's expectations and the SLAs. It also confirms that the QRs are technically implementable and that the resulting architecture design satisfies the contractually defined business requirements. Each participant was heavily involved in this process. Sixteen participants considered validation their own responsibility; four deemed it the business analysts' job. We assume that the participants' active and persuasive behavior regarding QRs validation could have been due to the explicit contractual agreements (for example, SLAs), controls, and project-monitoring procedures (KPIs).

No participant witnessed a contract requiring the use of an automated (model checking) tool to validate QRs. Instead, common-sense, down-to-earth practices prevailed. The participants felt that requirements walkthroughs, documentation reviews, and building up communication processes around artifact development (for example, escalating if a QR isn't clarified in a timely fashion) were simple, yet powerful ways to validate QRs.

Fourteen participants regularly performed requirements walkthroughs with clients and business analysts, in which clients confirmed the functionalities to which the QRs applied. The participants deemed such walkthroughs as part of the client expectation management process that the project manager established.

Other participants used internal architecture standards and QFD to demonstrate the related strength between a QR definition and its operationalization, in terms of features or architecture design choices. If deviations crystalized, the participants took the problem to project managers or a steering committee, depending on how large the misalignments were.

QRs Conflict Resolution Should Be Objective

In the participants' experience, contract-based system delivery included explicit QRs conflict-resolution procedures and using information that was as objective as possible. For most participants, the business case was the most important vehicle for supporting their negotiation positions when resolving conflicting QRs. As P20 noted,

You need to express yourself in money terms that they [the clients] can understand very well.

Ten participants routinely used the business case to justify tradeoffs; three others used effort and budget allocation figures derived from the business case. Other participants used QFD, EasyWinWin (a collaborative process to achieve a win–win situation among parties), or the Six Thinking Hats method[10] to reason about QRs in negotiation meetings. Six Thinking Hats is a general approach to resolving complex issues that companies can use for any negotiation situation.

Your Contract Is a Resource

We were surprised that the participants actively used the contract to achieve different, yet complementary, goals. Every participant considered it an important source of learning about QRs. As P5 explained,

You learn not only about the quality levels in the resulting system but also about the culture of the client. The greater the knowledge of our clients in systems development, the more detailed the terms they use to specify the contract.

Seventeen participants used the contract continually to stay focused on what counted most in the project. They mentioned that the contract helped ensure that the system included "the right things" (P11) and "those that they needed in the first place, and that we are billing them for" (P13).

Twelve participants perceived the contract as a vehicle to maintain control. They believed this was because every comprehensive contract usually came with SLAs, KPIs, and measurement plans that addressed multiple perspectives (clients or vendors). For example, the Balancing Multiple Perspectives technique[9] can help the involved parties understand and confirm all the things to do in a contract-based project.

How Contracts Affect Architects

Our data revealed three ways a contract shapes the architects' coping strategies regarding QRs:

- The contract aroused the participants' cost consciousness, which led them to habitually estimate QRs' cost.
- The contract stipulated QRs levels (for example, in the SLA) that the participants discussed with the stakeholders.
- The contract predefined the priorities for a small but important set of QRs.

However, three participants disagreed with that third item in the list. They considered the contract just the beginning of the conversation on QRs, not a reference guide to consult routinely. (They felt it was the people who made the contract work.)

Did any contract-caused incentives motivate the participants to approach QRs the way they did? We found that specific elaborations on price, rewards, and penalties made project organizations default to some choices while setting up their delivery process.

For example, fixed-price contracts created an incentive for vendors to staff projects with the most qualified resources. So, they perceived that engaging experienced architects to engineer QRs end-to-end in a project was conducive to executing the QRs activities in a disciplined and systematic fashion. Vendors' managers also found it discouraging to redirect staff to different capacities mid-project.

In addition, fixed-price contracts made the participants conscious about the possibility of facing disputes due to poorly done QRs. This was because the original project scope often didn't include deliverables that were only implicitly assumed in the project. To counter this, the participants at all times maintained complete awareness of what QRs were and weren't in scope. As P8 said,

Against this backdrop, I don't think you can afford … to ignore QRs.

Applying These Strategies

Because our study was exploratory and aimed to get a snapshot of possible strategies for coping with QRs, we can't claim that our findings are generalizable. Nevertheless, we think that such strategies might be useful in situations in which managers in large projects hire experienced architects, teams use process-oriented thinking and are aware of what's in the contract (and how this shapes their process), and standards define the terminology to use for QRs factors.

Large, contract-based system delivery projects favor joint requirements engineering and architecture design. So, they often ensure that architects are integral to QRs processes. Our study participants' coping strategies reveal four clear payoffs to this strategy.

First, engaging the architects is instrumental in dealing with QRs with the same due diligence that the functional requirements and architecture design demand. Second, mitigating the risks of contract disputes becomes the architect's responsibil-

ity, who takes leadership in ensuring that QRs tradeoffs align with the contract. Third, leveraging the architect's domain knowledge helps proactively clarify QRs. Finally, socially positioning the architect as a bridge and gatekeeper is conducive to the ongoing conversation on QRs.

References

1. B. Berenbach et al., "Contract-Based Requirements Engineering," *Proc. 3rd Int'l Workshop Requirements Eng. and Law*, 2010, pp. 27–33.
2. X. Song et al., "Categorizing Requirements for a Contract-Based System Integration Project," *Proc. 2012 Requirements Eng. Conf.*, 2012, pp. 279–284.
3. S. Lauesen, *Software Requirements: Styles and Techniques*, Addison-Wesley, 2002.
4. D.J. Wu et al., "IT Implementation Contract Design: Analytical and Experimental Investigation of IT Value, Learning, and Contract Structure," *Information Systems Research*, vol. 24, no. 3, 2012, pp. 787–801.
5. M. Daneva et al., "Software Architects' Experiences of Quality Requirements: What We Know and What We Do Not Know?" *Requirements Engineering: Foundation for Software Quality*, LNCS 7830, Springer, 2013, pp. 1–17.
6. Y. Akao, *QFD: Quality Function Deployment—Integrating Customer Requirements into Product Design*, Productivity Press, 2004.
7. F. Buschmann et al, "Architecture Quality Revisited," *IEEE Software*, vol. 29, no. 4, 2012, pp. 22–24.
8. *Software Non-functional Assessment Process (SNAP) Assessment Practice Manual*, release 2.2, Int'l Function Point Users Group, June 2014.
9. L. Buglione and A. Abran, "Improving Measurement Plans from Multiple Dimensions: Exercising with Balancing Multiple Dimensions—BMP," *Proc. 1st Workshop Methods for Learning Metrics*, 2005, pp. 205–214.
10. E. de Bono, *Six Thinking Hats*, Little, Brown, & Co., 1985.

ABOUT THE AUTHORS

MAYA DANEVA is a senior scientific staff member of the University of Twente's Services, Cybersecurity and Safety Research Group. She's a specialist in requirements engineering, effort estimation of large systems, and empirical software engineering. Previously, she spent nine years as a business process analyst in the Architecture Group at Telus. Daneva received a PhD in computer science and software engineering from St. Clement Ochridsky University. Contact her at m.daneva@utwente.nl.

ANDREA HERRMANN is a software engineering trainer and researcher at Herrmann & Ehrlich. Her research interests include requirements engineering, project management, and software architecture. Herrmann received a habilitation in software engineering from the University of Heidelberg. Contact her at herrmann@herrmann-ehrlich.de.

LUIGI BUGLIONE is a process improvement and measurement specialist at Engineering Ingegneria Informatica and an associate professor of software measurement at École de Technologie Supérieure. His research interests are software measurement, process improvement, and service management. Buglione received a PhD in management information systems from LUISS Guido Carli University. Contact him at luigi.buglione@eng.it.

SOFTWARE TECHNOLOGY

Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Looking into the Future

Christof Ebert

No matter what business you're in, you're also in the software business. But where are we heading? This 50th installment of Software Technology tries to look into the future. To mitigate bias, I spoke with many software business leaders around the world. They pointed to five success factors that will advance the software business. They left unaddressed whether software will evolve humankind to Humanity 2.0—or a posthuman society. Read more in this column and let me know your opinion and own stimulus. I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —*Christof Ebert.*

SOFTWARE MAKES the world go round—at ever-increasing speeds. From the embedded software in cars, avionics, and automation to the ubiquitous computers in smart phones and consumer electronics, software provides features and functions in daily use. Our society depends on reliable software systems, which depend on innovative companies, which depend on competent software engineers.

In preparing this column, I spoke to business leaders and industry experts to get an impression of what they see as the future of software. Vector Consulting Services' annual industry benchmark performed with different companies from various industries helped provide a sound quantitative baseline.[1] All the companies are in the software business, either directly with IT solutions or indirectly with products in which software is critical for value creation, such as embedded systems, or value delivery, such as services. One-third of the companies have more than 100,000 employees worldwide, half have 10,000 to 100,000 employees, and the rest are small and medium-sized companies with approximately 1,000 employees. The companies include ABB, Bosch, GE, Google, Lufthansa, Philips, and Siemens, among others.

> We are confronted with the two faces of our industry—innovation and complexity.

The State of the Software Business

The software business is different from traditional industries.[2,3] Industry experts and leaders both agree that we are heavily confronted with the two faces of our industry—innovation and complexity:

- Software attracts more investment than other industries, which is making its market valuation grow faster than any other industry, thus continuously accelerating innovation.
- Creativity matters, whereas production and logistics are of limited relevance and incur marginal costs, which makes it easy to enter a market and sell globally.
- Software is flexible and easy to change, which results in a high frequency of products and

0740-7459/15/$31.00 © 2015 IEEE

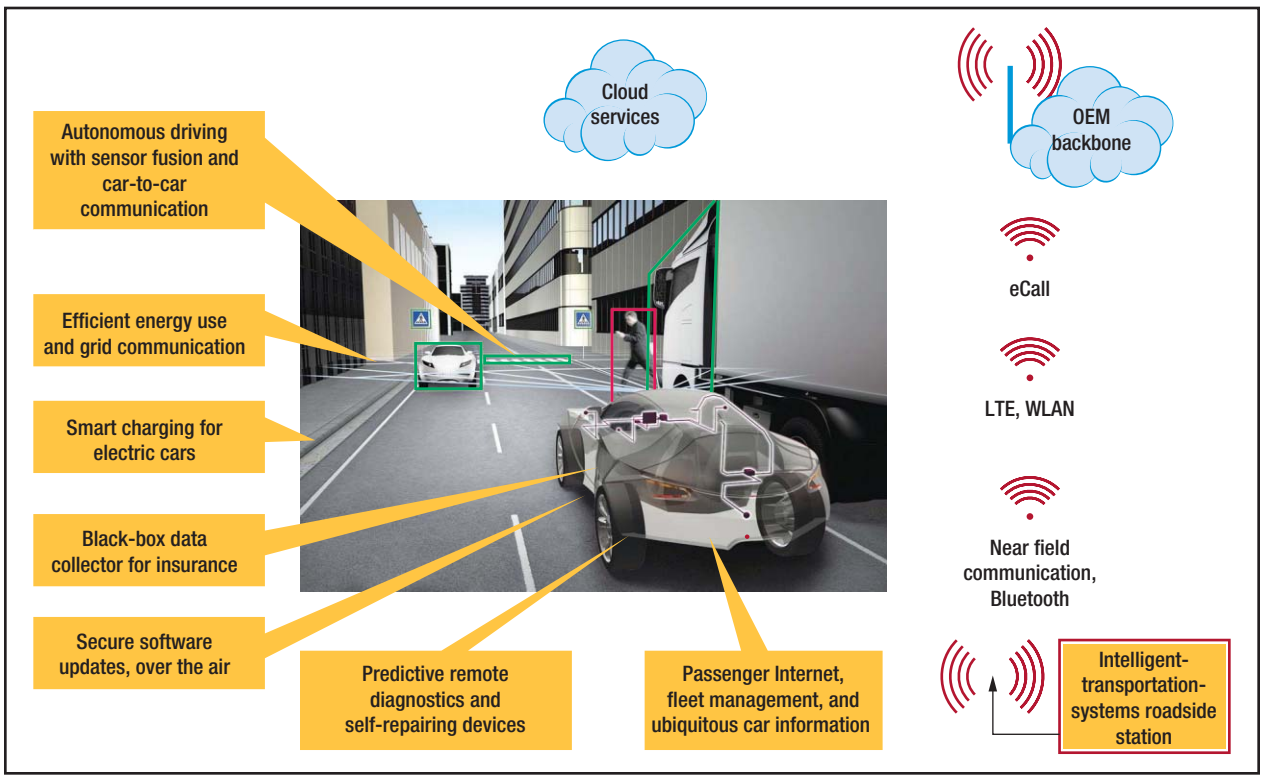


FIGURE 1. Software innovation fuels automotive advances, from smart energy efficiency to autonomous driving.

releases that don't always create real value.
- The overwhelming complexity of technology and products, combined with insufficient competences, severely cannibalizes the quality of software-driven products.

One impression clearly dominates. Software is changing practically all industries and is the major driver of innovation across all industries. While we used to distinguish components, systems, and services, we today see flexible boundaries driven entirely by business cases to determine what we should package, at which level, and in which component, whether it's software or silicon.

Take consumer and communication systems. A TV in the 1970s had no software, whereas today its competitive advantages are software-driven. Or consider aerospace systems. In 1960, only 8 percent of the F-4 fighter's functionality was implemented in software; by 2000, software provided 80 percent of the F-22's features.[2,3] Today we see an increasing number of military aircraft operating as autonomous drones.

The automotive industry is following the same pattern. Modern cars contain more than 100 embedded computers, with more software than any airplane. Cars sell primarily because of their software-based functions, whether those functions involve the power train, energy efficiency, an electric-car platform, or communication facilities for driver assistance, safety, diagnosis, and so on. Autonomous driving is being introduced, thus moving the industry to a primarily software business. Not surprisingly, the North American International Auto Show recently featured the slogan that a car is a microprocessor with wheels and an engine (see Figure 1).

The Future of Software

Five dimensions characterize the future of software:[3]

- *collaboration*—for example, consumer Internet, social networks, single-customer segmentation, product and service configurators, digital money, computer-assisted collaboration, and crowdsourcing;
- *comprehension*—for example, semantic search, big data handling, smart data, data analytics, data economy, online data validation, and data quality;
- *connectivity*—for example,

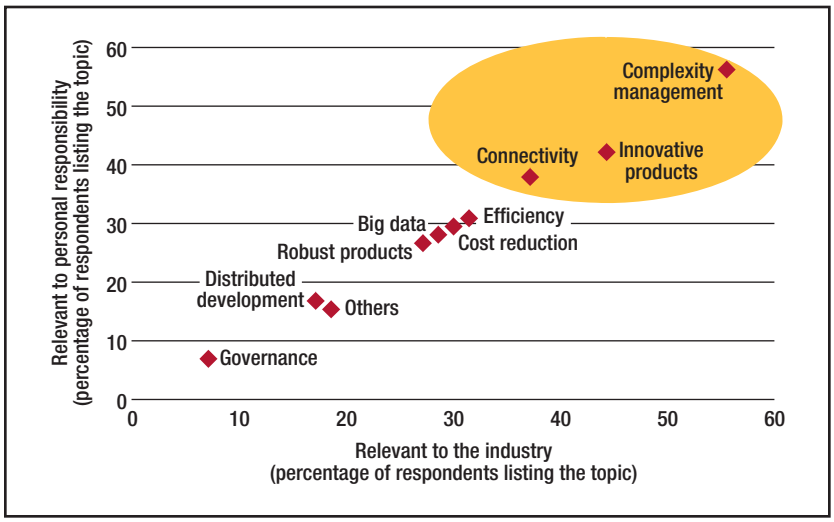


FIGURE 2. Industry trends.[1] Innovation, complexity, and connectivity are the major issues for business and technology leaders. The sums are greater than 100 percent because the survey allowed three answers per question.

ubiquitous mobile computing, mobile services, cyber-physical systems, Industry 4.0, machine-to-machine communication, sensor networks, and multisensor fusion;
- *cloud*—for example, applications and services in the cloud, location-based networks, new license models for software and applications, sustainability, and energy efficiency; and
- *convergence*—for example, mobile enterprises, bioinformatics, the Internet of Things, pervasive sensing, and autonomous systems.

These dimensions, coupled with the underlying complexity and scale, demand new software solutions based on new computing paradigms and infrastructure. Examples include IT architectures that facilitate seamless connectivity, robust infrastructures for cyber-physical systems in safety-critical environments, and data analytics to predict choices and behaviors to improve the customer experience. Such software-driven solutions can create nontraditional market entry points and consequently new mechanisms to provide each customer time-specific and location-specific services.

What are the dominating expectations and needs from a software business perspective? From our benchmarks with companies, we found they'll continue to invest in growth through innovation by developing new products and solutions because this determines their market position. At the same time, they are aware of the volatile market and are thus trimming their development teams worldwide to be as lean and as efficient as possible.

Figure 2 provides a recent aggregation of industry trends from our annual business survey.[1] In that anonymous survey, we asked managers from different companies, industries, and hierarchy levels what will be relevant in the near future for their own industry and for their own responsibility. They could select from different topics and add their own topics.

Most industry leaders push for more innovation—balanced with a clear need for not only more efficiency and cost reduction but also product robustness. Obviously, the complexity caused by continuous innovation, specifically in the software market, must be balanced with more rigid instruments to keep business processes lean and products robust. The evolution from our preceding surveys is interesting. In previous years, software companies focused on cost reduction and efficiency. Now, with full order books and a positive outlook, they're focusing on managing homegrown complexity.

Software Risks

Success in the software business isn't a given. Products and solutions must meet increasing quality requirements but must be designed to be inexpensive and easily adaptable and to exploit modern platforms' advantages. New competitors are entering markets with new solutions, without being hampered by legacy systems and traditional business models. The technology landscape has become increasingly complex.

In general, the interviewees revealed four key obstacles to success:

- a dysfunctional organization with unclear responsibilities and silo work, which results in continuously changing focus and schedules;
- lack of strategy or an unclear strategy and roadmaps with unclear dependencies and fuzzy technical requirements and impacts;
- no standardized business processes across the company, with

slow, cumbersome decision making and many individual ad hoc agreements; and
- insufficient requirements that are often just collections of what was heard at customer visits and other such events and that haven't been mapped to value creation and business cases.

Innovation, globalization, and complexity are fueling the software business worldwide but also creating many risks that can easily endanger or kill a business. So, the software business has many challenges, ranging from the creation process and its inherent risks to direct balance sheet impacts.

Software is getting more complex, more connected, and more life-critical. This complexity's sources are hidden in the nature of software, which often consists of many components from different vendors and runs on hardware manufactured by different vendors. Also, software teams frequently are multifunctional, and team members are responsible for many activities such as planning, developing, and executing plans, roadmaps, and strategies—without adequate training. This differs from other industries in which, for instance, production and logistics are strictly separated from development. Even within development teams, the people who design a product or component are often those who test it. No wonder that quality is often below expectations and that specifications are inconsistently implemented.

Stimulus for the Leading Practitioner

How do software companies find appropriate starting points to manage complexity? How do they bring innovations to the market faster and improve connectivity? Which techniques have been proven in practice by successful companies?

Here are some concrete experiences and advice from Vector Consulting Group's projects around the world. Looking at the articulated challenges on one hand and the successful companies on the other hand, we found five success factors that will help companies master future challenges. These factors aren't comprehensive but can help companies focus on what really matters.

Focus Where the Money Is

Products are pushed to the extreme to be ever more efficient and low cost, but they don't always sell as expected. Customers demand many changes, thus reducing margins dramatically from the initial targets. Many managers with whom we talked tend to concentrate on what they know best, which too often in the software industry are technical topics, rather than basic business principles. Targets on each level must be set up consistently and monitored continuously against actual performance. Changes in the business climate must be managed, or they ripple through uncontrolled.

On the senior-management level, we recommend a balanced-scorecard approach, whereas on the operational level, value-driven steering, such as earned value management, should be used. Each company and software team can introduce these two techniques to focus on the right things and balance short-term survival needs with medium- to long-term strategy evolution, investments, and improvements.

Sell Value Rather Than Features

When we work with clients on product strategy and requirements engineering, we first ask, what market, what user, and what usage? The overwhelming number of responses can be reduced to, "The spec says so." Now this might be formally true, but it will lead to a downward spiral of complexity and cost.

Although complexity sells, it must be balanced with other factors. Complexity creates much extra cost for service, evolution, variance management, and regressions along the life cycle. We often face clients who don't really know how to effectively control complexity. They trap themselves with slogans such as "We're too expensive," rather than nailing it down to concrete levers they can directly address. Around half of all requirements don't create marginal value but contribute to cost.[2] Variants and specific features for individual customers further increase complexity. Steve Jobs was one of the few taking concrete lessons from this principle, demanding simplicity in his products.

> Industry leaders push for more innovation—balanced with efficiency and product robustness.

Our Vector RACE (Reduce Accidents, Control Essence) method provides the tools to effectively manage complexity. Reducing accidents means cutting overheads such as gold-plating and rework due to late defect removal or too many requirements changes. Controlling essence enforces looking to what customers really pay for. Each requirement must be justified to support the business case and to allow managing changes and priorities. Ask a tester to write a test case before processing the requirement. Ask the team's marketer to check whether he or she can sell the feature as described. Determine what's good enough, and ensure that any further insight is adequately considered. With all that focus on value, ensure that agile and lean don't lead to arbitrariness.

> It's up to today's software engineers how to embed software in humans and how to embed humans in software.

Manage Relevant Stakeholders

Stakeholders matter in both small and big companies. Often they follow their own agendas, optimize locally in their silos, and don't actively own and drive the company and its products.

So, we strongly recommend creating a core team with the product, marketing, project, and operations managers for each product (release) and making it fully accountable for the product's success. These persons not only represent the major internal stakeholders in product or solution development but also sufficiently represent different external perspectives. The success factor is to give this core team a clear mandate to own the project.

Apply adequate risk management techniques to make your portfolio and commitments dependable. Projects might need more resources, suppliers could deliver late, or technology won't work as expected. Replace traditional labor-cost-based location decisions with systematic improvement of business processes in a distributed context. The benefits will be tangible, as our clients emphasize: multisite collaboration, clean variant management, and transparent workflows are the most common reported benefits. Replace looking merely at labor cost with a holistic strategy taking into account onsite presence, customer ecosystems, and reduction of friction losses.

Master the Life Cycle

Software products have the same liability requirements as any other product, but their suppliers often fail to establish the necessary life-cycle processes. All product releases and projects should follow a standardized product life cycle. Most companies have defined such a life cycle but rarely use it as the pivotal tool to derive and implement decisions. Requirements changes are often agreed on in sales meetings without anyone checking feasibility, and technical decisions are made without considering business case and downstream impacts.

Automated product life-cycle management (PLM) and application life-cycle management (ALM) are the primary mechanisms for efficiently integrating engineering processes, tools, and people across the domains of system, software, hardware, and mechanical engineering. Process shouldn't mean a burden and documentation, but efficiency in repetitive activities. Life-cycle management ensures governance and thus provides the safety net that almost all businesses need, not only vis-à-vis their owners and investors but also in the market for legal reasons.

Continuously Improve

Nearly 90 percent of our interviewees claimed that their companies want to improve performance, but only a third of them were satisfied with the results. The same held for improving their own competences and behaviors. Insufficient change management is a common reason for failing.

Cost and cycle times can be reduced when you're working with continually optimized development processes. Lean development and agile approaches are useful for streamlining interfaces and reducing rework and inefficiencies. Earned value management, value stream mapping, and Scrum are proven techniques that can easily be tailored to organizations. Streamline workflows and related tools stepwise, with an overarching strategy, incremental goals, and a future-oriented IT architecture. Set concrete improvement targets on a quarterly basis.

Train employees in lean development. Have each team develop an action plan for reducing waste, rework, and interface conflicts—with

reference to your company-wide efficiency targets. Train people to grow technical and—even more relevant—soft skills. Evaluate performance—for example, by sales per developer, lead time, fault detection rates, cost drivers, competences, and innovations. Apply professional change management.

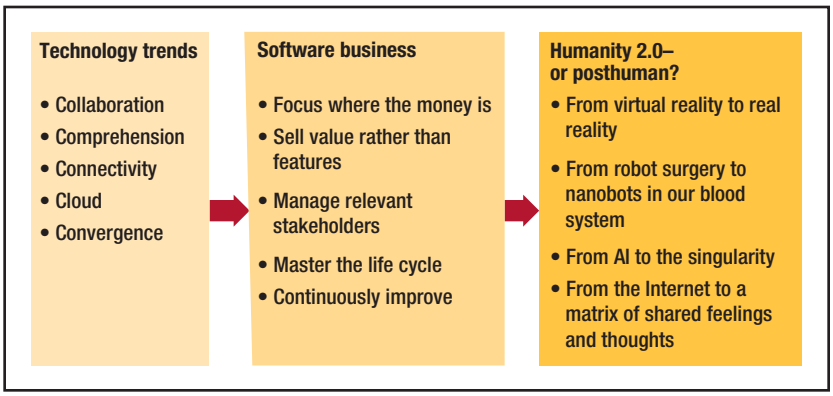


FIGURE 3. The way ahead—Humanity 2.0 or a posthuman society?

Our business climate will remain volatile. At the same time, complexity and technology will quickly grow. The resulting competence gap will lead to an even stronger fight for skills. As we mentioned before, companies will continue to invest in growth through innovation and will require development teams to be lean and innovative. Let's look briefly at what the next 30 years might hold.

Virtual environments and augmented reality have set the path toward software directly creating our own reality. Many kids today can't imagine life without software-driven games and embedded AI. Many people see games and social networks with many so-called friends as more realistic and more appealing than our classic Humanity release 1.0. Direct interaction with our nervous system will advance such virtual reality toward real reality, or Humanity 2.0. The impact on society will be manifold.

For example, consider cars, transportation, and their negative impact on the environment. If we can meet people at any place without actually travelling, pollution and accidents will be very much reduced. With blood-cell-sized devices in our bodies fighting against diseases and improving our intellectual and cognitive abilities, life will fundamentally change.[4] Software will extend the brain the same way it has extended our behaviors today. Our computers will successfully pass the Turing test and soon after achieve the singularity.[5]

What used to be civilization in the classic sense of humanitarianism will become increasingly dominated by nonbiological and posthuman intelligence. The Internet will evolve to a global network connecting billions of people and machines to share data, information, and eventually thoughts and feelings. All this will dramatically affect how we live, behave, and evolve not just as individual humans but also as societies and as humanity. Software will be at the center of this change, so we'd better shape today the necessary competences to do it the right way, which is more about values and soft skills than technology.

Humanity 2.0 will be fueled by the software technology we build today, as we innovate processes and products, while their development, production, operation, and service are increasingly done by machines. In doing so, we'll advance humanity—hopefully for the better. Figure 3 illustrates these major trends but leaves one key question unanswered. Will it be a Humanity 2.0 with better quality of life, or will it be a posthuman environment? It's up to today's software engineers how to embed software in humans or how to embed humans in software, while preserving a society of humans who act human.

References

1. "Industry Trends 2015," Vector Consulting Services, 2015; www.vector.com/trends.
2. C. Ebert and S. Brinkkemper, "Software Product Management—an Industry Evaluation," *J. Systems and Software*, vol. 95, 2014, pp. 10–18.
3. C. Ebert, G. Hoefner, and V.S. Mani, "What Next? Advances in Software-Driven Industries," *IEEE Software*, vol. 32, no. 1, Jan. 2015, pp. 22–28.
4. E. Brynjolfsson and A. McAfee, *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*, Norton, 2014.
5. R. Kurzweil, *How to Create a Mind: The Secret of Human Thought Revealed*, Penguin, 2013.

CHRISTOF EBERT is the managing director of Vector Consulting Services. He is on the *IEEE Software* editorial board and teaches at the Universities of Stuttgart and Paris. Contact him at christof.ebert@vector.com.

PRACTITIONERS' DIGEST

Editor: **Jeffrey C. Carver**
University of Alabama
carver@cs.ua.edu

Software Quality, Energy Awareness, and More

Jeffrey C. Carver, Aiko Yamashita, Leandro Minku, Mayy Habayeb, and Sedef Akinli Kocak

FOLLOWING ON last issue's column, this column reports on six more papers from the 2015 International Conference on Software Engineering (ICSE) and its satellite events.

"Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality," by Mathieu Lavallée and Pierre Robillard, identified 10 organizational factors that can decrease software quality. To identify the factors, Lavallée and Robillard observed 10 months of weekly status meetings about an in-house software project at a large telecommunications company. Representative factors include these:

- *Internal dependencies*. Many dependencies exist between software modules, and conflicts on scheduling deployments exist between projects.
- *External dependencies*. Long-term dependencies on third-party libraries exist, and change requests to those libraries cause delays.
- *Organically grown processes*. Processes emerge as needed, usually after a crisis, and are often introduced locally rather than organization-wide.
- *Budget protection*. Developers feel it's cheaper in the short term to build a wrapper than to solve a problem.
- *Scope protection*. Rather than prioritizing a global scope, teams prioritize a local scope and transfer as many requirements as possible to other projects.
- *Undue pressure*. Managers and senior developers circumvent team policies to give direct orders to the team and threaten it.

> Better training and process management can avoid operational faults and misuse.

For each factor, the authors suggested corrective actions.

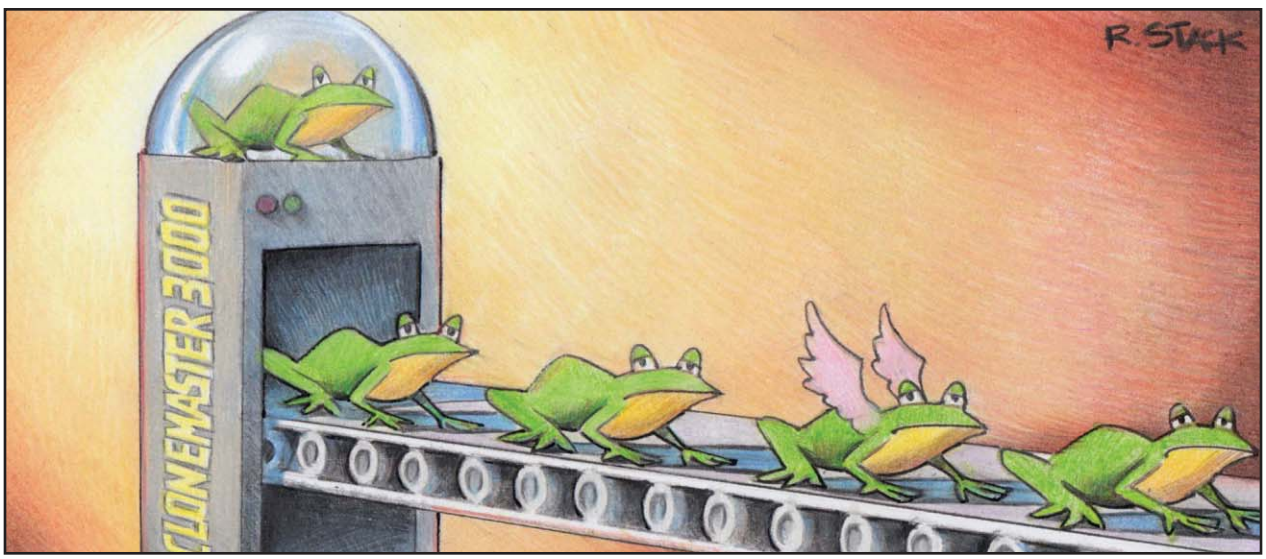
Although the authors' findings indicate that these problems might not affect project success, they do affect software quality, which in turn increases software maintenance costs over time. Other key takeaways include these:

- The lack of centralized strategies for making key decisions might be reflected through conflicting software modules.
- Practitioners can use these organizational antipatterns as hints to warn stakeholders, customers, managers, and team members when their actions might result in code that's costlier to maintain.

This paper appeared in the main research track of ICSE '15; access it at http://goo.gl/yrCVHh.

"The Last Line Effect," by Moritz Beller and his colleagues, discussed the phenomenon in which the last line or statement of a micro-clone (a very short segment of duplicated code) is much more likely to be faulty than any other line or statement. (The last-line effect has caught the reddit community's attention; see www.reddit.com/r/programming/comments/270orx/the.)

0740-7459/15/$31.00 © 2015 IEEE



To detect such microclones, which traditional clone detection tools miss, Beller and his colleagues used the PVS-Studio static-analysis tool (www.viva64.com/en/pvs-studio). They analyzed 202 microclones from 208 well-known open source systems. They found that, when a fault was present, the last statement was 17 times more likely to be faulty than all other statements combined. For faulty microclones consisting of only two statements, the second statement was always the faulty one.

These results suggest that developers must be extra careful when reading, modifying, reviewing, or creating the last line or statement of a microclone, especially after performing copy-and-paste. As code quality consultant Thomas Kinnen said, "I perform tons of code reviews. After having read 'The Last Line Effect,' I check the last line or statement in a microclone extra carefully." This paper appeared in the International Conference on Program Comprehension; access it at http://goo.gl/bbkGH9.

"An Empirical Study on Quality Issues of Production Big Data Platform," by Hucheng Zhou and his colleagues, described an analysis of 210 incidents from Microsoft ProductA (anonymized), a company-wide multitenant big data computing platform serving thousands of customers from hundreds of teams. System and customer factors led to the highest proportion of incidents. This suggests that

- broader testing, especially online testing in real production, is vital to detect problems early and
- better training and process management can avoid operational faults and misuse.

Another important aspect of this research is the catalog of telemetry data, which other providers of big data frameworks can use. The metrics in this catalog include job- or vertex-specific metrics (for example, latency metrics and task I/O metrics), performance counters (for example, CPU usage and memory usage), and job or vertex logs (for example, log entries of interesting execution points). The paper also described mitigation strategies that other providers of big data frameworks can use. It was part of the ICSE '15 Software Engineering in Practice track; access it at http://goo.gl/AYcGhR.

"Mining Energy-Aware Commits," by Irineu Moura and his colleagues, examined techniques developers use to save energy, which is especially important for devices with limited battery life, such as mobile devices. Users often factor energy efficiency into their choice of mobile apps. Even though this quality is important to users, little is known about the strategies adopted to minimize energy consumption or their impact on software quality.

The information mined from 371 energy-aware commits from GitHub identified these energy-saving techniques:

- altering the frequency and voltage of the CPU and peripherals such as Wi-Fi,
- using power-efficient libraries,
- disabling features,

PRACTITIONERS' DIGEST

- fixing energy-related bugs,
- implementing low-power idling, and
- manipulating time-outs to stop computation.

The authors also identified possible negative side effects, including corruption of serial transmission and low responsiveness or performance. Furthermore, the results showed that developers were often unsure whether their energy-aware strategies were effective. There is a need for better-documented energy libraries. This paper was part of the 12th Working Conference on Mining Software Repositories; access it at http://goo.gl/AcRzDq.

"An Empirical Study of Architectural Change in Open-Source Software Systems," by Duc Minh Le and his colleagues, reported on an analysis of several hundred versions of 14 open source Apache systems. The authors' key findings include these:

- A semantic (conceptual) view reveals notably different aspects of system evolution than the corresponding structural view.
- Architectural changes can occur inside components even when the overall architecture remains stable.
- The package structure provides only a limited indication of the system architecture.
- Dramatic architectural changes tend to occur both between the end of one major version and the beginning of the next and across one or more minor versions.

These findings provide insight into how architectures change over time. This paper was part of the 12th Working Conference on Mining Software Repositories; access it at http://goo.gl/YM4cPT.

"Supporting Physicians by RE4S: Evaluating Requirements Engineering for Sustainability in the Medical Domain," by Birgit Penzenstadler and her colleagues, presented Requirements Engineering for Sustainability (RE4S). RE4S is a method that uses checklists and reference models to guide software engineers in including sustainability throughout requirements engineering, from identifying stakeholders, to analyzing the domain, to defining a usage model, and finally to specific requirements. To help software engineers identify sustainability concerns, the checklist starts with four questions about the system's purpose, impact, stakeholders, and goals and constraints. RE4S also provides reference models for sustainability goals and stakeholders.

The paper included a case study of Cognatio, a system that supports communication between patients and physicians. Patients can track prescribed medications and observed symptoms; physicians can send reminders and review patient data. Using RE4S to consider sustainability during requirements engineering improved the software's social aspects (for example, interaction between user groups analyzed from different perspectives) and environmental aspects (for example, avoiding overprescription and misaligned prescriptions). So, most of the developed artifacts reflected sustainability concerns that would have been missed otherwise. This example illustrates how using RE4S can help developers systematically integrate sustainability goals and requirements with other requirements, and refine them into software-specific constraints considered during design and implementation.

An online version of RE4S is at http://birgit.penzenstadler.de/se4s. This paper was part of the 4th International Workshop on Green and Sustainable Software; access it at http://goo.gl/RuyBJ0.

Feedback or suggestions are welcome. In addition, if you try or adopt any of the practices included in the column, please send the paper authors and Jeffrey Carver (carver@cs.ua.edu) a note about your experiences.

JEFFREY C. CARVER is an associate professor in the University of Alabama's Department of Computer Science. Contact him at carver@cs.ua.edu.

AIKO YAMASHITA is a data analyst and entrepreneur at Yamashita Research and is an adjunct associate professor at Oslo and Akershus University College of Applied Sciences. Contact her at aiko.fallas@gmail.com.

LEANDRO MINKU is a lecturer (assistant professor) in the University of Leicester's Department of Computer Science. Contact him at l.l.minku@cs.bham.ac.uk.

MAYY HABAYEB is a master's student in Ryerson University's Data Science Laboratory. Contact her at mayy.habayeb@ryerson.ca.

SEDEF AKINLI KOCAK is PhD candidate and research assistant in Ryerson University's Data Science Laboratory. Contact her at sedef.akinlikocak@ryerson.ca.

Editor: **Robert Blumen**
Salesforce Desk.com
robert@robertblumen.com

Barry O'Reilly on Lean Enterprises

Johannes Thönes

EPISODE 234 OF Software Engineering Radio features a conversation between host Johannes Thönes and guest Barry O'Reilly, coauthor of *Lean Enterprise: How High Performance Organizations Innovate at Scale*. A lean enterprise is a large organization that manages to innovate while keeping its existing products in the market. Other topics from the interview, omitted here for space, include the management of lean projects as a portfolio, the importance of failure, and how the British government adopted a lean approach in response to a spectacular failure in their healthcare system. Download the entire episode at www.se-radio.net. –Robert Blumen

Johannes Thönes (JT): Your book *Lean Enterprise* is, in essence, an extension of the lean-startup idea [a method for developing business and products that Eric Ries proposed in 2011] to big companies. Is that correct?

Barry O'Reilly (BO): Yes, in part. We wanted to capture the essence of the lean-startup principles, such as scientific thinking, and then apply that across an entire organization. In the book, when we talk about new-product development, the lean-startup approach is a great way to discover what works when building new products. But it's also a great way to reduce uncertainty—for example, what kind of CMS [content management system] and tools to use, or how to experiment when building new processes.

Start by thinking about the outcomes [you] want to achieve, then devise experiments to move toward that outcome, and learn along the way. Those are the core elements: building a culture of experimentation and learning in your organization so that it can adapt to the changing circumstances both in the market and internally. By creating that culture of learning and empowering people, you create a high-performance culture where the people who are doing the work don't have to ask for permission to try new things, make things better, and become engaged. That's where we see the most radical improvements in organizations.

JT: You said that lean startup is about scientific thinking. What does that mean for you?

BO: Eric Ries talked about the concept of build, measure, learn. What's the smallest thing you can build to test [an] idea, specifically with the customers that it's designed for? The scientific method is what we use to explore uncertainty. For years, humans have made observations, formulated hypotheses, and then created experiments to test that hypothesis. The outcome of most experiments is some sort of learning. Either you invalidate your hypothesis, in which case you might want to change what you do, or

0740-7459/15/$31.00 © 2015 IEEE

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

Recent Episodes

- 236—Rebecca Parsons, chief technology officer at ThoughtWorks, shares her thoughts about evolutionary architecture with host Johannes Thönes.
- 235—Ben Hindman and host Jeff Meyerson discuss Apache Mesos, a distributed-systems kernel.
- 233—Fangjin Yang, creator of the Druid real-time analytic database, talks with host Robert Blumen about online analytical processing and making it real-time.

Upcoming Episodes

- Host Josh Long talks with Gradle's Cedric Champeau about the Groovy language.
- Cofounder of arc42 Gernot Starke talks to host Eberhard Wolff about architecture documentation.
- Host Stefan Tilkov discusses coordination in distributed systems with Stripe's Kyle Kingsbury.

you get confidence that you're on the right path and you want to continue.

JT: Can you give an example of an experiment you would run in a lean enterprise?

BO: A lean startup I [was involved in] was trying to create a wine company. We formulated a business hypothesis: people who take photos of bottles of wine might want to buy them. Under the business model, if someone took photos of a bottle of wine, we could send two bottles to them in two days. How do you test that hypothesis? Typically, you would start by crafting an amazing business case and then spend months analyzing how the supply chain works and trying to sign deals with different wineries. It could be months before [that idea gets] to market. But the lean-startup approach is to find the smallest possible experiment or test to reduce the uncertainty of that business model.

We mapped out, literally, who we thought our customers were and how they would interact with the business. Then we thought about how to test that. Where do people take photos, and where do they store them? I went on Twitter, did a search for red wine, and looked for people who had taken photos of bottles of red wine. I tested part of my hypothesis: Do people take photos of bottles of wine? Yes. I could move forward.

The second part of the hypothesis was, if they took a photo, they might buy [the wine]. How could I test that? What I did next was create a Twitter account of a fake company selling wine. Then I created a customized tweet. I [looked at a user's photo of red wine], read the name of the wine off the bottle, did a bit of research to find out how much it cost, and then sent [the user] a customized tweet based on the picture of wine he or she had taken. "Hi Johannes, it looks like you're enjoying [this wine]; would you like to buy two bottles for £20? Click this link to purchase." Very quickly, with very limited investment, I had created an experiment.

JT: And then you count the people actually clicking the link.

BO: Exactly. It shows that people are interested. It doesn't necessarily guarantee they'll buy, but it shows the method of reducing the uncertainty in a business model by making small investments from which you can quickly learn what people want.

JT: How long did it take you to build that experiment?

BO: About three minutes.

JT: What was the outcome?

BO: As we went forward, we started to build more fidelity. The first tweet just went to a wine wholesaler. We had evidence that people were clicking on the links, so we built a very simple landing page that was customized to the bottle of wine and showed the price and a buy button. This way we could validate our test. People might have clicked the link out of curiosity, but would they buy? As you get further in, you see very quickly that this isn't a scalable robust solution that can serve hundreds of millions of people. But I'm

finding out if the business model is valid. And I'm reducing uncertainty at a very low cost.

JT: How does that translate to the enterprise?

BO: There are many aspects of driving innovation in organizations, but the main thing is that innovation needs a strong executive mandate. It needs support from the board, people who will protect teams as they try new ways of working. Whatever it might be, we need to work in a short iteration because that forces us to have a feedback loop to measure how we're performing.

When you think about build, measure, learn, it's about creating a feedback loop. Iterations are another way to create a feedback loop and understand how you're moving forward on a regular basis. So you get continuous reevaluation. The challenge when you're working in longer feedback cycles is that it can take months to get feedback from customers to find out if you're building the right thing for them, if it's actually what they want.

JT: You discuss the three-horizons model in the book. Can you define that for us?

BO: It's a model from McKinsey [a management consulting firm]. It was formulated by one of their partners a long time ago. Horizon one is essentially your current businesses. What's generating today's cash flow? This normally consists of business models, products, or services that are within [their first] 12 months. Horizon two is from 12 to 36 months. These are things that are not necessarily generating the most of your revenue today, but you can see that they'll drive most of the revenue for your business in two or three years.

Then you have horizon three, which is anywhere from three to five years out. These are your highly experimental projects. When you ask, "What's the market going to look at in three or five years?," you don't know. But if you're not making any bets, how are you ever going to be able to maximize that? Some good examples of this might be, what does the Internet of Things mean now? What kind of services should banking organizations create to help maximize the leverage from the Internet of Things? What are the innovations three or five years out that are going to drive your future growth? What you need to do is make some investments to explore that idea and learn how the technology might work.

JT: Can you give an example of these three horizons in terms of products?

BO: The company we talk a lot about in the book is Intuit. Intuit is probably the largest self-service accounting firm in North America. They provide accounting services for small to medium-sized businesses

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

as well as individuals. Sixty percent of their investment budget goes into horizon one. That's their growing category. They want to learn what kinds of things are successful, like TurboTax and Mint. In horizon two, they use about 30 percent of their budget. That's really about increasing growth and efficiency, like QuickBooks and online accounting.

The last 10 percent of their operational expenses is really fun. They invest in really experimental things. One of the most interesting things to come out of that is a tool called SnapTax. I encourage listeners to go to the Apple App Store or Google Play and search for SnapTax. I've never seen a higher rating for any product in the App Store.

They started with a team of just three people. When the team started to build it, they were told "It's a crazy idea," "It's never going to work," "It's insane," and "What are you doing?" The concept of SnapTax is that at the end of the year when you get your tax [form], you take a photo of it. [The app] scans the whole document and does your tax return in two seconds. It's insane, right? But the team achieved it because Intuit is highly invested in using lean startup and lean thinking across their entire organization. They invest in ideas that are going to make them successful in the future. They challenge themselves around these horizons. They ask themselves, "Do we have a balanced portfolio?" "Are we working in horizon three?" "Are we looking after horizon one?" There are strategies to manage each one of those horizons. Horizon three is very experimental, two is about growing your business, and one is about sustaining and making sure that the business is as optimized as efficiently as possible.

JOHANNES THÖNES is a software developer at ThoughtWorks. Contact him at johannes.thoenes@gmail.com.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

Software Experts Summit

21 October 2015 • Beijing, China

The Future of Software Engineering

REGISTER NOW!

This dynamic event, brought to you by *IEEE Software* magazine, will feature a variety of speakers to cover the latest important technologies that are transforming our field to provide attendees with the information and resources to ride the wave of the latest changes in software engineering and keep up to date with the tools they need as practicing software engineers.

Learn from thought leaders from both industry and research. Speakers include:

Jan Bosch
Chalmers University of Technology

Hong Mei
Peking University

Diomidis Spinellis
Athens University of Economics and Business

Michiel van Genuchten
VitalHealth Software

Eoin Woods
Endava

Presented by IEEE Software

IEEE computer society

www.computer.org/ses15