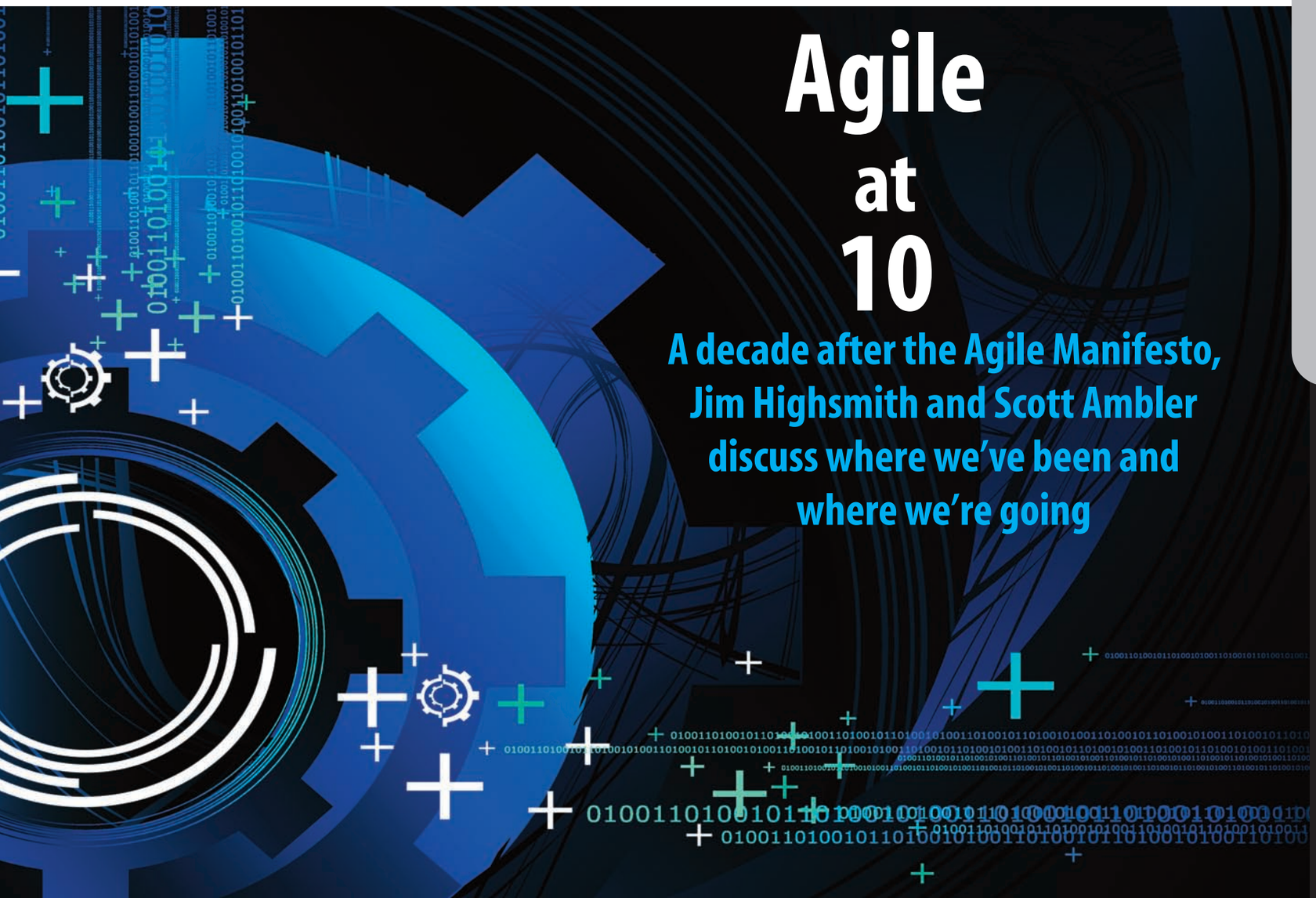
SPECIAL ISSUE

Dr. Dobb's Journal

APRIL 2011



Agile at 10

A decade after the Agile Manifesto, Jim Highsmith and Scott Ambler discuss where we've been and where we're going

ALSO INSIDE

UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

SPECIAL ISSUE April 2011

COVER STORY

More on DrDobbs.com

Overthinking Embedded Systems
Al Williams often says there are two things that are really simple: Things that are really simple (of course) and things that are really complex.
http://drdobbs.com/blogs/embedded-systems/229300220

A Base Class for Intrusively Reference-Counted Objects in C++
Smart pointers encapsulate pointers to dynamically allocated resources such as heap memory or handles. They relieve you of explicit resource deallocation, therefore simplifying resource management, exception-safe programming, and the "Resource Acquisition Is Initialization" (RAII) idiom.
http://drdobbs.com/cpp/229218807

Improving OLTP Database Throughput via Batching
Parallel threads and JDBC connection pools can be useful, but should not be used blindly. When handling numerous transactions with small overhead on heavily contended systems, batch processing often provides a better solution.
http://drdobbs.com/database/229218866

Jolt Awards: Security Tools
This year's Jolt awards in the security category show that this segment is very much alive, with new tools that show in how many different ways threats can arise and be parried.
http://drdobbs.com/joltawards/229300090

Azure Worker Role is Not There Yet
Microsoft's foray into the Cloud service providers (aka Azure) tries to play both in the Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) playgrounds.
http://drdobbs.com/blogs/architecture-and-design/229300304

Keep Agile Going

By Jim Highsmith

Last month, was the tenth anniversary of the Agile Manifesto. To mark the occasion, I ceded my editorial space to Jim Highsmith, one of the original signers of the manifesto, so he could share some reflections on the Agile movement, its present and future. — Ed.

My first reflection is that ten years of Agile methods have had an extremely positive impact on the industry. In his classic, *The Secrets of Consulting*, (http://is.gd/UaZI1f) Jerry Weinberg offers us his Law of Raspberry Jam, "The wider you spread it, the thinner it gets." I thought about this recently as I've read blogs and articles from Agilists who are bemoaning the state of the Agile movement. They are concerned that the movement has gone awry, that people are practicing prescriptive agility, that they are not living up to the vision of the founders. So, what did they expect?

As any movement expands from its narrow early base of practitioners, others take it in unforeseen directions — some good, some not so good. That's just the way movements go. We can wax nostalgic about the "good old days," can reflect on progress and try to redirect, or we can innovate and move forward. As we reflect on 10 years of Agile, I'd prefer to focus on the positive — how we've learned to deliver value to customers faster, how we've brought quality to the forefront in ways that haven't happened before, and how we've improved the quality of work places around the globe.

Innovators, Imitators, and Idiots

My second reflection is that while overall the Agile movement has had a positive impact on the world of software development, there are improvements to be made.

In a PBS interview concerning the financial meltdown, Warren Buffett commented on the natural progression of how good ideas go wrong. He called this the "three I's." "First come the innovators, who see opportunities and create genuine value. Then come the imitators, who copy what the innovators have done. Sometimes they improve on the original idea; often they tarnish it. Last come the idiots, whose avarice undermines the innovations they are trying to exploit." [see *Practically Radical,* by William C. Taylor: http://is.gd/ASct0p].

How this applies to Agile: We are more familiar with the technology adoption curve — enthusiasts, visionaries, pragmatists, conservatives, and skeptics. Many pundits project that the Agile movement has crossed the "chasm" (popularized by Jeffery Moore) into wide acceptance in the industry. But what about Buffett's progression? Are we in danger of being overtaken by the imitators and the idiots?

There has surely been a large influx of imitators into the Agile movement — an inevitable development as the market for Agile services and tools has expanded rapidly. Many of these imitators added improvements while some have tarnished the Agile brand. And, there have been a few idiots from time to time, people and companies who barely know how to spell Agile hanging out their Agile shingles, often giving Agile delivery a bad name in the process.

But the real question is how do we keep moving forward as a movement? There are at least four keys ways that come to mind: continue to innovate, balance idealism and practicality, reinvigorate our Agile value roots, and unify rather than splinter.

Let me explain.

Innovate. I'm encouraged by the continuous innovation I see in Agile: DevOps, continuous delivery, the conversations over technical debt, Lean, Kanban, Agile/Adaptive Leadership, and more. Continued innovation combats the creep of staleness that tends to infect movements after a few years.

Idealism vs. Practicality. As Agile permeates into larger organizations; we have to focus on both idealism and practicality. Many people don't care much about esoteric arguments — they care about results. Idealism and innovation are absolutely necessary for a vibrant movement, but they need to be balanced with a dose of practicality in organizational transitions.

Reinvigorate. The power and attractiveness of the Agile movement lies in its values as expressed in the Agile Manifesto and the Declaration of Interdependence. The more we can emphasize the dual importance of both doing Agile (practices) and being Agile (values), the better we can move forward on a more solid foundation.

Unify vs. Splinter. As any movement grows, there are times when it tends to splinter and times (sometimes) when it unifies. I appreciated Mike Cohn's recent Scrum Alliance update in which he said, "We want Scrum teams to look beyond the Scrum framework and experience the great ideas found in our sister approaches of Lean, Extreme Programming, Kanban, Feature-Driven Development, DSDM, Crystal, Adaptive, and more." Efforts like this to bring the Agile/Scrum/Lean/Kanban/etc. communities together, rather than continue to splinter further, leaves less space for the idiots to exploit.

The important goal is to rotate back and forth between innovators and imitators — advancing and then consolidating — without falling into the idiot trap as did the financial industry. I hope that by focusing on Agile values and principles, continuing to innovate, balancing between idealism and practically, and taking opportunities to unify rather than splinter will keep the idiots at bay.

Unorthodox, Unconventional, and Innovation

My third reflection is that the improvements to Agile lie, not in popularization but in continued innovation and innovators.

Paul Farmer was an unconventional doctor. Tracy Kidder (in *Mountains Beyond Mountains*; http://is.gd/zbEpWI) describes Farmer's Haitian clinic, located in the difficult to access highlands, where Farmer often hiked hours to see a single patient or treated multiple antibiotic resistant tuberculosis patients with expensive new drugs. Farmer didn't fol-

low the typical public health cost-benefit approach; he treated individuals and riled that very public health community with his unconventional approach.

In *Different: Escaping the Competitive Herd* (http://is.gd/F7jHbQ), Harvard Business School professor Youngme Moon, writes about "succeeding in a world where conformity reigns but exceptions rule." Product differentiation, sustainable differentiation, she says, "is rarely a function of well-roundedness; it is typically a function of lopsidedness."

"The industry appears to be at a tipping point to a far more strategic opportunity to implement agility at an enterprise level"

Walk through your neighborhood grocery and look at hand soap — hundreds of variations, little differentiation. Innovation requires stepping out of comfort zones and being different from others. I know one IT organization that has figured out how to capitalize more than 90 percent of their software development expense by looking beyond tradition. I know another IT organization that has eschewed the common off shoring wisdom, keeping development onshore and outperforming their competitors.

Ten years ago, in February 2001, a group of 17 unconventional, unorthodox individuals got together, wrote the Agile Manifesto and launched the Agile movement. In the last 10 years Agile delivery has often moved from the unconventional to the conventional, from the maverick to the conformist. What now? The roots of agility are in complex adaptive systems and the notion of operating at the edge-of-chaos — that knife-edged balancing point between chaos and stifling structure.

As we move into the teenage years of the Agile movement, we can't forget that agility isn't about structure, practices, and conventions. Agility is ultimately about living on the edge, of pushing the envelope, of standing out in a crowd, of being lopsided in a world of conformity.

Enterprise Agility

My final reflection on the Agile movement is about an expanded future. Enterprises are beginning to expand on their success with Agile software development. They're looking at bringing Agile principles and practice to other parts to the enterprise. The industry appears to be at a tipping point to a far more strategic opportunity to implement agility at an enterprise level. In the face of markets characterized by rapid change, complexity, and ambiguity, enterprise executives are finding new ways of harnessing their organization's creativity, adaptability, operating prowess, and customer relationships. In various circles, this leadership style for the future has been called creative leadership, adaptive leadership, Management 2.0 (and 3.0), collaborative leadership, light-touch leadership, and Agile leadership.

The Teenage Years

Teenagers are unpredictable. The teen years are bumpy, sometimes teens do great things, and sometimes they get into trouble. As the Agile movement matures past its 10th anniversary, it will be unpredictable also.

— Jim Highsmith is an executive consultant at ThoughtWorks.

Agile at 10: What We Believe

A Non-Update to the Agile Manifesto

By Scott W. Ambler

The Agile Manifesto had its beginnings in a meeting held in February 2001 at the Snowbird ski resort just outside of Salt Lake City. To celebrate the ten-year anniversary of this event, Alistair Cockburn, who organized the original meeting, invited a group of agile luminaries back to Snowbird to discuss the future direction of agile. The original 17 manifesto writers were invited, although not all were able to attend, and I was among the people who did participate. In this article, I discuss what we concluded, how the event was organized, and some of the challenges faced by the agile community.

What We Believe

First, let's jump to what most people will perceive to be the end results. As a group, we agreed on the following four belief statements:

1. Demand technical excellence
2. Promote individual change and lead organizational change
3. Organize knowledge and improve education
4. Maximize value creation across the entire process

Let's dive down into what these belief statements imply. The first statement, demand technical excellence, is a reflection both of the general values of the working group and also our frustration with what we're seeing in practice — or more accurately, not seeing in practice. The Agile Manifesto exudes the philosophy that technical excellence is critical to your success in the software game, yet this requires both skill and discipline to pull off in practice. Many agile teams do in fact manage this feat — various *DDJ* surveys have shown over the years that agile teams produce higher levels of quality on average. However, the adoption rate of quality techniques such as test-driven development (TDD), refactoring across all architectural tiers, and following common development conventions aren't as high as the agile rhetoric would lead us to believe. There is room for improvement, and our hope is that this first statement will motivate agilists to do so.

The second belief statement is to promote individual change and lead organizational change. Once again, we've definitely come a long way as a community but we have further to go when it comes to improving our approach to software development. At the individual level I believe that we need to life-long learn, teach people to observe what

works for them (and what doesn't) in the situations that they find themselves in, and to then act on those observations. At the organizational level it is important to understand both the mechanics and the supporting human behavioral aspects of change processes and then invest the time and resources to effectively execute on them. One aspect of this is to focus on the organizational ecosystem as a whole and not just on IT — in fact, many agilists are instead still mired in the details of software development and unable to see the enterprise forest for the development trees.

Organizing knowledge and improving education is the focus of the third belief statement. Several people in the group voiced their frustration around the lack of a coherent and consistent agile body of knowledge. In many ways, this is a reflection of our times — the Internet makes it easy for people to publish articles, post ideas in blogs, tweet, write wiki pages, and so on, which in turn promotes a dispersed cacophony of voices. This cacophony can be debilitating when you want resources which support teaching agile concepts, particularly at the college and university level but also at the individual learning level. Heresy aside, perhaps an organization such as the Agile Alliance could start amalgamating some of this knowledge, and even go so far as to create an Agile Book of Knowledge (ABoK). Minimally, it's clear that the agile community could do more to reach out to universities and colleges to help them understand and teach agile more effectively.

Last, and certainly not least, is the fourth belief statement that we should strive to maximize value creation across the entire process. One of the fundamental messages of the Disciplined Agile Delivery (DAD) process framework is that we need to look beyond the software construction focus of the Agile Manifesto to address the full solution delivery effort. But this is clearly not the full picture either as there is much more to IT than solution delivery, and certainly much more to your organization than IT. Some of the greatest challenges that we face with agile adoption are the business issues surrounding IT, particularly when it concerns financial issues such as funding delivery projects, IT governance, and understanding and interacting effectively with business stakeholders. As the lean community implores, we need to optimize the whole, and agile software development is only one small part of your overall organizational ecosystem.

How It Happened

To help bring a bit of transparency to this event, I'd like to briefly describe how it was organized. In early January Alistair Cockburn sent out invitations to a number of people whom he felt had added value within the agile movement over the years. Not everyone was able to attend for various reasons, and perhaps a few people who weren't invited should have been, but from what I could tell he brought together a group who was representative of the overall community. I'd like to once again express my thanks to Alistair for taking the lead on this effort.

Luckily Alistair decided to hire professional facilitators Robert Moir and Janet Danforth to organize the event, and I truly believe that we couldn't have succeeded without the two of them. A few weeks prior to the event, they called the confirmed attendees to discuss what they wanted to achieve when we got together. Then, the night before kick-off when we gathered in the evening for drinks, we discovered that index cards containing questions drawn from these pre-interviews had been distributed around the tables. The goal of the cards was to stimulate conversation, not that we needed any help with that, and to communicate to everyone the range of issues to consider the following morning.

The working session itself was held on Saturday February 12th, with an idea generation session in the morning from 8:30 to 12:30 and a focusing session held in the evening from 5 to 7:30. During the morning session we organized into seven subgroups, and we iteratively addressed seven topics — value; agile backlash/retrospective; technical/agile process; other communities and business/enterprise process; culture; education/training; and the future — which the index cards from the night before had been assigned to. Each subgroup started with one of the topics and identified issues pertaining to that category with sticky notes. Then the subgroups rotated to the next category to review the previous work and add any new ideas. We proceeded iteratively until each subgroup had contributed to each topic category.

"We recognized that there were several issues that we were avoiding in our discussions. This avoidance was due in part to the prevailing culture of the agile community, in part due to commercial interests of several people in the room, and in part because we were originally asked by Alistair to play nicely together. So we invested some time to identify these 'elephants in the room' so that they could be discussed."

The goal of the evening session was to come to a consensus around a statement of our beliefs. To do this, we re-categorized the issues we had identified in the morning into things that we can make good headway on, things we don't know how to address, and things that we believe can't be addressed. For example, better communicating the role of architecture in agile development is something we could make good headway on, whereas the fact that many organizations rate their employees in part based on degrees or certifications is something we'll never be able to address as a community. The issues which we believed could be addressed were clustered by topic, the topics were prioritized, and after much discussion and wordsmithing the four belief statements were identified.

The Elephants in the Room

Late in the morning, we recognized that there were several issues we were avoiding in our discussions. This avoidance was due in part to the prevailing culture of the agile community, in part due to commercial interests of several people in the room, and in part because we were originally asked by Alistair to play nicely together. So we invested some time to identify these "elephants in the room" so that they could be discussed. I've listed these below so that you can hopefully gain better perspective as to some of the issues holding back the agile community. These issues are:

- There are other effective ways of information discovery without writing code — Agile modeling, for example.
- Managers and management are bad — This is a common refrain

of many programmers, including agile ones.

- Politics within the community — There are many factions within the agile community, they don't always agree nor do they even get along.
- Failure to dampen negative behaviors — We often promote good behaviors within the agile community, such as allowing requirements to emerge over time, but do not push back sufficiently against poor behaviors such as documentation-based governance strategies.
- Technical debt — Are we really paying down technical debt within organizations or merely talking about it?
- Anarchism — Some of the alliances/organizations within the agile community purporting to be open efforts are little more than Ghaddafiesque groups in practice.
- Hypocrisy — The reality of agile is often very different than the reality, something I've explored over the years via *DDJ* surveys.
- Context and applicability — Many agile strategies are promoted as if they are the best way of doing something, yet in practice, they prove to work well in some situations but very poorly in others. The Agile Scaling Model (ASM) is a contextual framework that may be used to address this problem.
- Context gets in the way of dogma — Related to the previous point (Context and applicability), it's uncomfortable to admit that your best strategy isn't the only effective way of work, nor always the best.
- Certification — As I pointed out in January, the agile community continues to build up integrity debt by tolerating questionable certification schemes.
- Pretending agile is not a business — 'Nuff said.
- Abdicating responsibility for product success (to product owners) — Scrum's product owner concept is useful in some contexts, but in practice it is a very difficult role to fill and reflects a serious lack of understanding as to the difficulty surrounding requirements management activities.
- Agile Alliance — The AA showed great promise years ago and could potentially have lead the way in a true paradigm shift within IT, but in recent years has done little more than organize the annual Agile conference in the United States and provide funding for local agile events.
- Role of architecture and design — Agile strategies for architecture and design modeling have been long overshadowed by programming practices and Scrum certification.
- Self organizing teams — The 2010 "How Agile Are You?" survey found that only 56% of "agile teams" were self organizing, and indication that many organizations are still struggling with this concept.
- Elitism — The agile community needs to find ways to be more welcoming to project management professionals, data professionals, software architects, and other specialist communities within IT.
- Business value (what is it?) — We sure do talk a lot about providing business value, but don't seem to have a good handle on what it actually means.
- Scaling naivety — As I show in my work around the ASM, there's a bit more to scaling agile than addressing the issues around large teams and geographically distributed teams, and it's certainly more complicated than holding a "Scrum of Scrums" to coordinate everything.

- Commercial interests — Perhaps we'd see more evolution in agile methods if we weren't so focused on charging people to get certified in them.
- Sensing failures — Although *DDJ*'s Project Success surveys over the years have found that agile project teams enjoy a higher success rate than traditional project teams it isn't that much better and we don't have a perfect track record.
- How do we know that we're delivering value through software? — If we can't define value surely we must also be struggling to measure it as well.

The four belief statements are not meant to be an addendum to the Agile Manifesto. Several of us, myself included, felt that the manifesto should be updated to reflect what we've learned as a community these past ten years. An equal number within the group, however, felt that the manifesto shouldn't be updated. It was clear that we wouldn't reach any sort of consensus on this issue, let alone on what the changes should be, so we decided to leave well enough alone for now. Interestingly, many of the "elephants in the room" could begin to be addressed by improvements to the manifesto. Go figure.

Further Reading

For more information about the event, visit the 10 Years of Agile site at http://10yearsagile.org/.

I discuss The Agile Manifesto (http://www.agilemanifesto.org/) in detail in my article "The Agile Manifesto Examined" (http://www.ambysoft.com/essays/agileManifesto.html) and suggest several improvements in "Reworking the Agile Manifesto" (http://is.gd/e9DG2Z).

My January 2011 *Dr. Dobb's* article, "Certified Scrum Master Examined" (http://drdobbs.com/architecture-and-design/229000996), summarized the results of a recent survey exploring what people really think about the CSM.

The Disciplined Agile Delivery (DAD) (http://is.gd/kjdTKe) is a hybrid process framework capturing strategies from Scrum, XP, Agile Modeling, lean software development, Kanban, and other agile/lean sources.

"Scaling Agile: An Executive Guide" (ftp://public.dhe.ibm.com/common/ssi/sa/wh/n/.../RAW14211USEN.PDF) is a paper I wrote that explores the issues surrounding agility at scale.

The Surveys Exploring the Current State of Information Technology Practices page (http://www.ambysoft.com/surveys/) links to the results of all the *DDJ* surveys that I've run over the years.

My Agility@Scale blog at https://www.ibm.com/developerworks/mydeveloperworks/blogs/ambler/) discusses strategies for adopting and applying agile strategies in complex environments.

You can follow me on Twitter (http://twitter.com/scottwambler/).

— Scott W. Ambler is Chief Methodologist for Agile and Lean at IBM Rational.

Qt Application Development for Symbian

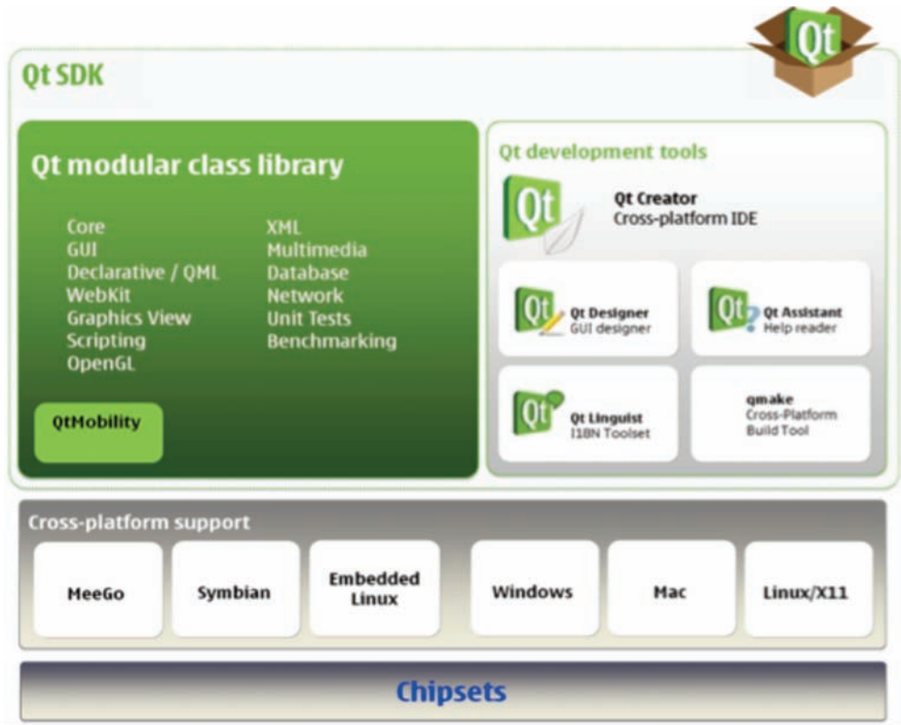
A full native application development environment for Symbian smart phones with Qt

By Eero Penttinen and Antti Saukko

In 2005, Nokia introduced standard C libraries for the Symbian OS. At that time, we wrote an article for *Dr. Dobb's* by the name of "Open C: Paving the Way for Porting" (http://www.drdobbs.com/cpp/198702204). Standard C infrastructure libraries can be found on all major operating systems today. With standard C libraries also in Symbian OS, application development became significantly easier, as did porting existing applications across different operating systems.

This time, we extend from the cross-platform standard C libraries to a full native application development environment for Symbian with Qt. Qt is a cross-platform native development framework for all major operating systems, including Linux, Windows, Mac, Symbian, and Maemo. This article explores developing applications with Qt for Symbian OS and deploying them on Nokia Ovi Store (http://store.ovi.com).

Qt supports Symbian-based S60 3.1, 3.2, 5.0 and the latest Symbian^3 devices. Today, Symbian holds the position of world's largest smart phone platform. This article focuses on writing applications with Qt,



and deploying them on Ovi Store. Please note that most of the steps presented here are valid for other operating systems as well, but we'll focus on the Symbian environment. We will also introduce our Qt SDK, which consists of IDE for developing Qt applications, called Qt Creator.

Getting Started

First, let's take a look at the components available for developers with Qt and Qt Mobility APIs. Then we'll delve into the Qt SDK, its IDE, Qt Creator, and the simulator. This will be followed by running an example on the simulator and the actual hardware. Once our application is done, we will demonstrate how to get it signed and deploy it on the Ovi store.

Qt SDK

Qt SDK is an all-in-one installation package that has environments for desktop, Symbian, and Maemo/MeeGo development. The SDK incorporates the latest version of the Qt framework and the IDE, Qt Creator. It is designed to provide all the resources you need to create great apps easily for both desktop platforms and mobile devices including Qt framework and Qt Creator. To further support mobile development, the Qt SDK contains mobility extensions called Qt Mobility APIs, support for on-device debugging, and Qt Simulator to enable quick mobile development on desktops.

The cross platform library Qt APIs are:

QtCore	Core non graphical classes used by other modules
QtGui	Graphical user interface (GUI) components
QtMultimedia	Classes for low level multimedia functionality
QtNetwork	Classes for network programming
QtOpenGL	OpenGL support classes
QtOpenVG	OpenVG support classes
QtScript	Classes for evaluating Qt Scripts
QtScriptTools	Additional Qt Script components
QtSql	Classes for database integration using SQL
QtSvg	Classes for displaying the contents of Scalable Vector Graphics files
QtWebKit	Classes for displaying and editing Web content
QtXml	Classes for handling XML
QtXmlPatterns	An XQuery & XPath engine for XML and custom data models
QtDeclarative	An engine for declaratively building fluid user interfaces, also known as Quick/QML.
Phonon	Multimedia framework classes
Qt3Support	Qt 3 compatibility classes (these are legacy support classes which are not supported in MeeGo and Symbian)

The Qt Mobility APIs are:

Bearer Management API	An API to control the system's internet connectivity state.
Camera API	Provides a framework to use Camera device features when supported by the platform.
Contacts API	An API enabling clients to request contact data from local or remote back ends.
Document Gallery API	Enables applications to search, filter and manipulate the meta data of files in local file system.

Feedback API	Allows applications to produce tactile feedback.
Landmarks API	As part of the Location API, it allows applications to create, retrieve, update and delete landmarks.
Location API	The Location API provides a library for distributing and receiving location data using arbitrary data sources.
Maps/Navigation API	Provides an API to access maps, landmarks and route information for navigation.
Messaging API	The Messaging API enables access to messaging services.
Multimedia API	Provides a set of APIs to play and record media, and manage a collection of media content.
Organizer API	Provides an interface to create, update, delete and share various organizer items such as events and to-do items
Publish and Subscribe API	The Publish and Subscribe API, containing Value Space, enables applications to read item values, navigate through and subscribe to change notifications.
Service Framework API	A set of Qt APIs to that allows clients to discover and instantiate arbitrary services.
Sensors API	The Sensors API provides access to sensors.
System Information	A set of APIs to discover system related information and capabilities.
Versit API	An API to manage Versit documents such as vCards.

Development of the Application

Before you start developing a Qt based application for Symbian, it is recommended that you register to Ovi Publisher. You will get a publisher identifier (UID), a certificate installer, and a developer certificate/key pair, which is needed during development and testing. The registration fee is 1 Euro. There is no additional fee on publishing to the Ovi Store. We will describe these concepts later in the article and in which phase these are needed.

For more details, see http://info.publish.ovi.com/ (and especially note the checklist for publishers).

The example in use, including all the files can be found at http://qt.nokia.com. In this specific case, we have used an example based on Qt Quick — a simple Calculator (the example comes with Qt itself in folder $QTDIR\demos\declarative\calculator).

With the Qt SDK, you can build the application for different targets depending on APIs used and speed up the development process significantly. It is always faster to test and debug an application in the desktop than in the device.

The desktop runtime itself is a good alternative for quick application development if your application doesn't depend on platform or Qt mobility APIs.

With Qt Simulator, you can test Qt applications that are intended for mobile devices in an environment similar to that of the device. You can change the information that the device has about its configuration and environment.

In Projects mode, select Qt Simulator and specify the needed Build and Run Settings for the project. Once done, Run the project and you will see how the application is launched in the simulator.

In the screen shot, you can see how the Calculator application has been launched in the Simulator which has N8 look and feel.

The QML Viewer is an application for loading QML documents that makes it easy to quickly develop and debug QML applications. It invokes the QML runtime to load QML documents and also includes additional features useful for the development of QML-based applications.

Build and Debug on Device

Once you are done with testing the application on your desktop, it is time build the application for the device. However, before doing that, it is important to understand what platform security on Symbian devices is all about.

To prevent the execution of unauthorized applications and to protect the end user, Symbian OS has platform security. For developers, this means that all the applications need to be signed prior to installation to a device. Signing authorizes applications to be both installed to a device and controls application access to mobile-specific data. Many basic applications can live with capabilities that can be self-signed for development purposes (so called "basic capabilities"). A later commercial deployment requires signing by Nokia. Whenever you use a feature that requires a special capability, you need to write the capability to the project file (.pro) that you can open up from the file view on the left hand side. Under the Symbian-tag you should see TARGET.CAPABILITY keyword.

In cases where Restricted or Manufacturer capabilities (CommDD, DiskAdmin, NetworkControl, MultimediaDD, AllFiles, TCB or DRM) are needed, signing needs to happen in Symbian Signed, where a publisher ID is required (and you need to have a company to get one). However, it is very rare that an application would need these kinds of capabilities, so in majority of cases, you can have your application signed via the Ovi publisher program. This is the recommended approach (since it is free of charge).

In the Calculator example, no specific capabilities are needed.

Just to illustrate how capabilities are specified in the .pro file, the following code snippet is given where *Location*, *NetworkServices* and *ReadUserData* capabilities are specified.

```
symbian {
  TARGET.CAPABILITY += Location \
                       NetworkServices \
                       ReadUserData
}
```

The best source to confirm required capabilities is the reference documentation for Qt and Qt Mobility — capabilities are documented in dedicated sections.

For more information about different capabilities, see http://wiki.forum.nokia.com/index.php/Capabilities

Build for Device

Select Symbian Device as the Target (as in Figure 1) and build the application.

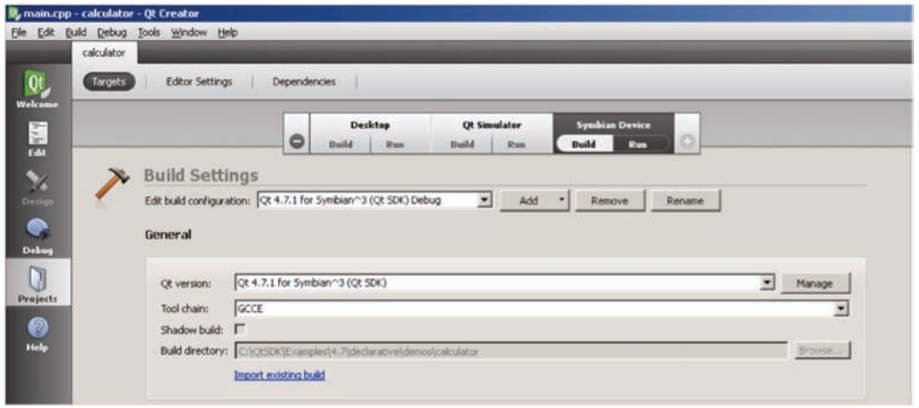


Figure 1.

Run applications on your device connected to your development PC via USB. Qt Creator will automatically detect your Nokia devices. Launching is done from the IDE itself. The Run Settings should look like Figure 2 (the Device information has been queried from the device). Also, in Run Settings dialog, you should specify the developer certificates you get from Ovi Publisher Program in the Create SIS package deployment dialog. This is needed if your application requires more capabilities than the basic capability set provides. Applications signed with a developer certificate are locked to a particular mobile device (or set of devices) and always display a warning that the application is untrusted on installation; see Figure 3.

Debug applications on your device. This is also supported for Symbian devices. The debug agent must be installed on the device, which can be done at the same time that the needed runtimes are installed on the device (Install TRK Debug Agent) to

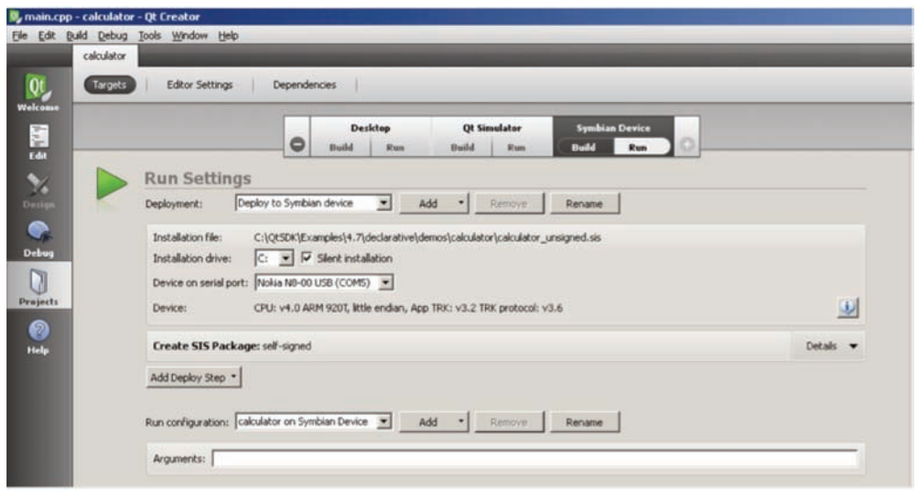


Figure 2.

Figure 3.

Symbian^3 device from Qt SDK menu. This is a powerful feature when debugging issues that are happening only in the device.

Do An Icon

SVG 1.1 and SVG 1.1 Tiny icons can be used in Qt applications. Some guidance how to create icons for Symbian platform can be found at http://wiki.forum.nokia.com/index.php/How_to_create_application_icon%28SVG%29_in_S60_3rd_edition.

Once you have created an icon for your application, add following statement to the .pro file and build the application.

```
Symbian {
   ICON = myicon.svg
   …
}
```

Your Application (UID)

During development, you can use any UID3 from the 0xE0000000 to 0xEFFFFFFF range. However, before deployment, you need to get proper UID3 from Ovi Publishing program.

There are three different types of UIDs: UID1, UID2 and UID3. UID1 defined the binary types, like executables, libraries and data. UID2 is not always used since it describes the installation package with additional information. UID3 is used to uniquely identify the binary (EXE or DLL) within the system.

Qt uses project files (.pro) to generate system specific make files with qmake. These project files also carry information needed to deploy your Qt application on the Symbian device. The UID3 is specified in the project file (.pro) like this:

```
symbian {
    TARGET.UID3 = 0xE0000069   # UID3 which can be used during
                               # development. Remember to change
                               # this before deployment.
}
```

Deployment of the Application

Qt applications can be published in Ovi Store. The Ovi Store functionality for Qt content has been tested and enabled for the following 20 Nokia device models (at the time this article was written):

- Symbian^3: N8-00, E7-00, C7-00 and C6-01 (these devices ship with Qt 4.6.3)
- S60 5th Edition: X6-00, C6-00, N97, N97 mini, 5800 XpressMusic, 5530 XpressMusic, 5235, 5233, 5230, and 5228
- S60 3rd Edition: E72, E71, E66, E63, and E52

The Nokia Smart Installer for Symbian makes the deployment of Qt applications to millions of Symbian devices easier even if those devices do not ship with Qt. The Smart Installer is an executable that is bundled with your Qt application with needed dependency information (such as the minimum Qt version for your application). The Smart Installer will detect during installation whether needed Qt libraries are already

installed and if they are, it will only install the Qt application. Otherwise, Smart Installer will automatically download the needed Qt libraries and install them to your device. With the Smart Installer, there is no need to distribute Qt binaries with your application, making the installation package significantly smaller and better suited for distribution using channels like Ovi Store.

Use of Smart Installer is mandated, even though devices such as the Symbian^3 based Nokia N8 have Qt pre-installed.

Qt Creator supports automatic generation of Smart Installer enabled packages. In Projects page, select the Symbian Device target and Run Settings and tick the box "Create Smart Installer package."

Modify the Application .pro File

The UIDs of the main executable and the application .sis file (myQtApp.sis) must be the same and must match the one specified in the *AppUID* field of Ovi Publish. This will ensure the application will launch within the Ovi Store client once the application is installed.

Vendor information in your application's package file must match your publisher name and cannot contain Nokia or Vendor.

```
symbian {
    TARGET.VENDORID = 0x70000002     # Use vendor ID you will get
                                     # from Nokia
    …
}
```

Create an Installable (SIS file) with Qt SDK

SIS is an acronym that stands for Software Installation Script. It is an archive for Symbian OS. Package files (.pkg) define the content of an installable sis-file. The package file also defines the platforms the package is intended for, the name of your software, and the version and

Qt *Capability Model*

- APIs protected with capabilities to protect the end user.
- Indicates application signing. Needed signing process depends on the application's capabilities.

Qt *Process Identity*

- Each process has a unique identifier (UID)

Qt *Data Caging*

- Apps can't access all data on the device

identifier of your application. Qt Creator takes care of creating template package files, which you can then modify if you so choose.

The Qt Creator makes this simple during development phase — needed SIS file is created when you want to run the application on a Symbian device (note: this has been changed in last edition of Qt SDK — earlier, SIS packages were created during build time). Qt Creator allows you to use either self-signed certificate or custom certificate when creating the SIS package for development.

Distribute the Embedded Package

Submit the app (unsigned SIS file with the UID provided) to the Ovi Store intake tool. The application will be tested based on Nokia content and store guideline, specific operator guideline and Symbian Signed test criteria. If it passes, it will be express signed by Nokia and published into Ovi Store.; and you're done!

— Antti works as a Software Architect in Nokia/Qt, holds a MSc in Computer Science , and can be contacted at antti.saukko@nokia.com. Eero works as Domain Lead in Nokia/Qt, holds a MSc in Computer Science, was a program manager for Qt for Symbian port, and can be contacted at eero.penttinen@nokia.com.

From the Vault

AntiPatterns

The Gang of Four's **Design Patterns** (http://is.gd/07vDMe) is a classic description of envisioning and managing object-oriented development. Back in 1998, **Dr. Dobb's Journal** published the work of a couple of authors who expanded on the Design Patterns theme —albeit in a bizzarro-world manner. **Antipatterns** (http://is.gd/dSs2iq) proposes that if patterns are good ideas that can be reapplied to new situations, unfortunate "AntiPatterns" can recur as well. By studying repeated failures in software development efforts, the authors diagnosed common AntiPatterns and wrote a book about identifying and avoiding/overcoming them. This article, pulled from our vault, introduced the concept.

— DDJ

By Hays W. McCormick and Raphael Malveaux

Unlike design patterns, AntiPatterns focus on software failures in an attempt to understand, prevent, and recover from them. To developers, AntiPatterns are a tool that bridge the gap between architectural concepts and real-world implementations.

The emergence of the design patterns movement has gone a long way toward codifying a concise terminology for conveying sophisticated computer science thinking. Still, the likelihood of success for practicing managers and developers is grim. According to J. Johnson in "Creating Chaos" (*American Programmer*, July 1995), for instance, studies of hundreds of corporate software-development projects indicate that five out of six projects are considered unsuccessful, about a third are canceled, and those remaining deliver software at typically twice the expected budget, having taken twice as long to be developed as originally planned.

While we can assume that all software was intended to solve some problem, these solutions are sometimes as bad as (or worse than) the problem they originally were intended to solve. It is these repeated failures, or negative solutions, that provide the source for the study of what we call "AntiPatterns."

AntiPatterns versus Design Patterns

Unlike design patterns, AntiPatterns start with an existing solution (or legacy approach). Most patterns assume a situation where there are few, if any, preexisting concerns that affect the design of a system. However, experience has demonstrated that situations with legacy and existing problems are much more commonplace in practice. The AntiPattern background and general form amplifies the problem in a way that helps you recognize the problematic structure, symptoms, and consequences.

The AntiPattern then presents a common solution that refactors the system to improve benefits and minimize consequences.

As Erich Gamma et al. point out in *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994, http://is.gd/07vDMe), design patterns focus on a particular software-design problem or issue. They contain a detailed discussion of the forces that influence the issue. In the design-pattern solution, the forces are either resolved or an appropriate tradeoff between the forces is established. This results in a divergence-convergence pattern where the context and forces diverge from the problem to provide a greater scope, and the solution diverges by resolving the forces into a single focused solution.

AntiPatterns, on the other hand, are based on a different rhetorical structure. AntiPatterns begin with a compelling problematic solution. From this solution, as Figure 1 illustrates, a discussion of the root causes focuses how the problematic solution is the result of incorrectly resolving the forces for a specific underlying set of problems. This convergence from a concrete situation to the more abstract underlying forces is a key component in communicating an understanding of how and why the problem exists. The Symptoms section provides additional clues for recognizing the improper resolution of the key forces. In the Consequences section, the implications of the problematic solution are discussed, thus providing a divergence similar to a design-patterns discussion of context. Finally, the Refactored Solution provides a better convergence of the underlying forces that lead to a better understanding of the problem and an effective method of resolving it.

AntiPatterns are therefore a natural extension to design patterns. Rather than codifying pure computer science, AntiPatterns focus on repeated software failures in an attempt to understand, prevent, and

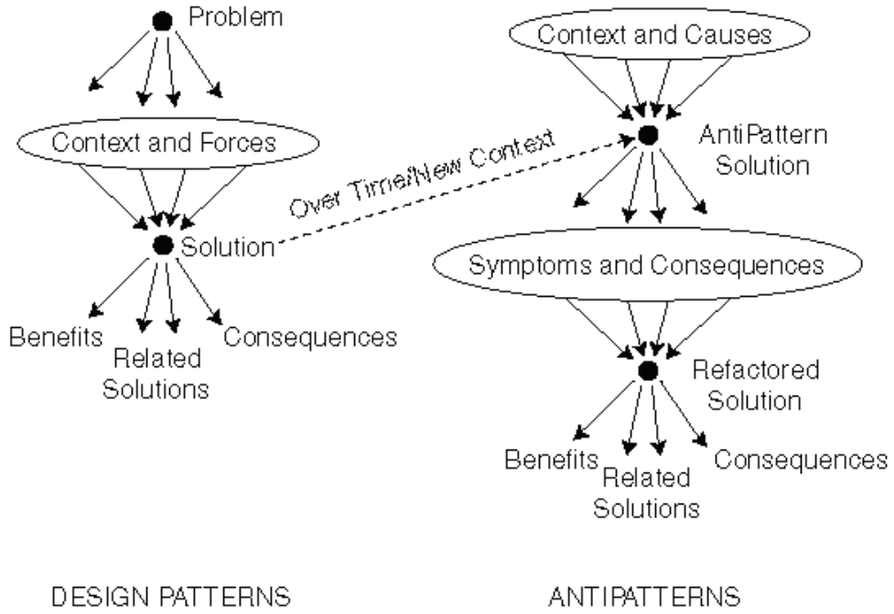


Figure 1

recover from them. To developers, AntiPatterns are a tool that bridge the gap between architectural concepts and real-world implementations. They are used to illustrate the common, critical problems faced by most software developers and enable learning from other developers' experiences. Only with an understanding of the causes and motivations of an AntiPattern can you ensure that your mistakes are not continually repeated within an organization.

AntiPattern Components

The use of a template defines the difference between design patterns and AntiPatterns and other forms of technical discussion. Templates

ensure that important questions are answered about each pattern in a pattern language, pattern catalog, or pattern system. The full AntiPattern template provides a wide range of information about a particular class of problems, including a discussion of the AntiPattern background, the general form, symptoms and consequences, root causes, a refactored solution, and an example detailing how the refactoring process can be applied to create an effective solution..For this discussion, we'll use an abbreviated version of a pair of AntiPatterns (the Lava Flow and Stovepipe System). The solutions include the following:

- General Form. Often including a diagram, the general form section identifies the general characteristics of this AntiPattern. This is not an example, but a generic version. A prose description explains the diagram (if any), and provides the general description of this AntiPattern. The refactored solution resolves the general AntiPattern posed by this section.
- Symptoms. Symptoms provide the observable indicators that suggest an AntiPattern might exist in a specific situation. They manifest through comments from individuals, the observed structure of software, organizational changes, the inability to meet targeted goals, and so on.
- Typical Causes. This section summarizes the key underlying causes that typically lead to the existence of a particular AntiPattern.
- Known Exceptions. AntiPattern behaviors and processes may not always be wrong. There are often specific occasions when this is the case. This section briefly identifies the primary exceptions to each full AntiPattern.
- Consequences. The Consequences section describes the harmful effects of the AntiPattern. They include the continuing damage caused by the AntiPattern as well as likely outcomes if corrective measures are not taken.
- Refactored Solution. This section explains a refactored solution that resolves the forces in the AntiPattern identified in the General Form section. (This solution is described without variations that are addressed in the Known Exceptions section.) The solution can be structured in terms of solution steps.

AntiPattern Name: Lava Flow

As Figure 2 illustrates, the Lava Flow AntiPattern is commonly found in systems that originated as research but ended up in production.

General Form

```
_
// This class was written by someone earlier (Alex?) to manage the indexing
// or something (maybe).  It's probably important.  Don't delete.  I don't
// think it's used anywhere - at least not in the new MacroINdexer module which
// may actually replace whatever this was used for…
class IndexFrame extends Frame
{
   // IndexFrame constructor
   //-------------------------------------------------------------------------
   public IndexFrame(String index_parameter_1)
   {
   // Note: need to add additional stuff here…
   super (str);
   }
   //-------------------------------------------------------------------------
_
```

Figure 2

It is characterized by the lava-like "flows" of previous developmental versions strewn about the code landscape, but now hardened into a basalt-like, immovable, generally useless mass of code that no one can remember much (if anything) about. This is the result of earlier developmental phases where developers tried out several ways of accomplishing things, perhaps while in a research mode. Predictably, a deadline arises and, in the rush to deliver, sound design is cast to the winds and the documentation of the system becomes sketchy, irrelevant, or altogether nonexistent.

The result is several fragments of code — wayward variables, classes, and procedures that are not clearly related to the overall system. In fact, these flows are often so complicated and spaghetti-like that they seem important but no one can really explain what they do or why they exist. Sometimes a developer can remember certain details, but everyone has decided to "leave well enough alone" since the code in question "doesn't really cause any harm, and might actually be critical, and we just don't have time to mess with it."

"Lava Flows are expensive to analyze, verify, and test — All such effort is expended entirely in vain and is an absolute waste"

While it can be fun to dissect these flows, there is usually no time in a production schedule for such meanderings. Instead, people take the expedient route and work around them. This AntiPattern is common in innovative design shops where proof of concept or prototype code rapidly moves into production. It is poor design, for several key reasons:

- Lava Flows are expensive to analyze, verify, and test. All such effort is expended entirely in vain and is an absolute waste. In practice, verification and testing are rarely possible.
- Lava Flow code can be expensive to load into memory, wasting important resources and impacting performance.
- As with many AntiPatterns, you lose many of the inherent advantages of an object-oriented design. In this case, you lose the ability to leverage modularization and reuse without further proliferating the Lava Flow blobs.

Symptoms

- Frequent, unjustifiable variables and code fragments in the system.
- Undocumented, complex, important-looking functions, classes, or segments that don't clearly relate to the system architecture.
- Very loose "evolving" system architecture.
- Whole blocks of commented-out code with no explanation or documentation.
- Lots of "in flux" or "to be replaced" code areas.
- Unused (dead) code.
- Unused, inexplicable, or obsolete interfaces in header files.

Typical Causes

- R&D code placed into production without thought toward configuration management.
- Uncontrolled distribution of unfinished code. Implementation of several trial approaches toward implementing some functionality.
- Code written by a single developer.
- Lack of configuration management or compliance with process-management policies.
- Lack of architecture, or nonarchitecture-driven development. This is especially prevalent with highly transient development teams.
- Iterative development process. Often the goals of the software project are unclear or change repeatedly. To cope with the changes, the project requires rework, backtracking, and prototype development efforts. In response to a demonstration deadline, there is a tendency to make hasty changes or to code on-the-fly in order to deal with immediate problems. The code is never cleaned up, leaving architectural consideration and documentation postponed indefinitely.
- Architectural scars. Sometimes architectural commitments are made during requirements analysis, only to be deemed obsolete after some amount of development. The system architecture may be reconfigured, but these in-line mistakes are seldom removed. It may not even be feasible to comment-out unnecessary code, especially in modern development environments where hundreds of individual files comprise the code of a system.

Known Exceptions

Small-scale, throwaway prototypes in an R&D environment are ideally suited for implementing the Lava Flow AntiPattern. It is essential to deliver rapidly and the result is not required to be sustainable.

Consequences

- If existing Lava Flow code is not removed, it can continue to proliferate as code is reused in other areas.
- If the process that leads to Lava Flow is not checked, there can be a geometric growth as successive developers, too rushed or intimidated to analyze the original flows, continue to produce new, secondary flows as they try to work around the original ones.
- As the flows compound and harden, it rapidly becomes impossible to document the code or understand its architecture enough to make improvements.

Refactored Solution

There is only one sure-fire way to prevent the Lava Flow AntiPattern — ensure that sound architecture precedes production code development. This architecture must be backed up by a configuration-management process that ensures architectural compliance and accommodates mission creep (changing requirements). If architectural consideration is short-changed up front, code is developed that is ultimately not a part of the target architecture, and is therefore redundant or dead code. Over time, dead code becomes problematic for analysis, testing, and revision.

In cases where Lava Flow exists already, the cure can be painful. An important principle is to avoid architecture changes during active

development. In particular, this applies to computational architecture — the software interfaces defining the systems-integration solution. Management can postpone development until a clear architecture has been defined and disseminated to developers. Defining the architecture may require one or more system-discovery activities. System discovery is required to locate the components that are really used and are necessary to the system. System discovery also identifies those lines of code that can be safely deleted. This activity is tedious; it can require the investigative skills of an experienced software detective. As suspected dead code is eliminated, bugs will be introduced. When this happens, resist the urge to immediately fix the symptoms without fully understanding the cause of the error. Studying the dependencies will help you better define the target architecture.

To avoid Lava Flow, it is important to establish system-level software interfaces that are stable, well defined, and clearly documented. In the long term, up-front investment in quality software interfaces can produce big dividends, especially compared to the cost of jackhammering away hardened globules of Lava Flow code. Tools such as source-code control systems (SCCS) assist in configuration management. An SCCS is bundled with most UNIX environments and provides a basic capability to record histories of updates to configuration-controlled files.

AntiPattern Name: Stovepipe System

The Stovepipe System AntiPattern concerns how the subsystems are coordinated within a single system.

General Form

The key problem in a Stovepipe System is the lack of common subsystem abstractions (see Figure 3).

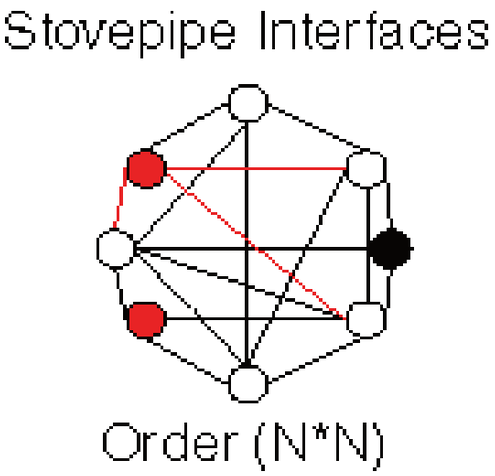


Figure 3

Subsystems are integrated in an ad hoc manner using multiple integration strategies and mechanisms. All subsystems are integrated point-to-point. The integration approach for each pair of subsystems is not easily leveraged toward the integration of other subsystems. The system implementation is brittle because there are many implicit dependencies upon system configuration, installation details, and system state. The system is difficult to extend since extensions add additional point-to-point integration links. As each new capability and change is integrated, system complexity increases. Complexity increases throughout the lifecycle of the Stovepipe system. System extension and maintenance become increasingly intractable.

Symptoms

- Large semantic gap between architecture documentation and implemented software. Documentation does not correspond to system implementation.
- Architects unfamiliar with key aspects of integration solution.
- Project is overbudget and has slipped its schedule for no obvious reason.

- Requirement changes are costly to implement. System maintenance generates surprising costs.
- System may comply with most paper requirements but does not meet user expectations.
- Users need to invent workarounds to cope with limitations of system.
- Complex system and client installation procedures that defy attempts to automate.

Typical Causes

Multiple infrastructure mechanisms used to integrate subsystems. Lack of a common mechanism makes the architecture difficult to describe and modify:

- Lack of abstraction. Each interface is unique to each subsystem.
- Insufficient use of metadata. Metadata is not available to support system extensions and reconfigurations without software changes.
- Tight coupling between implemented classes, requiring excessive client code that is service specific.
- Lack of architectural vision.

Known Exceptions

R&D software production will often use the Stovepipe System AntiPattern to achieve a rapid solution. This is acceptable for prototypes. Sometimes, lack of knowledge about a domain may require a Stovepipe System to be developed initially to gain domain knowledge either for building a more robust system or for evolving the initial system into an improved system (see "Big Ball of Mud," by Brian Foote and Joseph Yoder, *Proceedings of Pattern Languages of Programming*, 1997). Finally, sometimes the wish to use a vendor's product can lead an organization to conclude that the benefits outweigh the consequences of this AntiPattern.

Consequences

- Lack of interoperability with other systems. Inability to support integrated system management and intersystem security capabilities.
- Changes to the system become increasingly difficult.

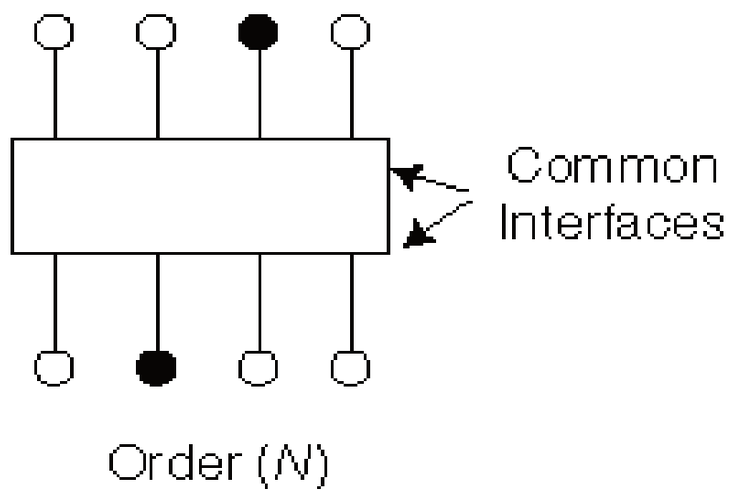


Figure 4

- System modifications become increasingly likely to cause serious new bugs.

Refactored Solution

The refactored solution is a component architecture that provides for flexible substitution of software modules (see Figure 4). Subsystems are modeled abstractly; so there are fewer exposed interfaces than there are subsystem implementations. The substitution can be both static (compile-time component replacement) and dynamic (run-time dynamic binding). The key to defining the component interfaces is to discover the appropriate abstractions. The subsystem abstractions will model the interoperability needs of the system without exposing unnecessary differences between subsystems and implementation-specific details.

To define a component architecture, decide upon a base level of functionality that the majority of applications should support. In general, the base level should be small and focus upon a single aspect of interoperability such as data interchange or conversion. Define a set of system interfaces that support this base level of functionality. Because most services will have an additional interface to express finer-grained functional needs, the component interface should be small.

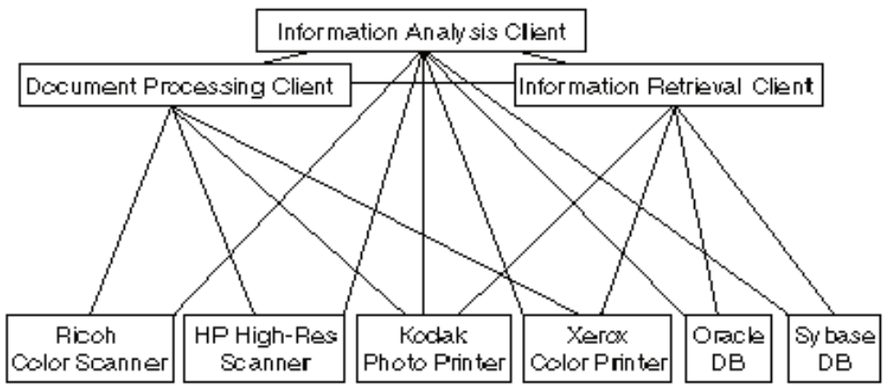


Figure 5

Having a base level of component services available to all clients in the domain encourages the development of thin clients that will work well with existing and future services without modification. Thin clients are those that do not require detailed knowledge of the services and architecture of the system; a framework may support and simplify their access to complex services. Having several plug-compatible implementations available increases the robustness of clients as they potentially have many options for fulfilling their service request.

Applications will have clients that are written to more specialized (vertical) interfaces. Vertical clients should remain unaffected by the addition of the new component interfaces. Clients that only require the base level of functionality can be written to the horizontal interfaces, which should be more stable and easily supported by other applications. The horizontal interface should hide all the lower-level details of a component via abstraction and should only provide the base-level functionality. The client should be written to handle whatever data types are indicated by the interface to support any future interchange of the horizontal component implementations.

Example

Figure 5 is a typical Stovepipe system. There are three client subsystems and six service subsystems. Every subsystem has a unique

software interface. Every subsystem instance is modeled as a class in the class diagram. When the system is constructed, there is unique interface software for each client corresponding to each of the integrated subsystems. If additional subsystems are added or substituted, the clients must be modified with additional code integrating new unique interfaces.

The refactored solution to this example considers the common abstractions between the subsystems (Figure 6). Since there are two services of each type, it is possible for each model to have one or more service interfaces in common. Then, each particular device or service can be wrapped to support the common interface abstraction. If additional devices are added to the system from these abstract subsystem categories, they can be integrated transparently to the existing system software.

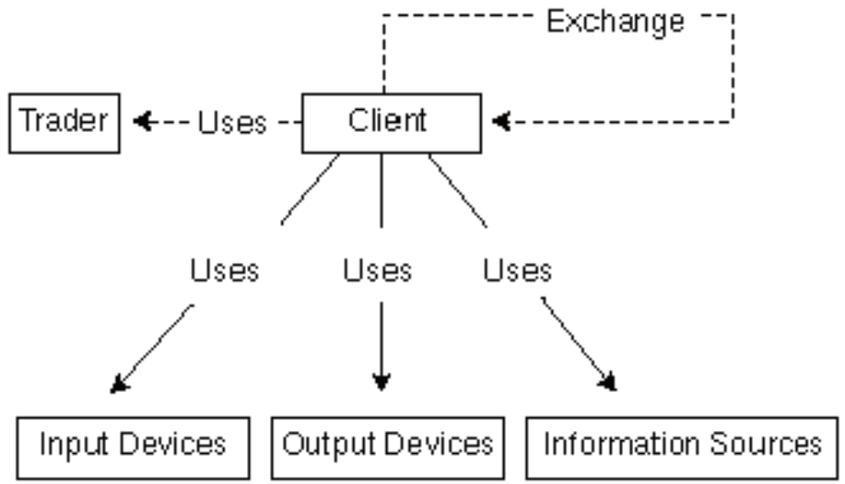


Figure 6

Conclusion

AntiPatterns provide guidance in identifying and resolving real-world problems in software development. Like design patterns, they are effective in capturing expertise by documenting commonly recurring software processes and designs. However, they go beyond current design pattern approaches in providing detailed assistance in recognizing when a solution exists. Additionally, AntiPatterns provide step-by-step guidance in changing an existing dysfunctional situation into a beneficial one that avoids the harmful impending consequences. The AntiPattern form has already appeared sporadically on the Internet, with informal AntiPatterns written by authors from the design-pattern community. Based on the Internet forums, the consensus is that AntiPatterns are a worthwhile research area. We hope this article and future work will be instrumental in promoting the AntiPatterns concept into the mainstream of the software development community.

Acknowledgments

Many thanks to our coauthors of *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Dr. Thomas Mowbray, chief scientist at Blueprint Technologies (mowbray@www.serve.com), and William J. Brown, process director for product development at Concept Five Technologies (brown@concept5.com), and John Wiley & Sons publishers.

— Hays was a lead engineer at MITRE. Raphael was chief scientist at Eidea Labs. They are coauthors of* AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis *(John Wiley & Sons, 1998).

Mailbag

Some readers are suspicious about the cloud, are not sure that F# is doomed for niches, and think acceptance tests are important.

In Response to "Head in the Clouds" (http://drdobbs.com/229300666)

In your editorial, you wrote : "Amazon and GAE both use highly distributed data stores, which have different architectures and distinctly different APIs. Microsoft offers modified versions of its SQL Server for a fee in its cloud instances."

While I do not believe anything to be untrue in the statement, it is perhaps unintentionally misleading. While Microsoft does offer SQL Azure — which is indeed a cloud version of SQL Server — they also offer a highly distributed data store, just like Amazon and GAE. This highly distributed data store — or NoSQL database, if you like — is called Windows Azure Table Storage and in fact pre-dates SQL Azure.

Bill Wilder
Founder/Leader of Boston Azure User Group

"Cloud computing is a direct result of Wall Street interjecting itself into the business [of] the tech industry. This is not speculation, I know for fact that Wall Street brokers went to the tech companies, especially and including Microsoft, and said that they needed to level out their income. Having the traditional tech peaks and valleys caused by the staggered product release cycles was not good for their companies and discouraged Wall Street investment. So over the years, the likes of Microsoft and others have been trying to devise a means to level out income. How many different Software-as-a-Service have we seen out of Microsoft? Those failed. Why? Because they invented that model for their needs, not mine. Cloud Computing is just the next version of that attempt.

Thus my first reaction to cloud computing is this is just another attempt to solve your business need, not mine. I have no problem with new technologies. If they make sense I'm there, as evidenced by my use of SL. Why is Cloud Computing so difficult? Because they are trying to invent a market, not solve a need.

Sandy P. Petrella
PS Group, Inc.

Andrew Binstock replies: "That's a unique take. I have to strongly disagree. Clouds, internal or external, solve several big problems: dynamic growth and contraction of project needs, administration of hardware, and they convert capex to opex (your main point). If I were to predict the clouds' potential, I think they will be adopted radically

and most hardware not at the client end point will be provided as a service, in much the way phone access is provided today."

In Response to "Acceptance Test Driven Development" (http://drdobbs.com/229219511)

In saying that we need acceptance testing, beyond unit testing, I feel you are comparing apples and oranges.

Unit testing is (usually) testing performed on an individual component in vacuo. Its key limitation is the need to simulate inputs from and outputs to other parts of the system…where the simulations may not faithfully or completely simulate system behavior or assess the adequacy of response. But it may have the advantage of observability and controllability far beyond what would be achievable in a complete system, and so may be far more effective in stressing the components and assessing their response.

System testing is (usually) whole system functionality/stress testing performed on a complete running system. The goal may be to get around the limitations of unit testing, but it is more commonly to see "how the whole product performs"…because so many key processes turn out to be the results of complex interactions among parallel and cascaded sub-systems.

Acceptance testing is testing against designated criteria, which may be a subset of unit and system test cases ... but should be performed on the bits-to-be-delivered as they will be installed and configured, in a way that is either highly representative of expected use or highly probative of key success criteria.

Any assertion that one of these is somehow categorically better than another seems naïve.

Mark Kampe

Andrew Binstock replies: " I agree that there is no inherent reason that acceptance tests are better or worse than unit tests. As you observe, they serve different purposes. But for TDD specifically, I think acceptance tests are the better test to use--for the reasons I stated.

In Response to Thither F# (http://drdobbs.com/windows/229300912)

I just saw your editorial on F# and some of the e-mail responses. I think both missed the mark a bit. I have used F# regularly for the last two years and I have been happy to see the growth of the community around it lately. It takes a long time to adopt a new language partly because it's hard to hire anyone with experience. I can't adopt a language unless I have a really compelling business reason. Visual Studio 2010 has only now made it widely available and easier for people to learn. Coupled with the existing functional capabilities in C# and greater attention in schools, I suspect in 3-5 years, the situation will be very different. For example, how long did it take for Java to reach any significant level of adoption aside from applets? I don't know the answer, just guessing that it's similar.

Jason McCampbell

Have a correction or a thoughtful opinion on Dobb's content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of Dr. Dobb's.

This Month on DrDobbs.com

The following are links to interesting items posted on www.drdobbs.com over the past month that you may have missed

HEAD IN THE CLOUDS

For developers, programming for the cloud, especially public clouds, is no simple task. Andrew Binstock explains how *DrDobb's* is going to help you tackle the cloud.

http://drdobbs.com/web-development/229300666

APPROXIMATE STRING MATCHES AND DYNAMIC PROBLEMS

Dennis Shasha's most recent "Tough Apps" column examines two classic algorithms as puzzles, showing that many problems have the property that they can be broken down into two or more subproblems whose solutions can be combined in some way.

http://drdobbs.com/tools/229218573

MANIPULATING THE DOM WITH JQUERY

One of the factors for the success of the jQuery library is its ability to work with the Document Object Model (DOM) in a way that is both powerful and easy to code. Dino Esposito discusses the ways in which jQuery is different from (and largely preferable over) the standard DOM API.

http://drdobbs.com/web-development/229300630

HIGH PERFORMANCE C++ CODE IN WINDOWS AZURE ROLES

If you have to migrate an existing high performance application to Windows Azure, you will probably have highly optimized C and C++ code that takes advantage of multicore hardware and SIMD instructions. A long time ago, Windows Azure Mix 09 CTP introduced the ability to run Web and/or Worker Roles in full trust. The newest Windows Azure versions allow you to make P/Invoke (short for Platform Invoke) calls to invoke native code.

http://drdobbs.com/go-parallel/blogs/high-performance-computing/229300715

TDD IS ABOUT DESIGN, NOT TESTING

TDD is not a way to more-thoroughly test code, although it might have that result

http://drdobbs.com/229218691

NETBEANS AND RUBY PART WAYS

Is the break overdue, or is it a symptom of Oracle's cavalier view of open-source projects?

http://drdobbs.com/open-source/229200212

IN THIS ISSUE

Dr. Dobb's

Jon Erickson Editor in Chief, Dr. Dobb's
jerickson@drdobbs.com

Andrew Binstock Executive Editor, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

CONTRIBUTING EDITORS

Mike Riley

Herb Sutter

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Group Sales, Kelly Glass
(212) 600-3327, kglass@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Western Regional Director, Matt Stovall
(415) 947-6245, mstovall@techweb.com

District Sales Manager, Rachel Calderon
(516) 562-5338, rcalderon@techweb.com

Account Manager, Alison Rubino
(415) 947-6248, arubino@techweb.com

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Sales Assistant, Ian Doyle
(415) 947-6105, idoyle@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Shoshana Freisinger
(415) 947-6349, sfreisinger@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

SALES CONTACTS—NATIONAL

Global CIO, InformationWeek 500 Conference

National Account Manager, Amy Neidlinger
(212) 600-3163, aneidlinger@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600 3106, sguttman@techweb.com

SALES CONTACTS—EVENTS

Director, Event Operations, Jennifer Russo
(516) 562-5094, jrusso@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Director of Marketing, Sherbrooke Balser
(949) 223-3605, sbalser@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Tony L. Uphoff CEO

John Dennehy CFO

David Michael CIO

Bob Evans Sr. VP and Global CIO Director

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz VP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing