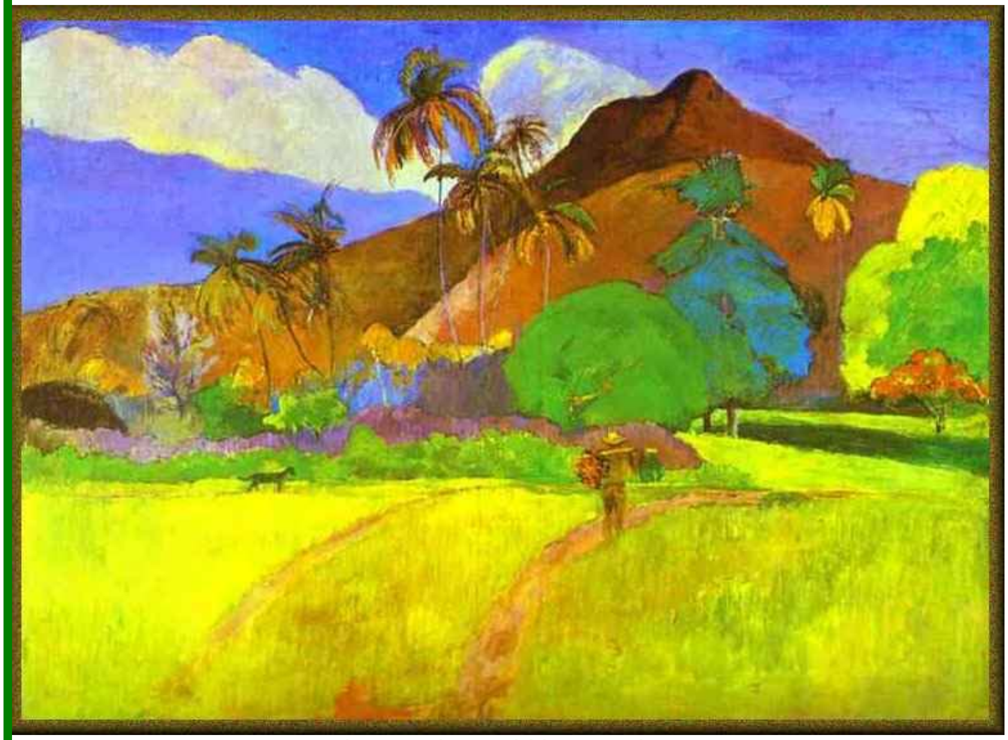
ObjectiveView

A Magazine for the Professional Software Developer


Paul Gauguin - Tahitian Landscape

published by

RATIO

www.ratio.co.uk
Development, Training and Consultancy

http://www.objectiveviewmagazine.com/
for back issues – see page 4 for listing

web distribution partner

ICONIX
Leaders in Object Technology

http://www.iconixsw.com/
Tel: + 1 (310) 474-8482
Fax: +1 (310) 474-8609
Email: umltraining@iconixsw.com

web distribution partner

ThoughtWorks®

http://www.thoughtworks.com/

web distribution partner

Software Reality

http://www.softwarereality.com/

web distribution partner

Spa

The BCS Software Practice Advancement SIG
http://bcs-spa.org/

web distribution partner

Ambysoft

http://www.ambysoft.com/

web distribution partner

jacoozi
>> thinking solutions

Information Technology services through lean software development

http://www.jacoozi.com/

web distribution partner

Dodeca
Technologies

http://www.dodeca.co.uk/

web distribution partner

OBTIVA

http://www.obtiva.com/

ObjectiveView

CONTENTS

CONTACTS

Editor
Mark Collins-Cope
mailto:oveditor@objectiveviewmagazine.com

Editorial Board
Scott Ambler
Kevlin Henney

Production Assistance
Adam Weremczuk

Free subscription
PDF by email – email to:
mailto:objectiveview@objectiveviewmagazine.com
(with subject: subscribe)

Feedback/ Comments /Article Submission /Letters
mailto:oveditor@objectiveviewmagazine.com

Circulation/Sponsorship Enquiries
mailto:oveditor@objectiveviewmagazine.com

Authors may be contacted through the editor. All questions or messages will be passed on.

Distribute ObjectiveView
It's easy. You link to ObjectiveView using our linkgif, and your logo will appear on *all* copies of the magazine. Contact oveditor@objectiveviewmagazine.com

web distribution partner
http://www.implementingscrum.com/

web distribution partner
http://www.agilemovement.it/

web distribution partner
http://www.codegeneration.net/

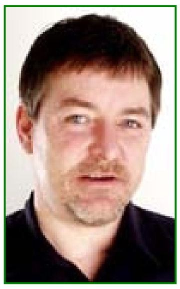
web distribution partner
Software Development Articles
http://www.softdevarticles.com/

web distribution partner
http://www.cpttm.org.mo/

web distribution partner
Methods & Tools
Software development free e-newsletter
http://www.methodsandtools.com/

web distribution partner
International Association of Software Architects
http://www.iasahome.org/

Welcome to Issue 10



Never let it be said that we only ram our own opinions down reader's throats. In issue 9 we covered Ruby/Rails in some detail - and gave the Ruby/Rails approach a positive editorial. In this issue Matt Stephens of SoftwareReality gives a counter opinion. Not that we agree with him of course.

I was pleased that Kent Beck agreed to an interview with ObjectiveView. One of themes that emerges in this is the lack of *civil* discussion about issues in software development. It's fine that discussion groups sometimes have passionate debate - that just shows that we care about what we do - but all to often discussions degenerate into hate mail between different camps. This really isn't on. A good moderator can help of course (though many groups are unmoderated), and my OV colleague Scott Ambler pointed me at the following set of discussion guidelines: http://www.agilemodeling.com/feedback.htm#Rules

Otherwise, issue 10 is something of a mixed bag of articles - the most common theme being databases, with automated testing coming second. Mike Tauty of Microsoft gives us the lowdown on future directions for C# - in particular planned features for C# 3.0 - including C# facilities for "language-native" relational database queries. Scott Ambler shows us that it is possible to refactor database applications - this being a common objection to an agile development approach. Word on the street is that many data professionals are resistant to agile development approaches. Which is a shame as they can clearly play an active role on such projects – see http://www.ambysoft.com/books/agileDatabaseTechniques.html for more information.

Taking a different direction entirely, Rick Grehan overviews the current state of play in Object Databases. On the subject of which, a recent benchmark (007) found that db4o (an Object database) was up to 55x faster than Hibernate/ PostgreSQL. That *is* interesting, but please read the full press release before jumping to conclusions (http://www.db4o.com/about/news/release/2006_09_28.aspx).

On the automated testing front, Kevin P. Taylor tells us why we should treat our test code with the same reverence we treat our core application code. This assumes, of course, that we treat core code well in the first place! Many developers, however, have to deal with a code base that has been cruelly neglected. To this end, Michael Feathers explains how adding automated tests to legacy code can help nurture it slowly back to health.

On the opinion front Kevlin Henney gives what I think is an excellent rationale for why Waterfall development is more prone to failure than iterative and incremental approaches. Not that all waterfall projects fail of course, but there is little doubt in my mind that the risks of failure are significantly higher. Other opinion includes Grady Booch tackling the hype surrounding Service Oriented Architectures (SOAs) and, as I mentioned, Matt Stephens discussing why he doesn't back Ruby.

Last, but not least, Doug Rosenberg discusses the model/code synchronisation features of the Enterprise Architect UML tool. More often than not, models are simply discarded once they have served their initial purpose – the cost of maintaining them is considered too high. On highly iterative / incremental projects this is a pain - once past the first iteration (barely the start of the project) it becomes more and more difficult to use modeling intelligently. Perhaps this solves the problem?

On the more general front, here are some readership stats. ObjectiveView is currently going from strength to strength. Issue 9 had about 40,000 hits and it's still clocking up 2,000 a month.

	Issue 9
March	22285
April	5382
May	3675
June	2048
July	2026
August	2094
September	2031
	39541

Downloads issues 9 broken down over the last 6 months. Figures do not include email subscribers.

[illegible]

Mark Collins-Cope,
London, October 2006.

Starting in issue 11 we will be publishing a selection of letters from readers. If you would like to comment on *any* aspect of software development, or perhaps just on an article, email oveditor@objectiveviewmagazine.com with your full contact details.

Back Issues

ObjectiveView Issue #9 - Newer Technologies Focus

- Alex Ruiz on AspectJ
- Richard Vaughan on AJAX
- Amy Hoy introduces Ruby
- Obie Fernandez - Ruby on Rails
- Rebecca Wirfs-Brock - Specs are Bad!
- Kevlin Henney - Abstraction - Down on the Upside
- Scott Ambler - Glacial Development :-)!
- Ken Pugh - Prefactor and be Agile

ObjectiveView Issue #8 - Agile Development Special

- Kent Tong on Turning Comments into Code
- Elfreide Dustin on Unit Testing
- Tim Mackinnon on Agile Project Retrospectives
- Scott Ambler with the latest update on Agile Model Driven Development
- Doug Rosenberg and Matt Stephens on Combining UML with TDD

ObjectiveView Issue #7 - Focus on .NET

- Microsoft C# author Jon Jagger overviews C#.NET
- Paul Hatcher takes a look at VB.NET
- Richard Vaughan on Managed C++ under .NET
- Interview with author Doug Rosenberg and Matt Stephens on their forthcoming book: XP Refactored.

ObjectiveView Issue #6 - Component Development

- John Daniels & John Cheesman (Authors: UML Components) on UML Components
- Paul Allen (Author: Realizing e-Business With Components) on EBiz Components
- Mark Collins-Cope & Hubert Matthews on Layered Architecture
- Philip Eskelin (Author: Component Design Patterns: A Pattern Language for Component Based Development),with Kyle Brown & Nat Pryce on Component Distribution Patterns

ObjectiveView Issue #5 – Focus on Use Cases

- Interview with Ivar Jacobson
- Clemens Syperski (Author: Component Software) on Components vs. Objects
- Ralph Johnson (Author: Design Patterns) on Dynamic Object Model Architectures
- Keiron McCammon on e-Business Architectures
- Doug Rosenberg (Author: Use Case Driven Modeling with UML) and Kendall Scott (Co-author: UML Distilled) with "Goldilocks and the Three Software Processes"

ObjectiveView Issue #4 – Focus on XML

- Richard Vaughan with an Introduction to XML for Developers
- Author Jason Garbis on Designing Distributed Object Applications
- Author Jan Bosch on Software Product Lines and Architectures
- Why is UML topsy turvy? by Hubert Matthews and Mark Collins-Cope
- Author Brian Hendersen-Sellers describes the OPEN Process
- In-depth technical interview with author Robert C. Martin on eXtreme Programming

ObjectiveView Issue #3 – Focus on XP

- Yonat Sharon summarises Kent Beck's Extreme Programming Process...
- Authors Kendal Scott and Doug Rosenberg put the counter-case to Extreme Programming...
- Paul Crerand of BEA with a detailed technical article on M3 - their Object Transaction Monitor
- Brent Carlson of IBM discusses the use of Design Patterns in SanFransiso
- Author Robert C. Martin with An Introduction to UML Use Cases...

ObjectiveView Issue #2

- Author Thomas Mowbray gives an Introduction to CORBA.
- Author Robert C. Martin on the Open-Closed principle of OO design.
- Anne Thomas (Patricia Seybold Group) on Noblenet Nouveau - ORB/COM/RPC interoperability tool.
- Keiran McCannon of Versant with an in depth article on the case for the OODBMS (vs. RDBMS)

ObjectiveView Issue #1

- Structuring Large OO Projects - Avoiding the Pitfalls
- Object Management Group Analysis by the UK's OMG Representative
- Object Oriented Design Tips

for back issues visit *http://www.objectiveviewmagazine.com/*

Spa 2007

Software Practice Advancement 2007

25 - 28 March 2007

Homerton College, Cambridge, UK

Technology... Practice... Process... People

SPA brings together experts and practitioners from around the world to exchange the latest ideas and skills in software architecture, design and development.

SPA provides a unique high-energy learning experience that explores a broad range of subjects from lead-edge technology, through pioneering software development and deployment practices, to innovative techniques for managing projects and the people that make up the project team.

The SPA Conference is run by the British Computer Society's Software Practice Advancement group. We have a passion for advancing the art of software development and will be presenting thought-provoking sessions which explore emerging practices that software teams can leverage in their project work.

Breaking news...

- Super Early Bird Discount now available until 30 Nov 2006
- Dave Thomas (Bedarra Corp) confirmed as keynote speaker
- Exciting new venue in Cambridge for 2007

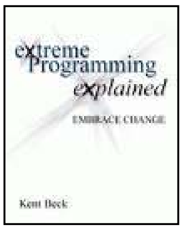
To book a place or find out more visit www.spaconference.org
or call +44 (0) 870 760 6863

BCS

http://bcs-spa.org/conferences.html

Interview with Kent Beck

***Mark Collins-Cope talks to eXtreme Programming creator** Kent Beck.*



XP

Mark: Thanks for agreeing to talk to ObjectiveView. Obviously no interview with Kent Beck would be complete without some discussion of XP. So let's start there. Where did the ideas behind XP come from?

Kent: *Many different sources: my experience with development, eclectic reading, and talking with other developers.*

"… there are many agendas at work in UML besides creating more value with software…"

Mark: Why did you add "respect" as a value to XP?

Kent: *The big shift between the first and second editions of XP Explained is in tone and perspective. The second edition acknowledges that many kinds of people need to be involved to create value with software. Each of these people have intrinsic worth as human beings, even as they contribute differently to the software.*

Acknowledging the worth of everyone makes for better work and better software. It's easier to say than practice for me, after most of a lifetime spent believing programmers were the chosen ones, but it works when I do it.

Mark: "XP matches observations of programmers in the wild" is a very amusing phrase – where did it come from? Is there a danger here that we pander to programmer wishes rather than customer needs? What differentiates work from pleasure is, after all, that work is doing something for someone else.

Kent: *I can't provide you with a source as I heard the phrase third-hand. The needs of the individual people on a project do affect the success of the project. Respecting each other's needs, roles and expertise allows for the maximum contribution from each participant. I don't think the distinction between work and pleasure is binary. I think what differentiates work from pleasure is my attitude toward the task at hand. I find some hard work satisfying. I get pleasure from working on JUnit, which is both work done for someone else and a form of service.*

"… civil conflict is in short supply in our profession …"

Mark: XP certainly became very popular in a fairly short period of time. Did you have a pro-active "marketing" campaign for it – perhaps including a series of must-do's before releasing ito to the wider world. Or was it just "the right thing at the right-time"?

Kent: *Timing is clearly a big part of any bonfire success, but I think there is a lot of philosophical and practical content in XP. I follow a similar process with any new idea—I try it myself, then I talk to a few people about it, then I talk about it publicly and pay attention to how larger groups of people use it. There was no marketing campaign. In fact, I just talked with people I met then wrote a book about what I had been saying in those conversations.*

Mark: When you visit projects using XP, how often are they using all the twelve practises, and if not all, which are most often dropped? Does dropping some practises cause problems?

Kent: *Every project does things differently. I think the most difficult practices to apply are those that require a change in personal beliefs. Which practices are difficult are different from person to person and from culture to culture.*

Mark: Is there one practise in particular that when omitted, causes most problems?

Kent: *I don't think that it is the omission or commission of practices that makes the biggest difference. I think the presence or absence of respect is the biggest differentiator on software projects. Unfortunately, many developers go their whole careers without accountability for acting with respect for others and themselves. With respect, you can work out the right practices together for the situation. Without it, pairing or not pairing, testing first or not testing don't really matter.*

Mark: If there was one thing you'd have done different with XP, what would it be?

Kent: *I would have involved my partner, Cynthia Andres, sooner.*

Mark: In your book XP Explained you say the cost of change curve has flattened out from the exponential rises reported many years ago when using a waterfall process. Do you really believe this – and if so, why?

Kent: *My experience is that the cost curve for adding functionality can be essentially flat. Defects are certainly much more costly to fix later rather than earlier. I think flattened cost curve for functionality results from a combination of better techniques, better tools, and more CPU cycles to run tests and automate refactoring.*

ObjectiveView Training
Application Architecture
1 or 2 days
Agile Project Management
1 day
Use Case Driven Object Modelling
(Iconix Process)
* Jumpstart (work on your project)
5 days
* Public Schedule *
2 days
C #
4 days
C ++
4 days
Ruby/Rails
3 days
AJAX
2 days
Java
4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

http://www.objectiveviewmagazine.com/

Mark: You also say that estimating, in the context of fixed price development, should be based on experience – of similar systems.

Kent: *Distributing risk in a business relationship is often a complicated negotiation. The problem I see with fixed-price/fixed-scope contracts is that they appear to shift the risk of non-performance to the supplier, but they don't in reality. If the supplier fails to deliver, the customer still doesn't have the software they need. I prefer the style of contracts used in the lean manufacturing world, where risks and rewards are explicitly shared. This gives both parties good economic reasons to work in their mutual best interest.*

Mark: Refactoring is obviously key to XP – but the need to refactor is important in any iterative and incremental project – would you agree?

Kent: *Almost all projects are iterative and incremental, looked at from the scale of decades. And yes, I think the ability to continually improve the design of a system based on experience is valuable. It reduces overall project costs, extends the working life of the system, and creates options for taking the system in new directions.*

Mark: Do you not think that by focusing on a bit more up front design we could reduce the amount of refactoring necessary in a particular increment?

Kent: *If you can correctly predict the eventual design of the system, then you can reduce the amount of refactoring. At one point in my career I explicitly chose to shorten my "design horizon" to two years, then one year, then one quarter, then a month, a week, a day, and finally the next test case. At each stage I developed with less stress, less over-engineering, better design, easier testing, and overall higher productivity. I discovered that my predictions were wrong enough of the time that designing incrementally was cheaper overall. Design time doesn't go away when working in this style, it just gets spread out across the project.*

Mark: So you've tried different "windows of design look-ahead" What do you see as the strengths and weaknesses of differing look-ahead periods? And how do you see the impact of window of look-ahead on, for example, the amount of refactoring required?

Kent: *There are issues in software development that require a long view: people's growth, customer relationships (including business models), organizational values, and in some cases technology. Other questions can be handled more effectively with a shorter view, because they change more quickly.*

Mark: Should "window of look-ahead" be linked to risk factors – e.g. new to a programming language = smaller window of look-ahead, new domain = smaller window of look-ahead, etc.

Kent:. *I think that people evaluate at all levels all the time. Decisions are made based on instincts that balance both long and short term perspectives. Each individual's mix is different and that is what makes concensus more difficult. It makes sense that in riskier choices should be reevaluated more frequently. The minute-by-minute rhythm of TDD, the hour-by-hour rhythm of pair programming, the daily rhythm of team development, the weekly rhythm of delivering new deployable functionality all provide a good basis for development. It is important that the discussion of the options not outweigh the value of those options. The discussion necessary to decide that* this *iteration should be two weeks instead of one can easily cost more than the gains realized by this micro-optimization.*

" … there is a software crisis …"

Mark: Historically speaking I think its true to say that testing was always the last activity an average developer wanted to do – and yet here we have an integral testing process that seems to have been taken on board by developers. How is that?

Kent: *I think the key was to find a style of testing that provides mutual benefit. As a programmer, I write tests partly because they contribute to my work. Also, I think a lot of testing was presented to programmers in a shaming way, "You know, if you were really conscientious you would test better."*

Mark: A lot of TDD tutorials say develop software one test at a time. Can this process not be optimised by developing software for multiple tests at one time?

Kent: *What often happens to me when I write multiple tests is that in making the first one work, I realize that the API is wrong and I have to go change all the tests. I start with an outline of all the test I want to write, but I actually write them and make them work one at a time. Writing multiple tests before making any of them work is a micro-optimization that leads to macro de-optimization.*

Mark: Should software be designed with testability in mind?

Kent: *Software should be designed to serve a number of purposes simultaneously: correct execution, future enhancement, ease of understanding, reuse, and testability. Improving coupling and cohesion as described 25 years ago by Yourdon and Constantine results in software that better serves all of these purposes.*

Mark: Extreme Programming Refactored - what did you think of this book? It's unusual for a book to be written against something, so why against XP?
Kent: *I didn't learn anything about software development from this book. You would need to ask the authors why they wrote it.*

Mark: Did you know that the origin of stand-up meetings is attributed to Queen Victoria and her meetings with the Privy Council in the UK – she apparently didn't like to hang around too long!

Kent: *I didn't know that ☺!*

Agile Development

Mark: Leaving aside XP, which of the agile approaches is your favourite and why?

Kent: *My favorite is the Toyota Production System as described by Taiichi Ohno in his book of the same name. I appreciate that he derives his methods from first principles and that he describes his techniques with vivid metaphors. He clearly separates the issues that need long-term planning, like capacity planning, from those that need daily planning, like planning tomorrow's production.*

Mark: Do you see a parallel with software in terms of planning in the appropriate detail/timeframe?

Kent: *Absolutely. Planning should work on the shortest possible timeframe that makes sense, and the organization should actively work to reduce the timeframes that make sense. For example, if customers won't accept a release more often than once a year, find out why and work to improve development and the overal relationship so the customers are ready for more-frequent changes.*

Mark: Agile development puts, quite rightly in my opinion, a strong emphasis on people. In the early days I was quite surprised to see many well-known independents who'd previously been writing on technical subjects suddenly turning their attention to man-management, people interaction and related topics. There's a wealth of general (non software development) literature and gurus out there who talk about getting the right team, empowering individuals, taking risks, etc. that can be applied to software development as much as any team based human endevour. Shouldn't software developers use these resources? And why did this emphasis arise - is our industry coming of age? Or is it that the well-known independents have come of age ☺?

"… The same topic on a list populated by geeks would end up in rancor and chaos. By the time it was over, someone would definitely call someone a Nazi…."

Kent: *I have the distinct advantage of having a partner whose background is in psychology. None of the psychological or sociological material programmers are talking about is new (and much of it is badly misinterpreted from the original work). As for the "well-known independents coming of age", I know that I am learning continuously.*

Mark*:* But *is* there a uniqueness about software development that means we in our industry should develop our own body of people knowledge here? Is there something special about the people side of it?

Kent: *I don't think that the nature of the work is socially unique. Programmers have historically started out with lower levels of social skills than some other professions, but I don't think that is because typing code into a machine requires that condition. In fact the younger generation of programmers seem to be much more in tune with their peers socially than my generation was. By working on people skills, programmers can improve their work interactions and thus their effectiveness. The myth that programmers are a "breed apart" has been used to justify belligerence, disrespect, and a lack of accountability. From my own experience, my life got better when I stopped acting like other people should treat me as if I was special and starting taking responsibility for myself.*

Mark: That feedback (on the software development process) is vital seems to me to be one of the key message of agile development. Would you agree with that?

Kent: *Yes. One school of thought says that to make the world safe you have to be able to predict perfectly. Part of the message of XP is that safety lies in getting good at listening and responding. Listening and responding are both skills that can be learned.*

Mark: How about feedback in terms of working software?

Kent: *I see two constraints on getting feedback from working software. Getting feedback from real use is extremely valuable, and worth restructuring a product plan to encourage. However, it is not an end in itself. The software should create some value for customers first. Then when they use it you have a chance to build a relationship. It takes careful listening and creative planning to find the kernel of value with a customer and deliver it quickly.*

UML and Modelling

Mark: It seems to me that some people like visual modelling, and some don't. Do you (personally) think visual modelling is useful? How do you think about software, if not visually?

Kent: *I think about software visually. I draw diagrams daily while programming. I wrote and sold a tool, the Object Explorer, that let people draw diagrams from running code, and transformed that into Spider for Eclipse. What I object to is diagrams being used from fear, as a way to avoid* feedback.

Mark: What *do* you think of UML?

Kent: *I think there are many agendas at work in UML besides creating more value with software.*

Long time in software development!

Mark: You've been developing software for quite a while now. Do you still get the same buzz out of it? In what ways has your approach to software development changed over the years? What, for you, is exciting at the moment in this field?

Kent: *About six months ago I had a great programming experience that reminded me how much I enjoy the act of programming. I have since been refocusing my business so I can spend more time just plain programming. One thing that I have changed is that I used to work very hard to get my ideas. I'm finally getting confidence in my ability to come up with ideas, so I don't get so agitated. I find that I get better ideas faster this way. What is exciting at the moment is the proliferation of new languages. Java seemed like it was going to be the new PL/I, and I suppose now it is. This has created space for new languages with interesting features.*

Mark: "The software crisis" is a term that pops up every now and then. Is there a software crisis? If so, what should we do about it? Do you think that problems in software development projects are often caused by customer's lack of understanding of what is going on – unless you are prepared to look at source code (or perhaps other models) the realities and constraints of software development are not exactly visible – unlike, say, building a building - where at least some of the rules of the game are clearly visible as building progresses.

Kent: *I think there is a software crisis, in that as a community we are so far under-performing compared to our potential. I lay responsibility for this squarely at the feet of the geeks. Until we offer the same level of accountability and responsibility as, for example, sales people, we have no right to expect to be treated as businesslike partners. Part of this is getting off the martyr/wizard pendulum and reaching out to people with different perspectives.*

Industry Fads or Ideas of Real Benefit

Mark: Our industry seems to home in on certain topics every so often, and these are - not infrequently - presented as the silver bullet that will solve all our problems. Here's a few that come to mind from the last 20 odd years: SOA, Agile, CBD, 4GLs, Structured Programming, OO programming, re-use, etc. How do you view these?

Kent: *Two things that are clear to me are that software development has made a lot of progress in the past half century and that it is nowhere near its potential. When we as a community try to pass responsibility off to others, we don't make progress. Some of the fads you mention seem that way to me. When we increase our accountability and transparency,add to our ability to build and maintain strong relationships with non-geeks, then we improve the state of our art.*

Mark: some of the debates that go on in our industry get quite vitriolic - and very negative. Take Ken Pugh's book, Prefactoring. Despite Ken apparently being on-side for agile development, some of the comments on Amazon were very negative and quite frankly nasty - apparently because Ken dared to say that it might be okay to think about code structure before writing it! How do you view these type of things. Should we be able to discuss these things without it getting personal?

Kent: *Ken is not the first person to receive nasty, personal comments about his writings. I think civil conflict is in short supply in our profession. Effective professionals in other fields are capable of thoughtful disagreement. I can think of no valid reason for us to behave any differently. We have used our genius/wizard status to excuse ourselves from many of the social graces for a long time. This attitude is not serving us well.*

A homeschooling mailing list my partner reads recently had a debate about creationism vs. evolution. What was remarkable to me was how clearly everyone spoke. They were confident in their own positions, stated them clearly, and listened carefully to other people's positions. The same topic on a list populated by geeks would end up in rancor and chaos. By the time it was over, someone would definitely call someone a Nazi. I think this is because we are not, by and large, comfortable with ourselves. We spend our time maintaining the illusion either that we are horrible or wonderful, and anyone who seems to threaten the illusion is met with aggression.

The experience of witnessing an impassioned-and-civil discussion and wishing for the same sort of discourse myself inspired me to give a talk entitled Ease at Work (http://www.agitar.com/downloads/20060516-webinar_-_lunch_with_kent_beck.html). I got emotional during the talk because I want that ease so much. The follow-up comments have been very interesting. For instance, Sarah Allen said she thought the pendulum (http://www.ultrasaurus.com/sarahblog/archives/000274.html) as a woman-only thing, because she'd only ever seen men in their heroic phase. I think that we have a lot of room for growth in the area of constructive rhetoric.

Mark: Regarding "we are horrible or wonderful" – why is that? Is is part of our collective personality? Is because our industry is young? Or why?

Kent: *I think that somewhere along the line we stopped listening to feedback. The "horrible/wonderful" pendulum is my own fear in the absence of knowledge of the reality of my situation. I think everyone has to find and maintain an accurate self-image. Geeks seem to have a hard time with this. We spend so much more time practicing relating to machines instead of people. The best way I know to get a balanced self image is to interact with a wide variety of people, but it's a conscious effort to do so.*

Mark: Kent, thank you very much for your time.
Kent: *You're welcome.*

Opinion • Matt Stephens • Ruby - I Love You (Not)

Issue #9 of ObjectiveView was a sort of Ruby Special: lots of articles devoted to the wonder that is Ruby, and Ruby on Rails. When a journal that is meant to take an objective view suddenly goes all misty-eyed on us, it's a warning signal that something might be up. Either Ruby really is the "wonder drug" of programming languages which will cure all our development ailments; or the Ruby hype machine has finally gone into overdrive.

But dig a little deeper and you'll quickly notice that it isn't being used on very many projects at all (as I'll discuss shortly). There's also a growing cynicism regarding Ruby's applicability on "serious" business projects. [1]

What's Good About Ruby?

First, the good things. Beneath all the hype [2], there's a lovely scripting language; and Ruby on Rails' almost rabid use of "sensible defaults" means that initially there's very little configuration to be done for your new web application. As long as your app falls inside the path most travelled, you'll find that, at first, you make lots of progress quite quickly. For many people this will be enough – especially for prospective converts evaluating Ruby.

Its syntax is nice, and the programs you end up writing tend to be quite clean and concise compared with their Java equivalent. Put simply, it's fun to write Ruby programs.

In Ruby, everything is an object, including numbers, Booleans, even nil. And you can extend existing objects at run-time. So it's perfectly legal to define a new message on 'Integer', say, and then call the message on a numeric literal, as in:

```
puts 5.factorial
```

(Pause for dramatic effect, while Java and C# programmers everywhere silently lay down their tools and realise how horribly unproductive they've been all this time).

But the jury – the objective jury, that is – is still out on whether Ruby is suitable for medium- or large-scale enterprise projects.

Ruby on Real Projects

With all the hype surrounding Ruby, it's noticeable that it hasn't exactly swept through the industry in the way that Java did a decade ago. It's swept through the blogosphere, but that doesn't equate to usage on real-world commercial projects.

A quick search on Jobserve.com in the UK revealed that 3,235 Java jobs had been posted in the last 7 days. And the number of Ruby positions that desperately needed filling? 26. [3]

In the City of London, investment banks and their ilk appear to have rediscovered Java, and currently the demand for Swing developers is at an unprecedented high. In the cosmic scheme of things that isn't surprising. Java is a mature language and platform; and desktop Java has come along in leaps and bounds. It's geared towards real-world pragmatism. This is why the language includes primitive types (which aren't objects). This design decision is sometimes derided by object purists, but it's a pragmatic concession to the real world: the need to get the job done, not to slow your whole system down in the name of design correctness.

Contrast this with Ruby's pure approach to OO, described earlier. The downside is performance (admittedly an issue that also detracted from Java in its early days). There are some benchmarks available, e.g. [4] shows that Java's "-server" VM is significantly faster (although Ruby consumes less memory).

Limitations

A pretty major limitation – and it's a difficult one to solve for any dynamic language – is that Ruby lacks a decent IDE, with modern time-savers such as code completion and built-in API documentation. I've seen Ruby die-hards argue that you don't need an IDE with good refactoring support (etc), because you'll write less code and it'll be purer. But frankly, that just doesn't wash. Big projects with groups of developers working together need a strong IDE, plain and simple.

Ruby has patchy Unicode support at best (it's especially problematic with Ruby on Rails [5]). This makes Rails suitable for websites about a programmer's pet cat, but not for high-end sites with stringent I18N requirements.

The need to invent a new language to overcome the (fixable) shortcomings of a mature, thriving platform like Java is like demolishing your house because you don't like the style of wallpaper. When replacing anything, it's worth asking: what problem am I solving? Is it worth the effort? When Java came along, there was no language as strong. There was C++ but it didn't have all the APIs, the memory management, the networking etc. Now, after many years in relative obscurity, Ruby hops onto the hype radar, and Java is 10+ years strong, solid and trusted. Where's the benefit for all the hassle of changing language?

Another issue that I see with Ruby – and it's the same uncomfortable danger that hangs over the extreme programming/emergent design crowd – is that Ruby's syntactical cleverness encourages programmers to write "clever" code: code that's more about itself and about technical minutiae and dynamic design patterns than about the business problems that it's meant to be addressing.

Ruby proponents seem to measure maintainability in program size: the smaller the program, the "cleaner" and more maintainable it is. If maintainability was really achieved by making the code more compact, we would all be writing our enterprise systems in Perl.

You Just Can't Get the Staff

Very much a "chicken and the egg" problem is that the critical mass of developers doesn't exist in the marketplace. If you're choosing a programming language/platform for your next project, you'll want to go with something for which you'll have no problem finding experienced staff. Unfortunately, that eliminates Ruby from the contest straightaway.

For all its purported simplicity and elegance, the language requires advanced OO knowledge. If there are no Ruby programmers available, you'll have to train some up: so their first exposure to this mysterious new scripting language could be trying to debug the in-house guru's closures, mixins, loop abstractions, and dynamic dispatch constructs. The commonality of such advanced features in the language scores highly against it when you're recruiting and can't find enough coders with prior Ruby experience.

Conclusion

Despite the hype, Ruby faces some serious issues if it's to be a genuine contender in the enterprise space. It's 10x slower than Java, lacks the libraries, the Unicode support and doesn't (yet) have a decent IDE.
Note, that's the current state: who's to say how far Ruby will go in the near future? Sun's adoption of the core JRuby developers [6] is an interesting development; it's entirely possible that Ruby will find a suitable niche as a scripting language inside the JVM; following in Rhino's footsteps (sorry, hoof-prints). That would be nice.
Whatever happens, I can see Sun incorporating many of Ruby's best features into Java, and extending the JVM to allow for dynamic scripting.

If we start to see the same mature, thriving, extensive ecosystem of libraries, platforms and experienced developers (not to mention Unicode support and a decent IDE) for Ruby as we have both for Java and the .NET world, then there might be a reason to switch. Until then though, Java continues to give developers what they ask for and need on real business projects.

References

Ruby on Rails: http://www.rubyonrails.org/

[1] http://www.redmonk.com/cote/archives/2006/03/mcgovern_comes.html
[2] http://www.ebcvg.com/press.php?id=1761
[3] http://www.jobserve.com/searchresults.aspx?jobType=*&d=7&q=java
http://www.jobserve.com/searchresults.aspx?jobType=*&d=7&q=Ruby
[4] http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=ruby&lang2=java
[5] http://www.ebcvg.com/press.php?id=1761
[6] http://headius.blogspot.com/2006/09/jruby-steps-into-sun.html
http://jruby.codehaus.org/

Matt Stephens is a programmer/team leader in Central London. He co-authored Extreme Programming Refactored: The Case Against XP (Apress, 2003) with Doug Rosenberg; and Agile Development with the ICONIX Process (Apress, 2005) with Doug Rosenberg and Mark Collins-Cope. Catch Matt on-line at www.softwarereality.com

C# 2.0 and 3.0

Mike Tauty ***overviews the features of C# 2.0, and gives us a peek at what's coming in version 3.0.***

A number of significant enhancements appear in the second version of the C# programming language that was released alongside Visual Studio 2005 in November of last year. This was the first major revision to the C# language in a short history which has as its other major milestones the first release of Visual Studio .NET in 2002 and preview releases stretching back to the year 2000.

At last year's Professional Developer's Conference in Los Angeles, the C# language was the centre of attention again as possible future enhancements were previewed including new capabilities that integrate data manipulation capabilities into the language.

This article provides a brief overview of the new features that C# version 2.0 introduced before moving on to look at some of the new features that are found in the current preview of the version 3.0 language.

C# Version 2.0

The current version of the C# programming language is a mixture of evolution and revolution with a number of additions that span from relatively simple enhancements to the syntax of the language such as *static classes* through to new capabilities such as *generic types* that exist at both the language level and the underlying .NET Common Language Runtime level.

This section provides an overview of the main new features in the language and provides comparison with the previous version and illustrations of usage.

Static Classes

It is common practise to define classes that have no instance methods. Consider the .NET Framework class **Math** which provides core mathematics functionality through a series of static methods such as **Math.Max, Math.Truncate**, etc.

In the first version of the C# language, there was no formal mechanism for indicating to the consumer of a class such as **Math** that the class did not need to be instantiated for use but, rather, was purely a collection of static methods. In order to aid the user of a class such as **Math**, the class author would typically mark the default constructor for the class as **private** as below in order to prohibit erroneous attempts to construct an instance of the **Math** class:

```
public class Math
{
    private Math()
    {
    }
    public static double Round(double d)
    {
        ...
    }
}
```

In version 2.0 of the C# language, this requirement for **static** classes which cannot be instantiated is formalised in the language through the addition of the keyword **static** as below:

```
public static class Math
{
    public static double Round(double d)
    {
                   ...
    }
}
```

The provision for **static** classes in the language (and in metadata) assists tools and the compiler enforces the **static** nature of the class by emitting errors if the author of the class attempts to add instance methods or state to the class or if the consumer of the class attempts to construct an instance.

Partial Classes

Unlike the C++ language, in version 1.0 of C# the definition of a class must reside in a single source file. The relationship of class definitions to source files is many-to-one.

This restriction is removed in version 2.0 of the language and a class may now be defined in any number of source files as long as those source files are made available to the compiler at the same time (that is, it is not possible to compile part of a class in one compilation and then compile additional parts of the class at a later point in time).

The language introduces a new keyword, **partial,** as illustrated in the figure below to indicate that the class being compiled may also have constituent pieces in other source files where the definition must also be marked as **partial**.

```
partial class MyClass
{
    public int myInteger;
}
partial class MyClass
{
    public int myOtherInteger;
}
```

One particularly compelling use for **partial** classes is in techniques that use code generation. As an example, in the case of user interface forms design, the visual aspects of the form may be generated as one part of a class definition whilst the second part of that definition may be manually written by the developer. This technique provides for elegant separation of the sections of code that are automatically generated by a tool from those that are "hand-crafted".

Generic Types

The major addition to the .NET Common Language Runtime for version 2.0 was the addition of the ability to use build and consume *generic* types. Generics have a certain degree of familiarity to the C++ programmer in that, at first glance, the technology appears similar to what is offered by C++ templates. However, generics differ greatly in their capability and implementation.

As a way of introducing generics, consider the typical implementation of a simple data structure such as a **Stack** written with a version 1.0 .NET language such as C#;

```
public class Stack
{
    public void Push(object o)
    {
    }
    public object Pop()
    {
        return (null);
    }
    private object[] storage;
}
```

In this sketched implementation of a Stack, the developer has written a single implementation that takes advantage of the .NET type system's ability to treat all data types as being derived from **System.Object**. Consequently, this implementation can be used to provide a general purpose stack that can store any kind of data – e.g. "Stack of float", "Stack of integer", "Stack of Customer" and so on as illustrated in the code fragment below which stores floating point values using this class;

```
Stack s = new Stack();
s.Push(100.0f);
float f = (float)s.Pop();
```

Whilst this technique is both valid and common amongst existing .NET code, there are two potential disadvantages to this type of implementation.

The first and most obvious problem is that of type safety. As the Stack implementation deals with all types as **System.Object** it is not possible to instantiate a Stack which only stores a particular data type. Consider the fragment below which attempts to construct a stack which only stores integers;

```
static void Main(string[] args)
{
    Stack stackOnlyForIntegers = new Stack();
    stackOnlyForIntegers.Push(10);
    CallMethod(stackOnlyForIntegers);
    int i = (int)stackOnlyForIntegers.Pop(); // may throw
}

static void CallMethod(Stack stackOnlyForIntegers)
{
    stackOnlyForIntegers.Push("This is a string, sorry");
}
```

Here, despite the caller's intention, the callee takes advantage of the lack of type-safety and misuses the Stack passed as a parameter in order to add a **String**.

The caller has to mitigate against the possibility that their Stack may not always contain data of the right data type casting the return value from the **Pop** method and that cast may fail causing a run time exception.

This lack of type-safety is particularly problematic in today's component-oriented development environment where an application may be constructed from many different components which are potentially sourced from different vendors and accorded different levels of trust within the application's runtime environment.
Beyond the issue of type-safety, there is also a performance penalty in treating all members of the .NET type system as the superclass **System.Object** and this performance penalty is inherent in the design of the type system. Consider the following innocuous looking lines of code:

```
int myInteger = 10;
object myObject = myInteger;
int myOtherInteger = (int)myObject;
```

In this fragment, the value of the Integer variable **myInteger** is assigned to a variable of type **System.Object**. In the .NET type system, an Integer is a *value-type* and is not strictly derived from the **System.Object** type. Consequently, the language compiler performs work to insert instructions which perform an operation known as *boxing* the integer. In short, a real **System.Object** is allocated from the .NET managed heap and both the value and the run-time type of the integer are copied into that heap location which is then assigned to the variable **myObject**. Similarly, when the subsequent assignment is made to the variable **myOtherInteger** a process known as *unboxing* occurs which involves the type-checking and copying of the value held in **myObject** into the integer variable **myOtherInteger.**

In version 1.0 of the language, there was little that a developer who wanted to build a general purpose class like the example Stack could do to work around these limitations of type-safety and performance. Version 2.0 of the language, with its capabilities for generic types, provides a much better solution.

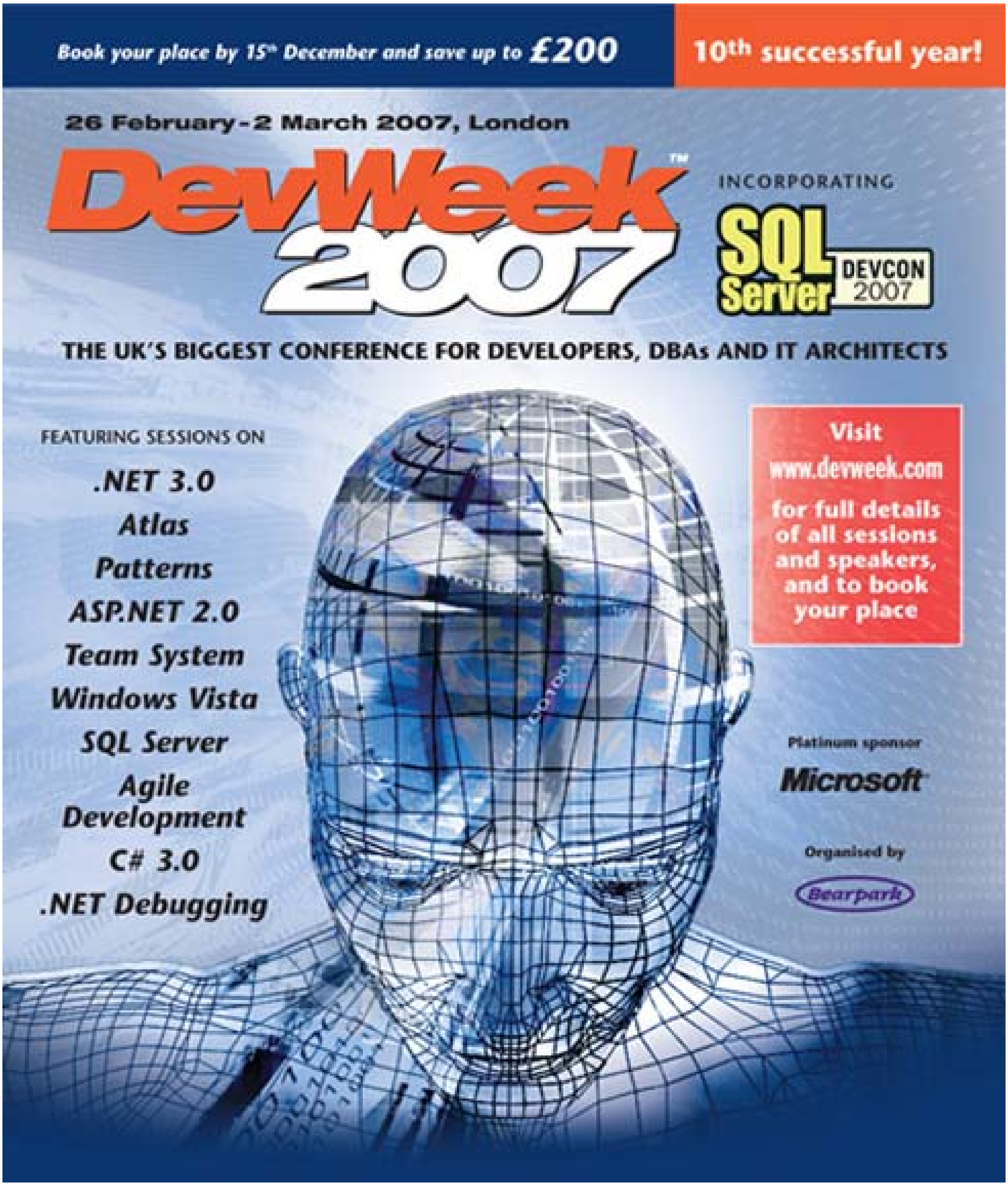
Consider, the following sketch of a C# 2.0 generic version of a Stack that has been *parameterised*;

```
public class Stack<SOMETYPE>
{
    public void Push(SOMETYPE o)
    {
    }
    public SOMETYPE Pop()
    {
      return (default(SOMETYPE));
    }

    private SOMETYPE[] storage;
}
```

In this revised version of Stack, generic code has been written around an (as yet) unspecified type referred to by a parameter, "SOMETYPE". When the Stack class is instantiated, the compiler requires a real data type to be provided in place of the "SOMETYPE" parameter as in;

```
Stack<int> intStack = new Stack<int>();
Stack<float> floatStack = new Stack<float>();
Stack<string> stringStack = new Stack<string>();
```


Book your place by 15th December and save up to £200
10th successful year!
26 February - 2 March 2007, London
DevWeek™ 2007
INCORPORATING
SQL Server DEVCON 2007
THE UK'S BIGGEST CONFERENCE FOR DEVELOPERS, DBAs AND IT ARCHITECTS
FEATURING SESSIONS ON
.NET 3.0
Atlas
Patterns
ASP.NET 2.0
Team System
Windows Vista
SQL Server
Agile Development
C# 3.0
.NET Debugging
Visit www.devweek.com for full details of all sessions and speakers, and to book your place
Platinum sponsor
Microsoft
Organised by
Bearpark

http://www.devweek.com/

Logically, the compiler replaces “SOMETYPE” with the data type provided such that the **Stack<int>** instantiation of the generic type is type-safe in that its version of the **Push** method will be defined as taking a parameter of type Integer and its version of the **Pop** method will return an Integer. The boxing/unboxing problem has also gone away as **Stack<int>** will store an array of Integers directly rather than converting and storing them an array of **System.Object**.

In reality, the compiler takes the generic data type Stack<SOMETYPE> and, unlike in C++ templates, compiles this into Common Intermediate Language as a generic class – that is, the Common Intermediate Language that .NET languages compile to represents Stack<SOMETYPE> as a generic type with corresponding metadata. At compilation time, the compiler can also check that specialisations such as **Stack<int>**, **Stack<float>** are correctly used.

At run time, as types such as **Stack<int>** are first instantiated, the Common Language Runtime takes the “template” provided by the generic Stack<SOMETYPE> and produces a version of that class which is specific to **Integer**. This process is repeated for all instantiations that involve parameters from the set of value-types (i.e. Integer, Decimal, DateTime, etc) but the process is only performed once for parameters from the set of reference-types where a performance gain is achieved by effectively specialising the generic type only for the true base class, **System.Object**, of all such types.

It is not only classes that can be constructed generically. Generics feature in many other aspects of the .NET type system allowing for generic classes, methods, properties, fields, delegates, events and interfaces such as the one illustrated by the following interface which shows how an interface for a generic dictionary lookup might be represented;

```
public interface IKeyValueLookup<KEYTYPE, VALUETYPE>
{
    VALUETYPE LookupKey(KEYTYPE key);
}
```

Given that generics provide type-safety, there is an unanswered question as to how generic implementations are actually constructed. For instance, consider the following generic method;

```
public static class Factory<T>
{
    public static T Create()
    {
        return (new T());
    }
}
```

The class above is not valid and will not compile because the compiler does not have enough information about the parameterised type **T** to allow the call that is made inside the **Create** method to the default constructor for **T.** If the compiler is to build the class **Factory<T>** at compilation time with type-safety then it needs to be provided with a guarantee that all types used as the parameter **T** will have a default constructor.

The C# language names these guarantees *constraints* and, for this specific example, **T** can be constrained to be from the set of types which provide a default constructor using the following code:

```
public static class Factory<T> where T : new()
{
    public static T Create()
    {
        return (new T());
    }
}
```

The C# compiler now accepts this definition for **Factory<T>** and, furthermore, it will ensure that any instantiations of **Factory<T>** satisfy the constraint that **T** is a type with a default constructor.

Any number of additional constraints can be specified for parameterised types and a type can be constrained in a number of different ways including constraining the type of be a value/reference type, constraining derivation from a particular base class or implementation of a particular interface.

As an example, consider a superfluous generic comparison method such as;

```
static int CompareTo<T>(T arg1, T arg2) where T : IComparable
{
    return (arg1.CompareTo(arg2));
}
```

One remaining point around generics is the compiler’s ability to infer generic types in certain situations. As an example, consider the following generic method signature;

```
static T Max<T>(T t1, T t2) where T : IComparable
```

when making calls to the **Max** method, it is possible to explicitly specify the parameterised type or it is also possible to omit that type making for less verbose, more clear code as in;

```
int i = Max(10, 20);      // type argument inferred
int j = Max<int>(20, 30);
```

Nullable Types

As has already been stated in this article, the .NET type system is clearly partitioned into value-types and reference-types.

Value-types such as Integers directly store their value whereas a reference-type is a type-safe pointer to a location on the .NET managed heap which contains the value. It is possible to have a reference that does not currently point to a valid location on the managed heap and in that case, the reference stores the special value of **null.** It is not possible to have “null” value types such as Integers.

This inability of the .NET type system to naturally model a “Nullable” value type can cause some friction when it is being used to manipulate the two most common data storage models of the day, namely relational databases and XML storage. Both of these models do present the

ObjectiveView Training
Application Architecture
1 or 2 days
Agile Project Management
1 day
Use Case Driven Object Modelling
(Iconix Process)
* Jumpstart (work on your project)
5 days
* Public Schedule *
2 days
C #
4 days
C ++
4 days
Ruby/Rails
3 days
AJAX
2 days
Java
4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

http://www.objectiveviewmagazine.com/

ability to model simple data types such as Integers which permit null values.

With the addition of .NET generic types into the Common Language Runtime, it is not too difficult to sketch a class **Nullable<T>** where T is constrained to be a value type which would facilitate working with nullable value-types. A sketch of **Nullable** is given below;

```
public class Nullable<T> where T : struct
{
    public Nullable()
    {
        isNull = true;
    }
    public bool HasValue
    {
        get
        {
            return (!isNull);
        }
    }

    // etc...

    private T value;
    private bool isNull;
}
```

The .NET Framework V2.0 includes such as class **Nullable<T>** which can be generally used to write code to deal with situations where nullable value-types are needed. Usage of the class **Nullable<T>** is illustrated below;

```
Nullable<int> i = null;
Nullable<int> j = 10 + i; // j is null

int k = j.Value; // Exception
int l = (int)j; // Exeption

int m = 10;

if (j.HasValue) // Explicit test
{
    m = (int)j;
}
```

Support for **Nullable<T>** is present throughout the .NET languages including C#, Visual Basic and C++ but the C# language has additional syntax to make working with **Nullable<T>** more convenient. In C#, the syntax for **Nullable<T>** can be shortened to **T?** and additional short-cuts exist as illustrated below:

```
int? i = null;
int? j = 10 + i; // j is null

int k = j.Value; // Exception
int l = (int)j; // Exeption

int m = j ?? 10;
```

It should be noted that whilst **Nullable<T>** and the short-hand syntax are primarly language features making use of the generic type system there is additional Common Language Runtime support in version 2.0 to ensure that scenarios such as the one below function as expected in that a nullable value type that is “boxed” and “unboxed” maintains its null status;

```
int? i = null;
object o = i;
int j = (int)o; // Exception as expected.
```

Anonymous Methods

.NET introduced the idea of a type-safe function pointer known as a *delegate* and makes use of it in order to formalise the observer pattern for monitoring the state of an object through the *event* system.

As an illustration, in .NET code it is possible to define methods such as **Print** and **PrintMore** below;

```
static void Print()
{
    Console.WriteLine("Hello");
}
static void PrintMore()
{
    Console.WriteLine("World");
}
```

and then define a *delegate* type with a signature that matches these methods;

```
delegate void Fn();
```

and, finally, to instantiate an instance of that newly defined delegate type and use that to refer to both the **Print** method and the **PrintMore** method (chained together in that order);

```
Fn f = new Fn(Print);
f += PrintMore;
f();
```

The invocation of the delegate **f()** causes the invocation of the **Print** method followed by the **PrintMore** method. Delegates can refer to both static methods as illustrated here or to instance methods of particular objects whereby the delegate then carries with it an implicit *this* pointer. As previously mentioned, the delegate forms the basis of the notion of an *event* in .NET where one object publishes an event and other code *consumes* or *handles* that event by adding appropriate delegates to the event’s list. A brief example is given below;

```
class Car
{
    public event EventHandler Started;
}
class Program
{
    static void Main(string[] args)
    {
        Car c = new Car();
        c.Started += new EventHandler(OnCarStarted);
    }
    static void OnCarStarted(object sender, EventArgs e)
    {
        Console.WriteLine("Car has started");
    }
}
```

Where **EventHandler** is a standard delegate signature that is used or extended by convention for .NET events. With version 2.0 of the C# language, this capability of taking a variable of *delegate* type is extended in that, firstly, the compiler has the capability to infer and generate an instance of the right delegate type which shortens the previous **Main** function to;

```
static void Main(string[] args)
{
    Car c = new Car();
    c.Started += OnCarStarted;
}
```

Note that whilst the creation of the **EventHandler** delegate is no longer specified, the C# compiler generates an instance of **EventHandler** implicity by calling its constructor with the **OnCarStarted** method as was done manually in the earlier code.

The compiler also has the ability to remove from the developer the burden of having to create a separate method to be referenced from their delegate variable by using the **delegate** keyword as illustrated below to create an *anonymous method*;

```
static void Main(string[] args)
{
    Car c = new Car();

    c.Started += delegate
    {
        Console.WriteLine("car has started");
    };
}
```

In this sample, the **OnCarStarted** method has been replaced with an in-line declaration using the anonymous method syntax indicated by the **delegate** keyword. This is logically equivalent to the previous code in that the compiler will (at the least) generate a separate method containing the code within the **delegate** block and will then create an instance (in this case) of the **EventHandler** delegate type and reference the anonymous method created from that instance.
Within the anonymous method definition, the arguments that would have been passed to the event handling method are still available to the developer using a slightly more explicit syntax as below;

```
static void Main(string[] args)
{
    Car c = new Car();

    c.Started += delegate(object sender, EventArgs e)
    {
        Console.WriteLine("car has started");
        Console.WriteLine("Event sent by {0}", sender);
    };
}
```

And, thus, no power of expression is lost by choosing this short-hand form of the definition of the delegate function over the longer variant.

The compiler can perform more code-generation around anonymous methods than simply generating the method itself. Consider the following code;

```
static void Main(string[] args)
{
    Car car = new Car();
    AddEventHandler(car);
}
static void AddEventHandler(Car car)
{
    int x = 10;

    car.Started += delegate
    {
        Console.WriteLine("Car Started");
        Console.WriteLine("Value of x is {0}", x);
    };
}
```

In this example, the **AddEventHandler** method is used to add an anonymous method to the **Started** event on an instance of the **Car** class. The anonymous method that is added contains code which prints out the value of the local variable **x** when the event is fired. Note that it is almost certain that the stack frame containing the local variable **x** will have been collapsed at the time that the event is fired and, yet, nonetheless the code will succeed. In this case, the C# compiler is performing more work to ensure that it *captures* the local state upon which the anonymous method depends and makes it available at the time that the method executes. An examination of the Common Intermediate Language code that is generated by the compiler reveals that, for these scenarios, the compiler generates a class with members to capture the state required and a method on that class to contain the code to be executed.

Note that, where the local state is represented by a value-type then the value can be expected not to have changed by the time the anonymous method is invoked. However, for a reference type it is the reference that is copied which opens the possibility that the underlying object will have changed by the time the anonymous method is invoked.
Anonymous methods offer interesting possibilities for simplifying code such as the following short example which sorts a list of integers in reverse order;

```
List<int> list = new List<int>();
list.Add(10);
list.Add(20);
list.Add(30);

list.Sort(delegate(int v1, int v2))
{
    return (v2 - v1);
);
```

Iterators

A standard pattern for enumeration is built into the .NET Framework through the **IEnumerable** and **IEnumerator** interfaces which are usually used in the C# language through the **foreach** statement as illustrated below;

```
foreach (object element in list)
{
}
```

The C# compiler looks for the **IEnumerable** interface on the **list** type and, if found, can call the **IEnumerable** method, **GetEnumerator** which it does to produce code similar to;

```
IEnumerator enumerator = list.GetEnumerator();
while (enumerator.MoveNext())
{
   object o = enumerator.Current;
}
```

Whilst this pattern is highly convenient for consuming enumerable types, it places a burden onto the developer of such types with the need to implement the **IEnumerable** and **IEnumerator** interfaces each time such an enumerable class is written. The **IEnumerator** interface involves implementing a small state-machine which stores and advances/reverses the current position of the enumeration as required.

The version 2.0 C# compiler has additional capabilities for the generating of enumerable types using a code generation technique named *iterators* to greatly reduce and simplify the amount of code written. The technique involves building methods that return one of the variants of the **IEnumerable/IEnumerator** interfaces and use the new **yield** syntax.

As a starting example, consider the following method;

```
static IEnumerable Strings
{
    get
    {
        yield return "One";
        yield return "Two";
        yield return "Three";
    }
}
static void Main()
{
    foreach (string s in Strings)
    {
    }
}
```

The **yield return** statement indicates to the compiler that the property in question is builidng an *iterator* for which the compiler requires that the property/method in question must return one of the **IEnumerable/IEnumerator** interfaces. The version 2.0 compiler can use the code in such an *iterator* property as **Strings** to generate a class which implements the **IEnumerator** interface and can correctly maintain the necessary state to navigate through the list of strings.

Whilst the previous snippet is a relatively simple example, *iterators* are not restricted to enumerating static collections but can be used to implement more complex enumerations. Consider the sketched example below which shows how a class might offer any number of iterator-based enumerations and how these can be consumed by callers;

```
public class SimpleList<T>
{
    public IEnumerable<T>Backwards
    {
        get
        {
            for (int i = list.Count-1; i >=0 ; i--)
            {
                yield return list[i];
            }
        }
    }
    private List<T>list;
}

public static void Main()
{
    SimpleList<int> list = new SimpleList<int>();
    foreach (int i in list.Backwards)
    {
    }
}
```

The full power of iterators comes to light in scenarios where the building of an implementation of **IEnumerable/IEnumerator** is non-trivial such as in the exposing of a more complex data-structure such as a tree. The recursive nature of tree-traversal does not easily lend itself to the **IEnumerator** interface but the compiler's *iterator* feature greatly simplifies scenarios such as tree-traversal as the following code sketch illustrates:

```
class TreeNode<T> : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator()
    {
        if (this.Left != null)
        {
            foreach (T data in this.Left)
            {
                yield return data;
            }
        }

        yield return this.Data;

        if (this.Right != null)
        {
            foreach (T data in this.Right)
            {
                yield return data;
            }
        }
    }
    public TreeNode<T> Left ...
    public TreeNode<T> Right ...
    public T Data ...
}
```

Whilst the full implementation details are not provided for the sample above, it is hopefully clear that the *iterator* approach to tree enumeration is greatly simplified versus the effort involved in building an implementation of **IEnumerable** that would correctly model the state needed to navigate a tree in a pre-order, in-order or post-order manner.

C# Version 3.0

At the Professional Developer's Conference in 2005, Microsoft made available early preview versions of the C# 3.0 compiler containing a new set of language features that combine to form the basis of a powerful new mechanism for extending the language to integrate data from varied sources.

The features are built on top of the current version 2.0 Common Language Runtime and are implemented in the C# compiler rather than the underlying runtime. Collectively, the feature-set is identified by the name LINQ which stands for Language Interated Query.
This article's coverage of the version 3.0 features will attempt to build up from the relatively simple new language features to present the constituent pieces which come together to enable LINQ.

It is important to realise that the version 3.0 C# language is currently in preview and, consequently, features may change before the technology is made generally available.

Object Initialisers

Consider a simple class such as Person below;

```
class Person
{
    public string FirstName
    {
        get ...
        set ...
    }
    public string LastName
    {
      get ...
      set ...
    }
}
```

Regardless of the constructors that the type has available, the prototype C# version 3.0 compiler allows the developer to initialise an instance of the class using an object initialiser as below:

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Person()
        { FirstName="Mike", LastName="Taulty"} ;
    }
}
```

In this instance, the compiler emits a call to the default constructor for the **Person** type followed by the calls to the two property setters for the **FirstName/LastName** properties. That is, the compiler is simply shortening the manual coding process that the developer would follow in the absence of an appropriate constructor which provided parameters for setting the **FirstName/LastName** values.

Collection Initialisers

In current versions of the C# language, it is legal to initialise arrays with syntax such as:

```
int[] integers = { 10, 20, 30 };
```

The C# V3.0 compiler takes this a stage further and extends that initialisation syntax to anything that implements **IList** which includes data types such as the generic **List<T>** allowing for syntax such as;

```
List<int> l = new List<int>() { 10, 20, 30 };
```

Once again, the compiler is automating a task that could easily be performed manually by the developer with more lines of code. In this instance, the compiler generates the explicit calls to **IList.Add** for each of the entries in the initialisation list.

Implicit Typing

Initially, the implicit typing feature of the prototype version 3.0 C# compiler looks to be one of the more controversial additions to the language. Below is a snippet that illustrates its use;

```
static void Main(string[] args)
{
    var v = 10;
}
```

In the fragment above, a variable **v** is declared without any specific data type being specified for the variable. The compiler *infers* the type of the variable from the right hand side of the assignment.

It is important to realise that in no way is the variable **v** loosely or dynamically typed. The variable **v** is typed as an Integer which the compiler infers from value being assigned to the variable. Consequently, the following code would fail to compile due to trying to assign a floating point value into the variable which has already been typed as Integer;

```
static void Main(string[] args)
{
    var v = 10;
    v = 1.0f;
}
```

This mechanism for declaring variables works only for a function's local (stack-based) variables. That is, class variables cannot be defined with the **var** keyword and nor can parameters to methods. This does not mean that variables defined with **var** cannot be passed to methods as in the fragment below:

```
static void Main(string[] args)
{
    var v = 10;
    SomeMethod(ref v);
}
static void SomeMethod(ref int i)
{
    i = 20;
}
```

With this set of limitations on implictly typed variables and with the arguable amount of obfuscation that such declarations add to a code-base it is difficult to put together a rationale for their use until they are matched with the new version 3.0 feature of Anonymous Types discussed in the next section.

Anonymous Types

Where version 2.0 of the C# compiler introduced the idea of the Anonymous Method, version 3.0 takes that further by introducing the appropriately named *anonymous type* which represents a complex data type that is used without first defining a class to represent it.

Consider the example below which ties together the previous sections on Implict Typing with that on Object Initialisers and introduces an *anonymous type*;

```
static void Main(string[] args)
{
    var v = new { FirstName= "Mike", LastName= "Taulty" };

    Console.WriteLine(v.FirstName);
    Console.WriteLine(v.LastName);
}
```

In the fragment above, an instance of a data type with two properties both of type **System.String** is generated. This data type does not have a name visible to the developer but the compiler can use the object initialiser given and infer types from it in order to build a data type to appropriately represent the data structure.

Because the data type does not have a name, it is impossible to declare a variable (or parameter or member) of that data type and, consequently, the variable used to store the value is defined implicitly with the compiler inferring the right type from the assignment. Thus, implicit typing comes into its own when working with anonymous types.

This type of declaration also works for arrays of anonymous data types which can be initialised in a similar manner as below;

```
static void Main(string[] args)
{
    var v = new[] {
        new {FirstName = "Mike", LastName = "Taulty" },
        new {FirstName = "Fred", LastName = "Smith" }
    };

    foreach (var item in v)
    {
        Console.WriteLine(item.FirstName);
        Console.WriteLine(item.LastName);
    }
}
```

Note that type-safety is preserved in all cases and only the data type becomes anonymous. In a list declaration such as in the previous fragment, data types must remain consistent across all members of the list in order for the compiler to generate the anonymous data type.

Extension Methods

C# version 3.0 supports the idea of extending a class by offering the illusion of appending new methods to any class at compilation time without modifying the definition of that class.

As an illustration, consider the simple class Point below with simple X and Y members;

```
class Point
{
    public int X;
    public int Y;
}
```

With the version 3.0 compiler, it is possible to give the appearance of adding methods to **Point** by writing an extension class as below (note the use of the **this** keyword on the method **Draw** to mark it as an extension method to the type **Point** and derived types);

```
static class PointExtension
{
    public static void Draw(this Point p)
    {
      // Take action to draw P.
    }
}
```

At the point where the compiler comes across an invocation to a method on the class **Point** that the class or its base-classes do not implement, the compiler is prepared to search for methods that extend **Point.** In essence, the compiler searches *all in-scope namespaces* in an attempt to locate a matching method such as **Draw** on the class **PointExtension** which extends the class Point. This allows for code such as;

```
static void Main(string[] args)
{
    Point p = new Point();
    p.Draw();
}
```

Note that if two or more matching **Draw** extension methods exist in the in-scope namespaces then the compiler will fail.

Notice also that in the presence of two such methods the developer can influence which is chosen simply by altering the in-scope namespaces to select one or the other.

Consider a more wide-reaching (if largely pointless) extension method such as the one below which can be applied to all types as it extends the ultimate base class, **System.Object;**

```
static class ObjectExtension
{
    public static string ToStringTwice(this object o)
    {
        return(o.ToString() + o.ToString());
    }
}
```

And corresponding code to make use of that extension method on a class such as the previous **Point;**

```
static void Main(string[] args)
{
    Point p = new Point();
    p.ToStringTwice();
}
```

Extension methods provide a powerful paradigm through manipulation of the in-scope namespace to control the methods that will be used to provide extensions to pre-existing classes.

Lambda Expressions and Lambda Statements

In their simplest form, Lambda expressions and statements provide an alternate syntax to the existing delegate syntax that exists in the C# language today. As a simple example, consider the following use of delegate syntax, using the anonymous method additions from the version 2.0 language;

```
delegate int AddDelegate(int x, int y);

static void Main(string[] args)
{
    AddDelegate d = delegate(int p, int q) { return p + q; };
    int x = d(20, 30);
}
```

The equivalent Lambda expression would be written as;

```
delegate int AddDelegate(int x, int y);

static void Main(string[] args)
{
    AddDelegate d = (int p, int q) => p + q;
    int x = d(20, 30);
}
```

Where the Lambda expression of **"(int p, int q) => p + q"** replaces the anonymous method code written previously. Similarly, a more complex anonymous method consisting of a block of statements such as the one below;

```
delegate void PrintFn(string s);

static void Main(string[] args)
{
    PrintFn f = delegate(string s)
    {
        Console.WriteLine(s);
        Console.WriteLine(s);
    };
}
```

Has an equivalent Lambda statement in:

```
PrintFn f = (string s) => {
    Console.WriteLine(s);
    Console.WriteLine(s);
};
```

Whilst Lambda expressions provide a neat alternate syntax to anonymous methods, their real power lies in the compiler's ability to perform more type inference than is performed for anonymous methods. For instance, whilst the compiler is prepared to infer type parameters for a Lambda expression that omits them such as:

```
delegate int AddFn(int x, int y);

static void Main(string[] args)
{
    AddFn f = (x,y) => x + y;
}
```

Such inference is not performed on the equivalent anonymous method syntax as below (which gives a compilation error);

```
delegate int AddFn(int x, int y);

static void Main(string[] args)
{
    AddFn f = delegate(x, y) { return(x+y); };
}
```

This capability of the compiler to infer parameter types and return types for Lambda expressions and statements becomes very important when combined with anonymous types as will be discussed in the following section.

The Basis of Language Integrated Query (LINQ)

Whilst some of the individual pieces of the version 3.0 C# language appear to be incremental changes when looked at in isolation, it is very illuminating to examine how these pieces can be used together to build up very powerful and flexible capabilities in the language. This section attempts to introduce such capabilities by putting together the previous version 3.0 language features.

Consider the following code-snippet using implicit typing to make reference to an array of anonymous data types;

```
var v = new[] {
    new { FirstName="Fred", LastName="Jones", Age=55 },
    new { FirstName="Bill", LastName="Smith", Age=66 }
};
```

Now, in the presence of a simple, generic delegate type which models any method with a single parameter and a non-void return value:

```
delegate U Func<U, T>(T t);
```

it is possible to sketch a generic method which, as an example, takes an array of some type and transforms it into some other type using a supplied conversion routine as:

```
static T[] Convert<T, U>
(U[] originalArray, Func<U, T> convertFn)
{
    T[] newArray = new T[originalArray.Length];
    for (int i = 0; i < originalArray.Length; i++)
    {
        newArray[i] = convertFn(originalArray[i]);
    }
    return (newArray);
}
```

Now, this routine can be extemely useful when combined with the compiler's inference capabilities for Lambda expressions. Consider code which takes the original list and uses this routine to transform it as below;

```
var v = new[] {
    new { FirstName="Fred", LastName="Jones", Age=55 },
    new { FirstName="Bill", LastName="Smith", Age=66 }
};

var w = Convert(v,x => new
   { ForeName = x.FirstName, HowOld = x.Age }
);
```

Taking these two lines separately:

```
var v = new[] {
    new { FirstName="Fred", LastName="Jones", Age=55 },
    new { FirstName="Bill", LastName="Smith", Age=66 }
};
```

In this first line, a variable **v** is declared with implicit typing causing the compiler to infer the type from the assignment. The right hand side of the assignment is initialising an array with elements of an anonymous type. The compiler can create the anonymous type as a tuple {FirstName, LastName, Age} of data types {string, string, int}. The second line of code:

```
var w = Convert(v,x => new
   { ForeName = x.FirstName, HowOld = x.Age }
);
```

makes a call to a **Convert** function passing a lambda expression. The compiler has visibility of the generic **Convert** function with signature:

```
static T[] Convert<T, U>
    (U[] originalArray, Func<U, T> convertFn)
```

The compiler can infer from the call the type of the first generic parameter **U[]** as being the array of anonymous types, **v.** It can then determine the argument type for the lambda expression (**x**). The lambda expression provided returns a new anonymous type which the compiler can construct and substitute as the type parameter **T** in the generic function call.

The result of this call is an array that can be enumerated using:

```
foreach (var entry in w)
{
    Console.WriteLine(entry.ForeName);
    Console.WriteLine(entry.HowOld);
}
```

More than a single line of code can be passed as a parameter to the **Convert** routine by making use of a Lambda statement such as:

```
var w = Convert(v,x => {
    string concat = string.Format(
    "{0} {1}", x.FirstName + x.LastName);

    return(new { FullName = concat, x.Age});
    }
);
foreach (var entry in w)
{
    Console.WriteLine(entry.FullName);
    Console.WriteLine(entry.Age);
}
```

The result of this new **Convert** function call is an anonymous type of form **{ FullName, Age }** where **FullName** is constructed from the existing anonymous type that is passed to the routine as a parameter.
With the use of *extension methods* described earlier in this document it is possible to give the appearance that the **Convert** method exists as a member of a particular data type as in:

```
static class Extensions
{
    public static T[] Convert<T, U>(this U[] originalArray,
    Func<U, T> convertFn)
    {
        T[] newArray = new T[originalArray.Length];

        for (int i = 0; i < originalArray.Length; i++)
        {
            newArray[i] = convertFn(originalArray[i]);
        }
        return (newArray);
    }
}
```

Used by code such as:

```
var v = new[] {
    new { FirstName="Fred", LastName="Jones", Age=55 },
    new { FirstName="Bill", LastName="Smith", Age=66 }
};

var w = v.Convert(
    x => { string concat = string.Format(
        "{0} {1}", x.FirstName + x.LastName);
        return(new { FullName = concat, x.Age});
    }
);
```

In the current preview of the LINQ technology, a number of extension methods which provide for common data access routines similar in nature to the **Convert** routine built above have already been defined and form the basis of Language Integrated Query. These extension methods are as below (taken from the C# 3.0 Specification);

```
delegate R Func<T1,R>(T1 arg1);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
class C
{
    public C<T> Cast<T>();
}
class C<T>
{
    public C<T> Where(Func<T,bool> predicate);
    public C<U> Select<U>(Func<T,U> selector);
    public C<U> SelectMany<U>(Func<T,C<U>> selector);
    public C<V> Join<U,K,V>(C<U> inner, Func<T,K>
        outerKeySelector,Func<U,K>innerKeySelector,
        Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K>
        outerKeySelector,
        Func<U,K>innerKeySelector, Func<T,C<U>,V>
        resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);
        public O<T> OrderByDescending<K>
            (Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
        public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E>elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);
    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}
```

```
class G<K,T> : C<T>
{
    public K Key { get; }
}
```

These extension methods can be used to provide advanced data querying facilities across different types of data from the C# language. Consider this example which takes an array of data and logically performs a query on that data: “Select Name, Country, Sales from data where Quantity > 5”:

```
var data = new[] {
    new { Country="UK",Name="Bob Smith",
        Quantity=10, UnitPrice=20.5m },
    new { Country="UK", Name="Jim Jones",
        Quantity=2, UnitPrice=10.0m },
    new { Country="UK", Name="Jack Williams",
        Quantity=5, UnitPrice=5.75m },
    new { Country="USA", Name="Chuck Jackson ",
        Quantity=8, UnitPrice=18.2m },
    new { Country="USA", Name="Art Arthouse",
        Quantity=6, UnitPrice=9.5m }
};

var meetingCriteria = data.Where(y => y.Quantity > 5);

var selection = meetingCriteria.Select(
  s => {
    decimal d = s.Quantity * s.UnitPrice;
    return new { s.Name, s.Country, Sales = d};
   }
);

var ordered = selection.OrderBy(o => o.Sales);

foreach (var v in ordered)
{
   Console.WriteLine("{0} {1} {2}",
        v.Name, v.Country, v.Sales);
}
```

Whilst the calls to the **Where, Select** and **OrderBy** methods could all be combined into a single, more complicated line of code the C# version 3.0 language takes a large step further by defining new keywords that map to these extensions methods. Consequently, the previous code can be rewritten to form a much more readable example as:

```
var data = new[] {
    new { Country="UK", Name="Bob Smith",
        Quantity=10, UnitPrice=20.5m },
    new { Country="UK", Name="Jim Jones",
        Quantity=2, UnitPrice=10.0m },
    new { Country="UK", Name="Jack Williams",
        Quantity=5, UnitPrice=5.75m },
    new { Country="USA", Name="Chuck Jackson ",
        Quantity=8, UnitPrice=18.2m },
    new { Country="USA", Name="Art Arthouse",
        Quantity=6, UnitPrice=9.5m }
};

var ordered =
    from d in data
    where d.Quantity > 5
    orderby d.Quantity * d.UnitPrice
    select new { d.Name, d.Country,
        Sales=d.Quantity * d.UnitPrice };

foreach (var v in ordered)
{
    Console.WriteLine("{0} {1} {2}",
        v.Name, v.Country, v.Sales);
}
```

It is important to realise that the **where, orderby, select** keywords here map directly to the extension methods previously discussed and the range of additions to the C# language discussed in this article come together to facilitate these data-like additions to the language.

It is also important to note that any set of implementations of the extension methods **Select** et al can be brought into use simply by including the appropriate namespace containing that set of extension methods and, thus, whilst the **select, where, orderby, groupby** keywords form a fixed set of new language keywords, their meaning can be altered by the introduction of a new namespace to the compilation unit using the keywords.

Work is ongoing at the time of writing in producing different sets of extension methods which take the common keywords and map them across object data as presented here but also against XML data (XLINQ) and relational data (DLINQ).

As an example of the power of expressivity offered by LINQ, consider the following short XML piece of XML data:

```
<?xml version="1.0" encoding="utf-8" ?>
<books>
    <book title="Hard Times" author="Dickens" price="5.99"/>
    <book title="Great Expectations" author="Dickens"
        price="7.99">
        <publishers>
            <publisher>Penguin</publisher>
            <publisher>Faber</publisher>
        </publishers>
    </book>
    <book title="The Pickwick Papers"
        author="Dickens" price="4.99"/>
</books>
```

XLINQ takes the set of query extensions already discussed in this document and applies them in a natural way to XML data without the need for the programmer to concentrate on specific XML programm interfaces. A simple example of the power of XLINQ for querying XML is given below where the query produces a list of {Title,Author,Price} from the XML document for books with a price higher than 5.0:

```
XElement element = XElement.Load(@"w:\temp\books.xml");

var result =
    from c in element.Descendants("book")
    where (decimal)c.Attribute("price") > 5.0m
    select new {
        Title = (string)c.Attribute("title"),
        Author = (string)c.Attribute("author"),
        Price = (decimal)c.Attribute("price")
    };

foreach (var entry in result)
{
    Console.WriteLine("Book {0}, {1}, {2}",
        entry.Title, entry.Author, entry.Price);
}
```

The power of LINQ is exhibited by the consistent way in which both object data and XML data is manipulated and, whilst not illustrated in this article, DLINQ further widens this consistent pattern to include relational data sources.

Conclusion

At the point where code becomes *boilerplate,* work can be done in the language or the factoring of the code to increase developer productivity. C# version 2.0 takes a number of *boilerplate* code tasks such as the generation of event handlers or the writing of enumerators and moves it into the compiler where it belongs. It also, through the surfacing of the Common Language Runtime's new generic data types, adds a whole new level of expressive power above the version 1.0 language. The current preview of Version 3.0 of the language includes additions that initially appear limited in scope such as object and collection initialisation and implicit typing for local variables. In isolation, these are useful programming constructs but it is only when combined with the power of Lambda expressions, anonymous types, compiler inference and extension methods that the full power of expressivity that the language offers begins to shine.

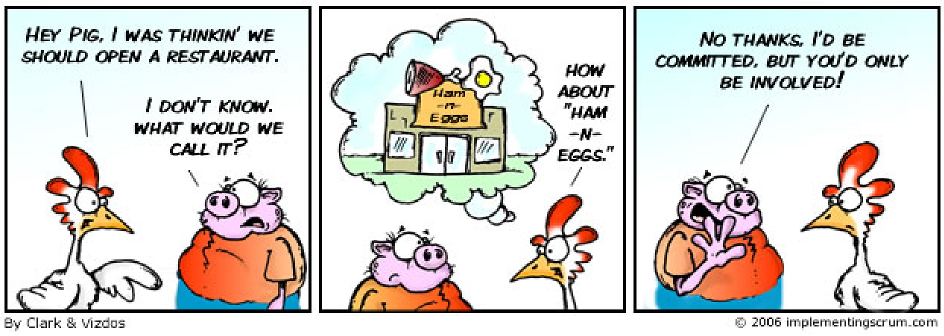
As has been illustrated, the great promise of the LINQ pattern and its current specialisations of DLINQ and XLINQ is to provide a consistent framework that narrows the *gap* that the C# programmer faces today between the constructs of the language in which they work and the different mechanisms that they use to manipulate data, whether that be through objects, hierarchical or relational paradigms.

References

See: The C# Specification V2.0, The C# Specification V3.0, The LINQ Project.

For an **introductory article to C# (V1.0)** see issue 7 of ObjectiveView at http://www.objectiveviewmagazine.com/

Mike Taulty is a member of the Developer & Platform Group at Microsoft in the UK. Mike has spent the past five years working for Microsoft and the previous ten years working as a professional software developer on a number of different platforms. You can contact Mike at http://mtaulty.com.


HEY PIG, I WAS THINKIN' WE SHOULD OPEN A RESTAURANT.
I DON'T KNOW. WHAT WOULD WE CALL IT?
Ham -n- Eggs
HOW ABOUT "HAM -N- EGGS."
NO THANKS, I'D BE COMMITTED, BUT YOU'D ONLY BE INVOLVED!
By Clark & Vizdos
© 2006 implementingscrum.com

http://www.implementingscrum.com/

Conference Report • BCS SPA Conference 2006

It's tempting, when writing articles on conferences, to try and use the conference name in novel and appealing ways…

… for example, this particular conference used to be run by the now renamed BCS OOPS (object oriented programming) group. That made it easy of course, "OOPS - I forgot to tell you that..." and other such puns would roll off the keyboard with ease - even if they were pretty unimaginative. BCS SPA, on the other hand, presents more of a challenge. But let's have a go...

Dictionary.co.uk defines SPA as: “a town where water comes out of the ground and people come to drink it or lie in it because they think it will improve their health: - Baden Baden in Germany and Bath in Britain are two of Europe's famous spa towns.” Not much to go on there!

Let's try some words that start with **SPA**:

- **SPA**nish. SPA is an international conference attended by delegates from many countries, including Spain!
- **SPA**cious. The residential (SPA conferences are always residential) conference facility was certainly spacious, the rooms were good and there was plenty of space to break out into groups, discuss topics, etc. Plus a nice chill out room with Internet access.
- **SPA**rk - Yep. There were lots of bright sparks at the conference, indeed networking and talking to other industry experts is one of the main reasons you might want to attend next year's conference.

Okay, enough frivolity for now. On to the serious business of explaining what the SPA conferences are all about. SPA is the British Computer Society's special interest group on *Software Practice Advancement*. The group in non-profit making, and that helps keep the conference price in the very reasonable bracket. The unique thing about SPA is that all the sessions are interactive. This means - yes really - that you do actually have a chance to learn something. I don't know about you - but sitting in front of endless unidirectional (presenter to attendee) presentations at most conferences is not the most exciting of propositions. SPA is different - you get to take part in the sessions, to ask probing questions as the session continues, and to test your understanding with the provided exercises or in work groups. In short attending a SPA conference is more like attending a set of highly interactive tutorials than attending a typical conference.

On to the topics covered. There were a wide range of topics covered - from a hands-on introduction to Ruby and Rails through Working with Legacy Code, Security Requirements and Patterns, Agile Process Metrics, on to Aspects and more common topics related to Java, etc. The biggest challenge I faced was deciding which of the multiple streams to attend - there was always more than one of interest.

A couple of sessions caught my eye to the degree I've included articles about them in this issue of ObjectiveView: *Mike Tauty* of Microsoft gave an excellent session on *C# 2.0*, and more interestingly *C# 3.0* features, Michael Feathers of ObjectMentor hosted a group on Working with Legacy Code. This is such a neglected real-life topic - presumably because it's not very sexy hacking someone else's pile of junk code - that reading Michael's book should be made compulsory! Two very informative sessions.

Other notable sessions (for me at least) included "Towards a precise business language for model driven development" run by Robert James and Christian Nentwich. At this session they showed work in progress on an eclipse plug-in that enabled structured English to be used to specify business rules and constraints in a manner that was checked against an object model. You know the sort of thing: "a customer may only have one account in debit at any particular time." Richard Mitchell is a leading world expert on modeling, and his session on modeling with views was good.

Dave Thomas - of Pragmatic Programmer's fame - gave an excellent keynote on "Angry Monkeys". This is such a great story I'm going to tell it to you now.

A team of researchers gathered a small group of monkeys in a room with a ladder in it. They hung bananas at the top of the ladder and surprise, surprise, monkeys being - well - monkeys, one of them climbed up the ladder to get them. As he did so, the researchers hosed down the other monkeys with water. This exercise was repeated until the monkeys learned not to go up the ladders.

Now comes the interesting bit. The researchers stopped hosing, and began to replace the monkeys one by one. The new monkeys, of course, tried to climb up the ladder, much to the consternation of the existing monkey who proceeded to jump on the newbee and beat the shit out of him. The researchers continued to replace the monkeys until none of the original hosed monkeys were left.

But still the new monkeys were jumped on when they tried to climb the ladder - despite the others never having experienced being hosed down. When interviewed, the monkeys were heard to say: "well, that's just the way do things around here!" (ok, they didn't really say that - poetic license please). Hmm… sounds familiar…

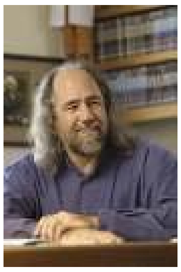
Particular thanks are due to Rachel Davies (Agile Experience) the Conference Chair and Jane Chandler (University of Portsmouth) the Programme Chair, and to Andy Moorley (Truedata Computer Services) for admin. The Conference Executive were: John Daniels (Syntropy Limited), Matt Stephenson (Royal & Sun Alliance), Helen Sharp (The Open University), and Eoin Woods (UBS Investment Bank).

Planning for next year's conference (25-28 March 2007 at Homerton College in Cambridge) is already well under way. If you're serious about software development, you really should attend. It's genuinely rare to get such direct access to the caliber of world-leading experts who attend this conference.

Be there or be **SPA**re ...

To find out more about BCS SPA 2007 – visit http://bcs-spa.org/conferences.html

Opinion • Grady Booch • Service Oriented Architectures



I not so long ago returned from some work with the SEI in Pittsburgh and then in Washington, DC where I conducted a number of customer visits primarily focusing on service oriented architecture.

Comments about hunting with Dick go over really, really well with the DC crowd.

My take on the whole SOA scene is a bit edgier than most that I've seen. Too much of the press about SOA makes it look like it's the best thing since punched cards. SOA will apparently not only transform your organization and make you more agile and innovative, but your teenagers will start talking to you and you'll become a better lover. Or a better shot if your name happens to be Dick. Furthermore, if you follow many of these pitches, it appears that you can do so with hardly any pain: just scrape your existing assets, plant services here, there, and younder, wire them together and suddenly you'll be virtualized, automatized, and servicized.

"… organizations with a poor approach to architecture... will fail and blame SOA instead…"

What rubbish.

SOA is, first and foremost, about the A part of the acronym (architecture). Organizations who already have a solid approach to architecture will likely do reasonably well with SOA; organizations who already have a broken architecture and/or a broken architectural governance practice will likely fail with SOA and then blame SOA on all their problems.

If you follow the history of web-centric systems, services (with a small s) are a very logical progression of web mechanisms. From a technical perspective, SOA is nothing revolutionary, it's evolutionary. BTW, in this context, the concept of an enterprise service bus can be easily explained as a very elegant and simple pattern for location independence/message translation.

"SOA will apparently not only transform your organization and make you more agile and innovative, but your teenagers will start talking to you and you'll become a better lover …"

There are places where SOA is suitable, and places where it is not. SOA, from my experience, is one specific architectural style appropriate for systems of systems wherein some but not necessarily all of those systems are already web-centric. This is an important point: SOA is a useful but insufficient mechanism for architectural decomposition. Some would suggest that SOA is all you need. This is seriously wrong.

To that end, services (with a small s) are best suited to relatively large grained/low frequency interactions rather than small grained/high frequency interactions. For that latter situation, other, more traditional, mechanisms of RPC and/or message passing are better suited.

A serious gap in the current state of the art of services is that we simply don't know how to specify quality of service very well at all. It's one thing to wire together services a la National Instrument's LabView, it's another if there are quality/performance/reliability/security/dependability issues for each of those channels and each of those ports.

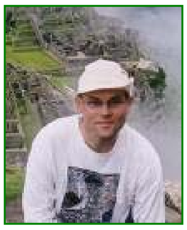
There are also services with a big S: there is a conceptual kind of service that is not manifest as a pure WSDL service but rather something else. Think of a service as a port on a system, with that port having a well-defined interface consisting of a vocabulary of classes, a protocol, and a particular set of messages and resulting behavior. It is a good thing that you can conceptualize a system as a web of services, some of which are Services and some of which are, well, services.

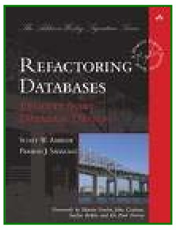
Going back to the A part of SOA, the issue then is one of abstraction, separation of concerns, and all the usual fundamentals of architecture. I've seen some folks suggest creating an SOA from the bottom up: look at a silo, identify the potential services, and publish them, then weave a system together from them. This is in essence technology first. In my experience, this is a recipe for disaster and/or serious over-engineering. You've got to start with the scenarios/business needs, play those out against the existing/new systems, zero in on the points of tangency, and there plan a flag for harvesting a meaningful service. These styles, and their resulting costs/benefits, are rarely discussed.

In a couple of weeks, I'm off to a very different venue, where I'll be giving a talk at the Game Developer's Conference in San Jose. Developing software for games is big business, and this community is starting to discover that the fundamentals are important: you can't build an enduring a business just by hiring bright people, throwing them in a room together, and hoping that they'll do great things.

Grady Booch is Chief Scientist at IBM Rational Software.

Refactoring Databases: Evolutionary Database Design





Refactoring is a key practise of many agile methodologies. In an article based on his recent book, *Scott Ambler* ***discusses how to refactor databases.***

Martin Fowler [1] describes refactoring as a disciplined way to restructure code in small steps. Refactoring enables you to evolve your code slowly over time and thereby take an evolutionary (iterative and incremental) approach to programming. A critical aspect of a refactoring is that it retains the behavioral semantics of your code. You do not add functionality when you are refactoring, nor do you take it away. A refactoring merely improves the design of your code – nothing more and nothing less.

Similarly, a database refactoring [2, 3] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics – in other words, you cannot add new functionality or break existing functionality, nor can you add new data or change the meaning of existing data. A database schema includes both structural aspects, such as table and view definitions, and functional aspects, such as stored procedures and triggers. I use the terms code refactoring to refer to traditional refactoring as described by Martin Fowler and database refactoring to refer to the refactoring of database schemas. The process of database refactoring is the act of making these simple changes to your database schema.

Informational semantics refers to the meaning of the information within the database, from the point of view of the users of that information. Preserving the informational semantics implies that if you change the values of the data stored in a column, the clients of that information should not be affected by the change – for example, if you apply the Introduce Common Format database refactoring to a character-based phone number column to transform data such as (416) 555-1234 and 905.555.1212 into 4165551234 and 9055551212, respectively. Although the format has been improved, requiring simpler code to work with the data, from a practical point of view the true information content has not. Note that you would still choose to display phone numbers in (XXX) XXX-XXXX format, you just would not store the information in that manner.

When preserving behavioral semantics the goal is to keep the black-box functionality the same – any source code that works with the changed aspects of your database schema must be reworked to accomplish the same functionality as before. For example, if you apply Introduce Calculation Method, you may want to rework other existing stored procedures to invoke that method rather than implement the same logic for that calculation. Overall, your database still implements the same logic, but now the calculation logic is just in one place.

Why Database Refactoring?

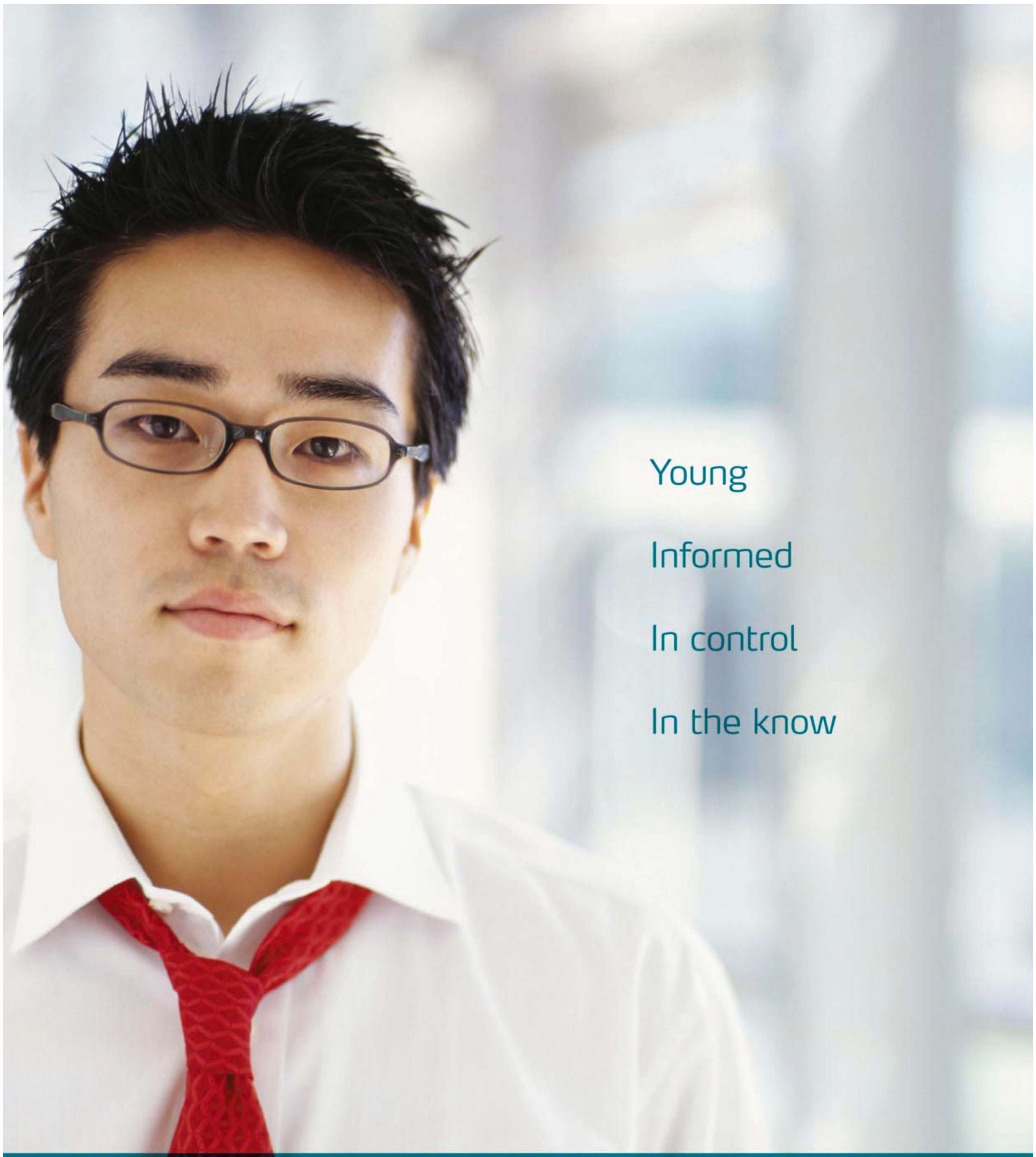
When I speak about database refactoring, and agile database techniques in general, at conferences I always like to get the audience thinking outside of the box. I do this by asking a collection of fairly straightforward questions and asking for a show of hands. Three of my favorite questions are “Do any of you work in organizations where you have perfect data sources?”, “If I was to ask you to go back to your organization tomorrow and rename a column in the most important table in your production database, could you successfully do so in less than a day?” and “Do you have application development teams going around your data group and doing the database design by themselves?”.

The audience will usually laugh at the first question, and frankly I’ve never seen anyone answer yes to it. I then ask the follow-up question “Do any of your organizations have a viable strategy for addressing your data-oriented problems, other than trying to make sure it doesn’t get any worse?” and very rarely does a hand go up. I then point out that the strategy of making sure things don’t get worse is a losing strategy because all it takes is one team to put in yet another silo database and the situation has grown. I can usually hear a pin drop after stating this. Clearly there are some serious problems out there in data land.

Laughter is also usually the reaction to the second question, particularly when there are many people in the audience working in large organizations, although I sometimes there are people in the audience who answer in the affirmative. This is

Database Refactoring - Key Points

- Database refactoring is a simple change to your database schema that improves the design without changing the semantics.
- Database refactoring is one of several techniques which enable data professionals to work in an evolutionary manner.
- Database refactoring enables your to safely fix existing legacy data sources.
- Due to high-levels of coupling within your data architecture, you will require a transition period during which you support both the old and new schemas.
- You need a database regression test suite in place to support database refactoring.
- Few data management organizations currently have coherent strategies for fixing legacy data or effective database testing in place. Sadly, few have even thought of the ideas.
- Just as the agile community has raised the bar for quality in application development, now we're raising the bar for data management.



Meet David Tran, 26, Senior Software Engineer

As a senior software engineer with a multi-national company, David manages a wide number of projects from inception to completion, typically under strict timelines. And with the added responsibility of mentoring and developing a team of junior developers, David's time is precious to say the least.

That's why he's a regular visitor to the InternationalDeveloper.com website. Through the site David has access to a broad range of relevant industry news, reviews and resources that help him stay on top of the big issues in minimal time. So if your time is precious and you want to stay in the know, visit InternationalDeveloper.com today.

international ::developer

InternationalDeveloper.com

either because they work in small organizations with few applications or database access is encapsulated; in either case renaming a column is a relatively trivial task. The people who laugh know that if they were to attempt such a thing they would break numerous applications. Sadly, they have no expectation of their data management group even being able to accomplish a trivial task such as renaming a column, let alone doing something that could actually add value to your organization. Although it may seem that I'm being a bit unfair to the data management folks out there, as far as I'm concerned if they want to be in that role then they need to be responsible for actually fulfilling its responsibilities. Worse yet, as I'll show you in this article, it is in fact possible to rename a column in a production database even when hundreds of heterogeneous applications are coupled to it.

The third question usually results in the majority of the audience saying yes, once again particularly so when there are many people from large organizations. I will often ask a follow-up question such as "And do the developers do a less-than-perfect job of the database design?" which often gets people laughing. After doing so, I usually see a few smug looks on some faces, so then I ask "And how many of you work in organizations that give developers the training that they need to do the database design properly?" they're often not smirking anymore. I suspect that the reason why developers avoid working with the data management groups in their organizations is that they find them too difficult to work with, or simply too slow. At the Software Development 2006 conference Dagna Gaythorpe, a well-respected data professional, started one of her talks with the joke "When you walk up to a data professional, before you can say a thing they blurt out 'It'll take 3 months, now what's the question?'". Although this is obviously an exaggeration, it isn't too far off the mark within many organizations.

The answers to these questions reveal two fundamental reasons why you want to be able to refactor your databases:

- To repair existing legacy databases *[Ed's note: Michael Feather's "working with legacy code" is also contained in this issue]*. Database refactoring enables you to safely evolve your database design in small steps, making it an important technique for improving the legacy assets within your organization This is much less risky than a "big bang" approach where you rewrite all of your applications and rework your database schema and release them all into production at once. It is much better than the "let's try not to allow things to get any worse" strategy currently employed by most data management groups.
- To support evolutionary software development. Modern software development processes, including the Rational Unified Process (RUP), Extreme Programming (XP), Agile Unified Process (AUP), Scrum, and Dynamic System Development Method (DSDM), are all evolutionary in nature. Craig Larman [4] summarizes the research evidence, as well as the overwhelming support among the thought leaders within the IT community, in support of evolutionary approaches. Unfortunately, most data-oriented techniques are serial in nature, relying on specialists performing relatively narrow tasks, such as logical data modeling or physical data modeling. Therein lies the rub – the two groups need to work together, but both want to do so in different manners. I believe that data professionals need to adopt evolutionary techniques, such as database refactoring, which enable them to be relevant to modern development teams. Luckily these techniques exist [3], and they work quite well, it is now up to data professionals to choose to adopt them.

Example Database Refactorings

- **Add Foreign Key Constraint**. Add a foreign key constraint to an existing table to enforce a relationship to another table.
- **Apply Standard Codes**. Apply a standard set of code values to a single column to ensure that it conforms to the values of similar columns stored elsewhere in the database.
- **Introduce Calculation Method**. Introduce a new method, typically a stored function, which implements a calculation that uses data stored within the database.
- **Migrate Method to Database**. Rehost existing application logic in the database.
- **Move Column**. Migrate a table column, with all of its data, to another existing table.
- **Replace One-To-Many with Associative Table**. Replace a one-to-many association between two tables with an associative table.
- **Replace One-To-Many with Associative Table**. Replace a one-to-many association between two tables with an associative table.
- **Use Official Data Source.** Use the official data source for a given entity, instead of the current one which you are using.

Implementing a Database Refactoring

Database refactorings are conceptually more difficult than code refactorings: Code refactorings only need to maintain behavioral semantics, whereas database refactorings must also maintain informational semantics. Worse yet, database refactorings can become more complicated by the amount of coupling resulting from your database architecture. Coupling is a measure of the dependence between two items; the more highly coupled two things are, the greater the chance that a change in one will require a change in another.

Some project teams find themselves in a relatively simple, "single-application database" architecture, and if so they should consider themselves lucky because database refactoring is fairly easy in that situation – you merely change your database schema and update your application to use the new version of the schema. I never seem to work in situations like this, but they're rumored to exist so I thought I'd mention them.

What is more typical is to have many external programs interacting with your database, some of which are beyond the scope of your control. In this situation you cannot assume that all the external programs will be deployed at once, and must therefore support a transition period during which both the old schema and the new schema are supported in parallel. This situation is more difficult because the individual applications will have new releases deployed at different times over the next year and a half. Figure 1 depicts a UML 2 Activity diagram that overviews the database refactoring process [3].

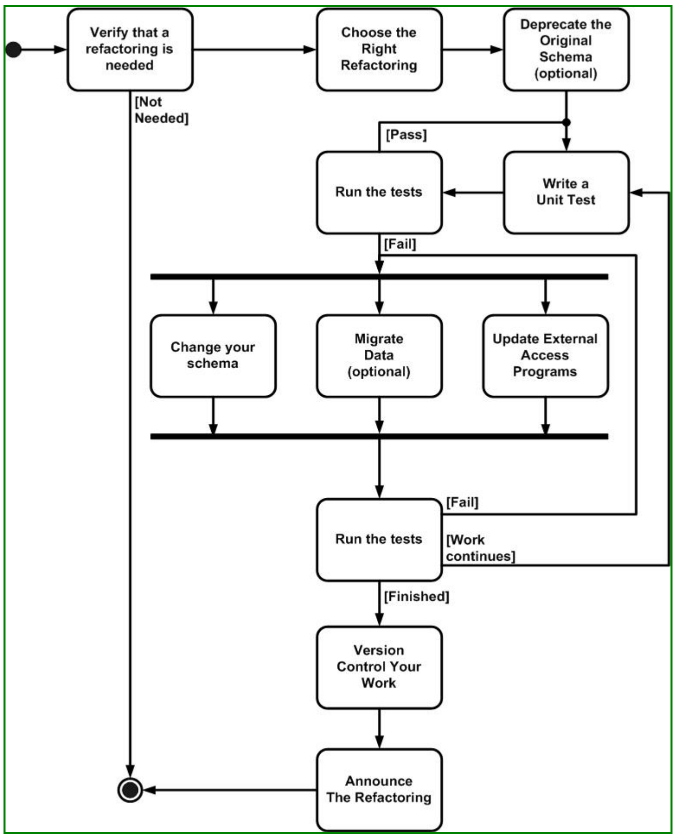


Figure 1. The database refactoring process.

To put database refactoring into context, let's step through a quick example. You are about to implement a new requirement which involves working with the first names of customers. You look at the existing database schema for the Customer table, depicted in Figure 2 , and realize that the column name isn't easy to understand. You decide to apply the Rename Column refactoring to the FName column to rename it to FirstName so that the database design is the best one possible which allows you to implement the new requirement.

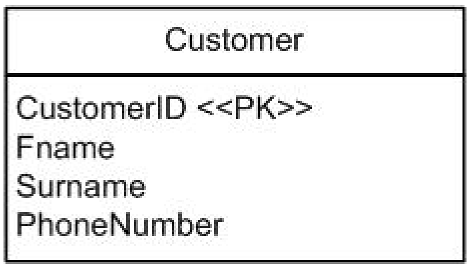


Figure 2. The initial database schema for Customer.

Agilists typically work together as a pair; one person should have application programming skills, the other database development skills, and ideally both people have both sets of skills. This pair begins by determining whether the database schema needs to be refactored. Perhaps the programmer is mistaken about the need to evolve the schema, and how best to go about the refactoring. The refactoring is first developed and tested within the developer's sandbox. When it is finished, the changes are promoted into the project-integration environment, and the system is rebuilt, tested, and fixed as needed.

To apply the Rename Column refactoring in the development sandbox, the pair first runs all the tests to see that they pass. Next, they write a test because they are taking a Test-Driven Design (TDD) approach [5, 6, 7]. A likely test is to access a value in the FirstName column.

After running the test and seeing it fail, they implement the actual refactoring. To do this they introduce the FirstName column and the SynchronizeFirstName trigger as you see in Figure 3, and the Oracle code to do this follows. Due to a lack of tooling at the time of this writing, this code would be captured as a single "change script".

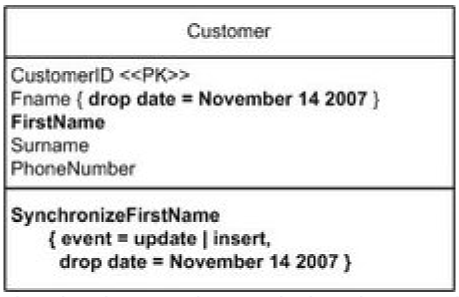

Customer
CustomerID <<PK>>
Fname { drop date = November 14 2007 }
FirstName
Surname
PhoneNumber
SynchronizeFirstName
{ event = update | insert,
drop date = November 14 2007 }

Figure 3. The database schema during the transition period.

The trigger is required to keep the values in the columns synchronized – each external program accessing the Customer table will at most work with one but not both columns. At first, all production applications will work with FName, but over time they will be reworked to access FirstName instead. There are other options to do this, such as views or synchronization after the fact, but I find that triggers work best.

```
ALTER TABLE Customer ADD FirstName VARCHAR(40);

COMMENT ON Customer.FirstName ‘Renaming of FName column,
finaldate = November 14 2007’;

COMMENT ON Customer.FName ‘Renamed to FirstName, dropdate =
November 14 2007’;
UPDATE Customer SET FirstName = FName;

CREATE OR REPLACE TRIGGER SynchronizeFirstName
BEFORE INSERT OR UPDATE
ON Customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
DECLARE
BEGIN
  IF INSERTING THEN
    IF :NEW.FirstName IS NULL THEN
      :NEW.FirstName := :NEW.FName;
    END IF;
    IF :NEW.Fname IS NULL THEN
      :NEW.FName := :NEW.FirstName;
    END IF;
  END IF;

  IF UPDATING THEN
    IF NOT(:NEW.FirstName=:OLD.FirstName) THEN
      :NEW.FName:=:NEW.FirstName;
    END IF;
    IF NOT(:NEW.FName=:OLD.FName) THEN
      :NEW.FirstName:=:NEW.FName;
    END IF;
  END IF;
  END;
```

The FirstName column must be populated with values from the FName column. The easiest way to do this is to simply run the following SQL code. This code would be captured as single script referred to as a “migration script”.

```
UPDATE Customer SET FirstName = FName;
```

You need to run both columns, FName and FirstName, in parallel during a transition period of sufficient length to give the development teams time to update and redeploy all of their applications. This transition period could be several years in length, depending on the ability of your project teams to get new releases into production. In this case we’ve decided that the transition period will run to November 14, 2007 (roughly 1.5 years in this case).

The pair rerun the test suite and see that the tests now pass. They then refactor the existing tests, to work with the FirstName column rather than the FName column. Once the database refactoring is completed in their development work environment, the pair promotes their work into the team’s integration sandbox where they rebuild and rerun the tests, fixing any problems which they find. To update the database schema, the pair runs the appropriate change and migration scripts in the appropriate order.

This promotion strategy continues into a pre-production integration testing environment and then eventually into production. Depending on your need, you could implement and then deploy the refactoring within a single day, although more realistically it would be several months until the next major release of your application and at that point you would deploy the refactoring along with any other updates that you’ve made.

After the transition period, you remove the original column plus the trigger(s), resulting in the final database schema of Figure 4. The Oracle code to do this is shown below, which would be captured in a “transistion script”. You remove these things only after sufficient testing to ensure that it is safe to do so. At this point, your refactoring is complete.

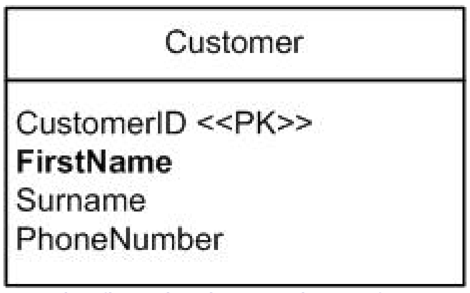


Figure 4. The final database schema for Customer.

```
--On or following Nov 14 2007 DROP TRIGGER
SynchronizeFirstName;ALTER TABLE Customer DROP COLUMN FName;
```

There is a little bit more to successfully implementing a database refactoring than what I’ve described. You need a way to coordinate the refactoring efforts of all the development teams within your organization, clearly something that may prove quite difficult. You also need to get good at deploying refactorings in production, once again coordinating the efforts of several teams. In Refactoring Databases [3], my co-author Pramod Sadalage and I discuss several strategies for doing these things.

Database Refactoring and Testing

You can have the confidence to change your database schema only if you can easily validate that the database still works with your application after the change, and the only way to do that is to take a TDD-based approach where you write a test and then you write just enough code to fulfill the test. You continue in this manner until the database refactoring has been implemented fully. You will potentially need to write tests that:

- Test your database schema. You can validate many aspects of a database schema: Stored procedures and triggers, referential integrity (RI) rules, view definitions, default value constraints, and data invariants [8].
- Test the way your application uses the database schema. Your database is accessed by one or more programs, including the application that you are working on. These programs should be validated just like any other IT asset within your organization.
- Validate your data migration. Many database refactorings require you to migrate and sometimes even cleanse the source data. In our example, we must copied the data values from FName to FirstName as part of implementing the refactoring.

Why Not Just Get it Right to Begin With?

I am often told by existing data professionals that the real solution is to model everything up front, and then you would not need to refactor your database schema. Although that is an interesting vision, and I have seen it work in a few rare situations, experience from the past three decades has shown that this approach does not seem to be working well in practice for the overall IT community [*Editor's note: see Ed Yourdon's retrospective on Structured Analysis in this issue*]. The traditional approach to data modeling does not reflect the evolutionary approach of modern methods such as the RUP and XP, nor does it reflect the fact that business customers are demanding new features and changes to existing functionality at an accelerating rate. The old ways simply aren't sufficient any more, if they ever were [11].

I suggest that you take an Agile Model-Driven Development (AMDD) approach [9, 10], in which you do some high-level modeling to identify the overall "landscape" of your system, and then model storm the details on a just-in-time (JIT) basis. You should take advantage of the benefits of modeling without suffering from the costs of over-modeling, over-documentation, and the resulting bureaucracy of trying to keep too many artifacts up-to-date and synchronized with one another. Your application code and your database schema evolve as your understanding of the problem domain evolves, and you maintain quality through refactoring both.

AMDD is different than traditional Model Driven Development (MDD), exemplified by the Object Management Group (OMG)'s Model Driven Architecture (MDA) standard (www.omg.org) , in that it doesn't require you to create highly-detailed, formal models. Instead, AMDD is a streamlined approach to development that reflects agile software development values and principles, providingway to create artifacts such as physical data models that are critical to the success of agile DBAs. The collaborative environment fostered by AMDD promotes communication and cooperation between everyone involved on your project. This helps to break down some of the traditional barriers between groups in your organization and to motivate all developers to learn and apply the wide range of artifacts required to create modern software – there's more to modeling than data models.

In Conclusion

Database refactoring is a database implementation technique, just like code refactoring is an application implementation technique. You refactor your database schema to ease additions to it. You often find that you have to add a new feature to a database, such as a new column or stored procedure, but the existing design is not the best one possible to easily support that new feature. You start by refactoring your database schema to make it easier to add the feature, and after the refactoring has been successfully applied, you then add the feature. The advantage of this approach is that you are slowly, but constantly, improving the quality of your database design. This process not only makes your database easier to understand and use, it also makes it easier to evolve over time; in other words, you improve your overall development productivity.

My experience is that data professionals can benefit from adopting modern evolutionary techniques similar to those of developers, and that database refactoring is one of several important skills that data professionals require. Unfortunately, the data community missed the object revolution of the 1990s, which means they missed out on opportunities to learn the evolutionary techniques that application programmers now take for granted. In many ways, the data community is also missing out on the agile revolution, which takes evolutionary development one step further to make it highly collaborative and cooperative.

References / Recommended Reading

1. **Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Menlo Park, California: Addison Wesley Longman, Inc.**
2. **Ambler, S.W. (2003). Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: John Wiley & Sons.** http://www.ambysoft.com/books/agileDatabaseTechniques.html
3. **Ambler, S.W. and Sadalage, P.J. (2006). Refactoring Databases: Evolutionary Database Design. Boston: Addison Wesley.** http://www.ambysoft.com/books/refactoringDatabases.html
4. **Larman, C. (2004). Agile and Iterative Development: A Manager's Guide. Boston: Addison-Wesley.**
5. **Astels D. (2003). Test Driven Development: A Practical Guide. Upper Saddle River, NJ: Prentice Hall.**
6. **Beck, K. (2003). Test Driven Development: By Example. Boston, MA: Addison Wesley.**
7. **Ambler, S.W. (2004c). Introduction to Test Driven Development (TDD).** http://www.agiledata.org/essays/tdd.html
8. **Ambler, S.W. (2006a). A Roadmap for Regression Testing Relational Databases.** http://www.agiledata.org/essays/databaseTesting.html
9. **Ambler, S.W. (2002). Agile Modeling: Best Practices for the Unified Process and Extreme Programming. New York: John Wiley & Sons.** http://www.ambysoft.com/books/agileModeling.html
10. **Ambler, S.W. Agile Model Driven Development (AMDD).** http://www.agilemodeling.com/essays/amdd.htm
11. **The Agile Data Home Page.** http://www.agiledata.org/

Scott W. Ambler is an industry-recognized software process improvement (SPI) expert and is the Practice Leader Agile Development within IBM's Methods group. www-306.ibm.com/software/rational/bios/ambler.html is his personal home page, and he is the author of several books and is a senior contributing editor with Dr. Dobb's Journal. Material for this article was modified from Refactoring Databases: Evolutionary database Design by Scott W. Ambler and Pramod J. Sadalage (Addison Wesley 2006).

The Object Database Alternative

With an increasing number of Open Source and proprietary Object Databases becoming available, do they really provide a viable alternative to relational databases? *Rick Grehan* ***tells us why he believes they are…***

When software developers think "database", they usually think "relational database". Think further, though, and that's a bit odd. Most development today is being done in an OO language (Java, C#, etc.) Moving data between objects and relational tables requires "translation" code that must work bidirectionally – extracting object data into SQL statements, or pulling data from returned tables and assembling the result into a new object.

Wouldn't it be nice if such translation code were unnecessary? Wouldn't code be easier to read and manage if objects could be placed in the database (and withdrawn from it) wholesale; with no conversion necessary? After all, the application code works with objects. Doesn't it make sense to have the database manipulate objects ... rather than "bits and pieces" of objects?

Objects In The Database

With the near-universal adoption of object-oriented languages as the foundation for new application development, interest in object databases is growing. While an object database enjoys several advantages over a relational database, its most significant edge is the simple fact that using an object database does not require the developer to master two different paradigms: the object paradigm for the application, and the relational paradigm for the database.

Evidence of the widening appeal of object databases can be found in the numerous commercial and open-source offerings. Commercial products include:

- **Versant Corporation (http://www.versant.com/)** is the home of the FastObjects object database. Formerly known as Poet, the FastObjects database is available in Java and .NET versions, and packs a remarkable punch for its size. (FastObjects has a bigger cousin, called simple "Object Database" that also provides a C++ interface.)
- **Matisse Corporation (http://www.matisse.com/)** markets a database also named after the French artist. The Matisse database is described as a "post-relational" database, which means that the database is just as comfortable storing and retrieving "native" objects as it is processing relational SQL statements.

Open-source offerings include:

- **Ozone (http://www.ozone-db.org/frames/home/what.html)** is an object-oriented DBMS implemented in Java whose goal is to allow developers to create POJOs (plain old Java objects) and "let them run in a transactional database environment."
- **Prevayler (http://www.prevayler.org/)** is a Java persistence engine that takes the somewhat unorthodox approach of keeping all objects in RAM. This manner of object persistence rests on the related facts that the cost of RAM is dropping as its density rises. Period 'snapshots' back the data to disk, so the database can be reconstructed in case of a crash.
- **The Apache ObjectRelationalBridge (OJB - http://db.apache.org/ojb/)** is, strictly speaking, an object- relational mapping layer. An application using OJB sees an object database, but the back-end converses with any JDBC-compliant RDBMS. The Apache OJB is impressive in that it supports at least four object-database APIs.
- **db4o (http://www.db4objects.com/)** is an object database available for Java, .NET, or Mono. It's outstanding characteristics are its straightforward API, and the ease with which it can

Why use an OODBMS

Class schema is database schema	With an ODBMS, once you've designed your application's class schema, your database schema is done.
No object-to-relational translation code	Because there is no relational database 'hiding behind' the application, you don't have to write code to translate between objects and relational tables.
Objects are manipulated as objects	Objects don't have to be peeled apart to be stored, an re-assembled when retrieved. In addition (given the proper ODBMS back-end) object relationships are automatically reflected in the database (rather than having to be mimicked by added tables and columns).
A single language covers all	Many ODBMS packages (e.g., db4o, as illustrated in the article) do not require a separate database manipulation language, such as SQL, to describe database operations. Simply put, the application is written in one language.

be incorporated into an application. We'll be using db4o later, to demonstrate some of the more compelling reasons for choosing an object database over a relational one.

The Relational Situation

With an RDBMS "behind" your application, queries to the database are typically specified as SQL strings. These strings express a different language than the language of the application. This "passenger language" carries its own semantics and syntax. And because the commands are strings, they are untouched at compile time. They must be parsed and executed by an SQL engine at runtime.

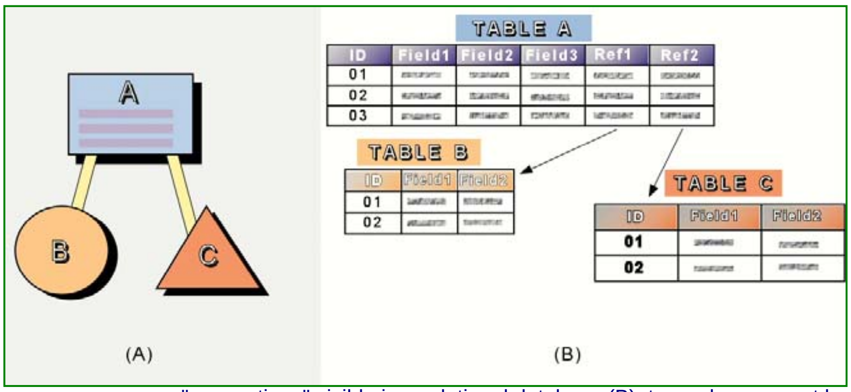
Furthermore, depending on the structure of your application, that SQL engine might reside entirely within your code's process space; which means that its execution takes place at the expense of CPU cycles that would be otherwise available to your application. Even if you employ stored procedures (which execute in the process space of the database server), your application must perform some sort of translation to move data between the world of objects, and the world of database entities.

We must recognize that there is much good to be said about relational databases; plenty of fine relational database systems are available. The RDBMS MySQL is possibly one of the finest demonstrations that the open-source world can produce top-quality software on par with the best commercial offerings. Nevertheless, an RDBMs is not the only answer for every database application. As already indicated, your application incurs overhead – both in terms of consumed memory and processor cycles – from the SQL parsing and execution engine. In addition, you are "sprinkling" your application with strings of procedural code that are not syntactically checked at compile-time. As a result, you won't know about even the smallest typographical errors until you run the application.

What's worse, you won't know if you have a semantic error in your database code until you execute the application and see the results. For example, if you've written your database code "by hand" there exists a small -- but not nonexistent -- danger that you'll put the wrong object member into the wrong column, or fetch the wrong column into the wrong object member. Such errors won't manifest themselves until you witness whatever effects those mistakes cause.

Also, object relationships must typically be modeled via columns that use foreign keys to reflect object references. These columns exist for no reason other than to provide a unique identifier for the row, so that other rows in other tables (representing other objects) can be "connected". (See Figure 1 [PICT1.JPG]) And, of course, code must be written to ensure that object relations are properly reflected into the database, and that objects drawn from the database are correctly "wired" according to the relationships expressed by the foreign keys.

Figure 1. Representing objects in an RDBMS
In this instance, (A) object A references objects B and C. To make these



"connections" visible in a relational database (B), two columns must be added to TABLE A. Each holds foreign keys that "point" to the rows corresponding to objects B and C in their respective tables.

Consider, for example, a class whose objects include an array (or some other collection) as a member. To model such objects in a relational database requires two tables. One, the 'parent' table, holds data corresponding to the non-array fields of the primary object. Another, the 'child' table, holds data corresponding to the array entries. Furthermore, each row in the child table must carry a foreign-key referece that provides the link between parent and child, so that array members (in the database) can 'know' which parent object they 'belong to.'. (See Figure 2.)

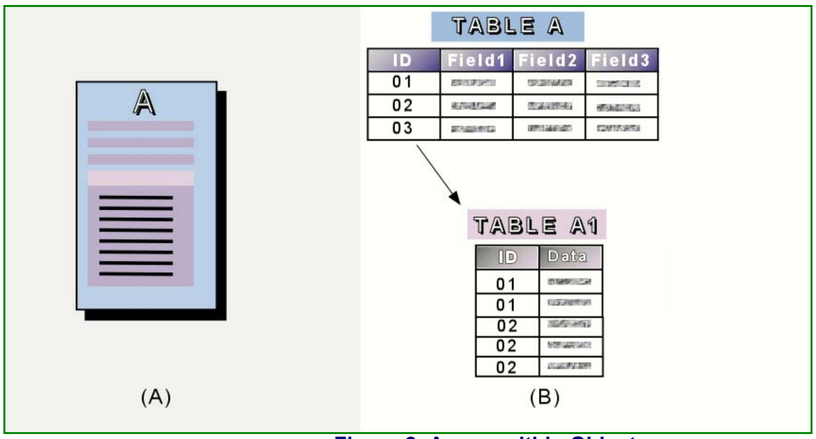


Figure 2. Arrays within Objects
(A) Object A contains an array object (or, possibly a container object such as ArrayList). (B) To model that in a relational database requires two tables -- a 'parent' table (Table A) for the parent object, and a 'child' table (Table A1) for the contents of the array. The unique ID of the parent object is written into those rows in Table A1 that belong to that parent.

In summary, classes must be mapped to tables, objects mapped to rows, and object relationships mapped to specialized columns. The developer must construct an elaborate framework so that a relational database can speak the language of objects, and objects can speak the language of the relational database. The term "impedance mismatch" -- borrowed from the world of electronics -- is often used to describe this gap between the object and the relational realms that the developer must bridge.

Object/Relational Databases

Object/relational (O/R) database systems offer a sort of middle- ground, allowing the developer to create an object-oriented application with a relational database "backend". And, in most cases, object/relational database systems offer some form of "assistance" -- relieving the developer of at least part of the tedium of building the translation code we mentioned above.

From the application's perspective, the database holds objects; from the database's perspective, the application is using relations. Huddling invisibly between the two is a translation layer that takes the objects passed in from the application, and turns them into relational operations for consumption by the relational database. Relational data moving through the translation layer in the opposite direction are converted by the layer into objects for the application.

Because the translation layer handles the conversion work, the programmer need not deal with SQL directly. And many object/relational database systems can "talk" to a variety of different relational back-ends, allowing the developer to select the specific RDBMS that he or she believes to be the most suitable for the application.

Moving data between objects and relational tables requires "translation" code…. Wouldn't it be nice if such translation code were unnecessary?

However, even though the developer doesn't have to create (and, in most cases, never even sees) the translation code, that code is still present in the application -- consuming space and eating CPU cycles. And the side-effects of the underlying relational database cannot be completely eliminated. The developer must somehow describe the object structures and class relationships to the O/R system, and -- in some cases -- provide guidance to the database as to how objects are to be stored in the tables. Typically, this specification is expressed in a schema mapping file.

A sort of reverse-example of such guidance can be seen in the Apache Torque database project's tutorial. Torque is a object-to- relational database mapping technology that requires you to define your database structure, and from that constructs classes that the Torque "engine" can shuttle to and from the database tables. An excerpt from a Torque database schema file looks like this:

```
<database
  name="bookstore"
  defaultIdMethod="idbroker">

  <table name="publisher" description="Publisher Table">
    <column
      name="publisher_id"
      required="true"
      primaryKey="true"
      type="INTEGER"
      description="Publisher Id"/>
    <column
      name="name"
      required="true"
      type="VARCHAR"
      size="128"
      description="Publisher Name"/>
  </table>
....
```

This schema file must be passed through a pre-processing step (automated by the Maven build tool) that generates four classes for each table. These include "peer" classes, that carry the logic for manipulating the corresponding Java objects (that hold the actual data). For example, to insert a "publisher" object in the database, you would call something like:

```
  PublisherPeer.doInsert(pubObject);
```

where pubObject is an instantiated Publisher object. (The Publisher class is also a generated class.)

In short, Torque takes a direct approach to mapping classes to tables, and instantiated objects to rows within those tables. You describe the table structure, and Torque builds the source code for the classes for you. Not a particularly bad idea, but – as illustrated – the relational roots can never be completely hidden.

Life Is Simpler

A database programmer's life becomes a good deal simpler with a "true" object database; that is, one that treats objects as, well, objects throughout.

Because an object database manipulates objects "wholesale", no translation code need be written to move data between objects in the application and tables in the relational database. The database and application both deal with objects in the same form. Hence, there is no need to embed a separate "database language" in the application's code, and the developer doesn't have to learn a programming language other than the application's. In addition, no database language engine is required (either in the application's processor space, or on a separate server) to parse and execute the database language's commands.

The list of "unnecessaries" goes on: no schema mapping file is required. In a very real sense, the developer writes the schema when specifying a class's structure. Everything that the compiler and runtime needs to know about an object's structure and its relationship to other objects is visible in the code. Similarly, all that the database needs to know about the architecture of persistent objects is in the definition of those objects' classes.

An Open-Source Example: db4o

db4o is an open-source object database available in Java and .NET flavors. It is a "native" object database in several senses. It runs entirely in the execution environment of the application. That is, the Java version is written in Java, and the .NET version is written in C#. A single library linked to an application is all that is needed to provide that application with all of db4o's capabilities. (A version of db4o also exists for development in the Mono environment.)

In addition, db4o works with unmodified objects. (See Figure 3.) That is, the developer need do nothing special to an object to make it "persistent-capable." For example, some object databases require that persistent object classes be derived from a persistence-capable base class. Others require that the classfiles of persistent classes be passed through a pre-processor. This pre-processor modifies the classfiles by "injecting" code, so that, at execution time, objects derived from such classes inherit the invisibly-added capability of persistence. With db4o, no additional work is required to make an object eligible for storage in the database; you simply put it there.

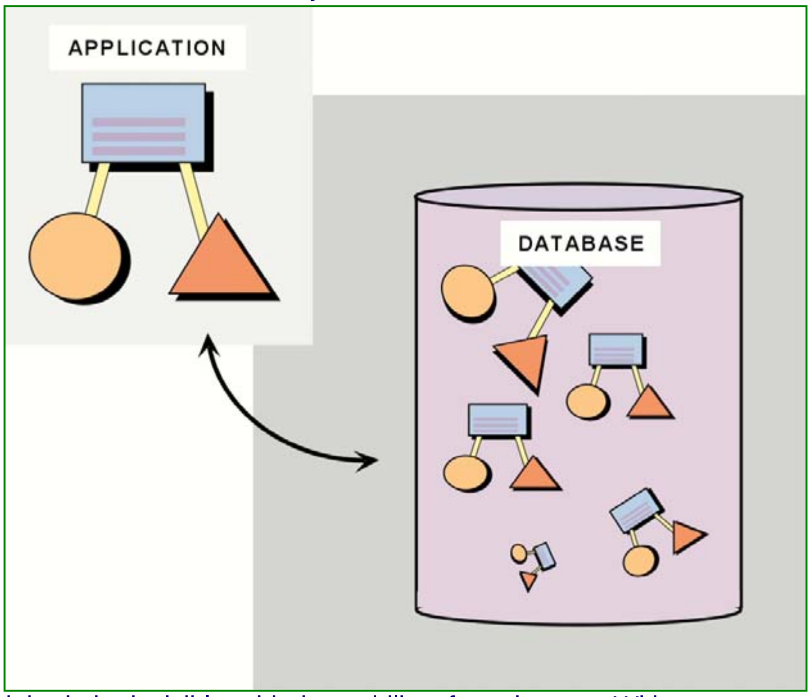


Figure 3. A Database Of Objects
db4o treats objects as objects. Objects are stored in the database "wholesale", with their structure and object references "intact". This is in contrast with a relational database back-end, wherein the application must disassemble objects to put them in the database, and re-assemble them to fetch them out.

One of db4o's outstanding features is its uncomplicated API. For example, suppose you had defined a class called MyPrinter, with the following structure:

```
class MyPrinter {
   public string name;
   public string manufacturer;
   public string model;
   public float cost;
   ...
}
```

To store an object to the database, the code looks something like this:

```
db4oDB.set(myPrinterObject);
```

where db4oDB is an ObjectContainer (an instantiation of the class that models the database), and myPrinterObject is a reference to an instance of MyPrinter that will be stored in the database.

Notice that you don't have to tell db4o anything about the structure of myPrinterObject. It could be as simple as an object with only primitive members, referencing no other objects; or as complex as the "root" of a tree of objects connected to other objects to an arbitrary depth. (So, if myPrinterObject referenced another object, say objectB, then storing myPrinterObject would also store objectB.) In short, you don't have to craft a schema definition file for db4o's benefit. The schema is in the class structure itself, and db4o discovers that structure by navigating object references via reflection.

Meanwhile, had our object been stored in a relational database, we might have had to use the following code (assuming, in this instance, using C# for .NET, and calling upon the Ole libraries in the .NET Framework);

```
. . .
String insstr = “INSERT INTO PrinterTable (name,
manufacturer, model, cost) “ +
  “VALUES (‘“ + myPrinterObject.name + “’, ‘“ +
  myPrinterObject.manufacturer + “’, ‘“ +
  myPrinterObject.model + “’, “ +
  myPrinterObject.cost +”)”;
OleDbCommand insCommand = new OleDbCommand(insstr,
connection);
InsCommand.ExecuteNonQuery();
. . .
```

Where we have taken the less-than-secure route of creating our INSERT SQL command string by simply concatenating strings together. (The connection object represents the connection to the database, which the code snippet presumes has already been opened.) Even had we used binding variables in our SQL statement, we would still have had to create a string, and issued calls to bind actual instance variables to their “markers” in the SQL string.

Deleting an object with db4o is a call of equal simplicity:

```
db4oDB.delete(myPrinterObject);
```

Again, we need provide db4o no information concerning the myObject's structure.

However, whereas storing an object also stores all referenced objects, deleting an object does not automatically delete all referenced objects. We have to tell db4o specifically if we want to do that, and the reason for this apparent imbalance is obvious once you ponder it a few moments. If two objects, A and B, reference a third, C; and deleting A will delete C, then object B is left with a dangling reference. The db4o API gives the developer the ability to "tune" the extent of deletions, precisely to preclude the possibility of inadvertently creating a dangling reference.

Similarly, suppose you fetch object A from the database. Object A references object B. Object B references C, and so on off to lots and lots of objects. You may not want to fetch the entire structure of connected objects when you fetch Object A. So, db4o provides an "activation depth" setting, which lets you control how much of an object tree you pull in when you fetch one object.

By default, the activation depth is set to 5, which is sufficient for even moderately large structures. Change the depth to 2, and only the root object and its immediate "child" objects are fetched from the database. If, however, you need even more control over the activation, db4o's API provides triggers and callbacks that let you tune the retrieval of objects from the database at runtime.

Multiple Query Mechanisms

db4o boasts several query mechanisms, each suitable for different query needs.

Query-by-example (QBE) is db4o's easiest-to-grasp style of searching the database, and is ideal for straightforward "is equals to" queries. To fetch an object (or set of objects) using QBE, you define a "template" object, pass that template to db4o's query engine, and db4o returns the set of matching objects.

The template object is nothing more than an object whose members are filled with those values you want matched in the target objects. For example, let us return to the MyPrinter class we had defined earlier. If we want to fetch all "Epson" printers from the database, the code looks like this:

```
MyPrinter thePrinter;
...
MyPrinter printerTemplate = new MyPrinter();
printerTemplate.manufacturer = "Epson";
ObjectSet result = db4oDB.get(printerTemplate);
while (result.hasNext())
{
  thePrinter = (MyPrinter)result.next();
  ...do something with thePrinter...
}
```

As earlier, db4oDB is our database's ObjectContainer. We can iterate through the result collection, which contains the objects fetched from the database that match the query.

As stated above, QBE's underlying comparison mechanism corresponds to "is equal to" matches. This is obviously a limitation. Also, QBE cannot match the numeric value of 0, because storing a 0 in a numeric field causes that field to be ignored for the query. Nevertheless, as a mechanism for quickly locating a "root" object to a network of objects, and then using object references to navigate throughout the network, QBE is ideal. (Note that you don't have to query an object to fetch it from the database. If Object A is in memory, and it references Object B in the database, db4o lets you retrieve Object B by "activating" it via the A-to-B reference.)

As a comparison, suppose we had wanted to perform that same query on an RDBMS. Again, using the OleDB library calls available in the .NET Framework, it would look something like this:

```
string selstr = “SELECT name, model, cost FROM PrinterTable
WHERE “ +
 “manufacturer = ‘Epson’”;
OleDbCommand command = new OleDbCommand(selstr, connection);
OleDbDataReader reader = command.ExecuteReader();
while(reader.read())
{  thePrinter = new MyPrinter();
  thePrinter.name = reader.GetString(0);
  thePrinter.model = reader.GetString(1);
  thePrinter.cost = reader.GetFloat(2);
  thePrinter.manufacturer = “Epson”;
  . . . do something with thePrinter . . .
}
reader.Close();
```

Notice, as already mentioned, that we had to “assemble” the object from pieces of data fetched from the table.

Native Queries

For more complex queries, db4o's Native Query system represents what is possibly the ultimate in query convenience. To implement a native query, you need merely write a method that filters out those objects you want the query to match. And the complexity of the filtering is practically limited only by the capabilities of the native language (Java, C#, VB.NET, etc.).

An example will make this clearer. Returning to our MyPrinter class above, suppose we wanted to construct a query that returned for us all printers by the manufacturer "Epson" that were also less than $200. We can implement a native query that accomplishes this by first defining an extension of db4o's Predicate class:

```
public static class CheapPrinters extends Predicate
{
  public boolean match(MyPrinter _printer)
  {
     return((_printer.manufacturer.equals("Epson") &&
            (_printer.cost < 200.00));
  }
}
```

Once we've defined this Predicate "query class", we can pass an instance of it to the query() method of an open database object:

```
  ObjectSet result = db4oDB.query(new CheapPrinters());
```

and the result ObjectSet can be accessed just as before. Only this time, its members are those database MyPrinter objects that satisfy the criteria established by the match() method.

The idea underpinning Native Queries is simple and powerful. The match() method, which can be as simple or as complex as the situation demands, returns true if the candidate object (passed into the method by db4o's query engine) matches whatever conditions are defined. The method returns false otherwise. It is as if -- when the query is executed – the db4o engine marches each candidate object up to the Predicate object's match() method, and the match() method announces "true" or "false". Only those objects that received a "true" pronouncement are placed in the returned results collection.

As with QBE, there is no SQL code to write. If there are syntax errors in the code, those are caught at compile time ... not at runtime.

S.O.D.A.

The last and most intricate query mechanism provided by db4o is S.O.D.A. (Simple Object Database Access). It is also the most powerful, because S.O.D.A. is db4o's underlying query system. Access to the S.O.D.A. API gives the most direct admission into db4o's database engine.

You build a S.O.D.A. query by erecting a tree of constraints on a query object, and issuing an execute() method call against that tree. For example, a S.O.D.A. query that returns all MyPrinter objects whose cost is less than $200 would look like this:

```
  Query query = db4oDB.query();
  query.constrain(MyPrinter.class);
  query.descend("cost").constrain(new Float(200.00)).less();
  ObjectSet result = query.execute();
```

The descend() method call attaches branches to the query tree, and the constrain() method call fastens leaves to those branches. Notice that the root of the query tree is a reference to the MyPrinter class, which establishes the candidate objects.

On the one hand, because portions of S.O.D.A. queries are specified as strings, such queries are not entirely typesafe (as are QBE and native queries). However, S.O.D.A. queries are extremely fast, can be altered at runtime, and are not restricted to "equals-to" conditions.

Most importantly, notice that all three of db4o's query styles are implemented in the native language. In no case does the developer have to "step out" into a different language to express the query. And, in all cases, objects are fetched "whole" from the database. There is no need to assemble objects from primitive data values at query time.

Quiet Transactions

Any time that a database application executes a method call that will modify the database (i.e. a set() or a delete() method call), db4o invisibly begins a transaction session. The transaction session remains in effect until one of three events occurs:

1) The application executes a commit() method call on the ObjectContainer. In that case, all changes performed since the transaction was opened are written to the database.

2) The application executes a rollback() method call on the ObjectContainer. This causes all modifications made to the ObjectContainer to be dropped. That is, the ObjectContainer (the database) is returned to the precise state it was in prior to the transaction's start. The result is as though all the modifications made during the transaction session never happened.

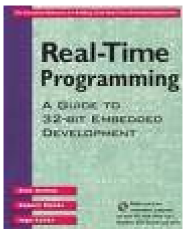
3) The application closes the ObjectContainer (calling the close() method). The close() method performs a silent commit(), so the effect on the database is identical to alternative (1).

So, db4o's transactions give you the ability to organize complex operations -- any mixture of additions, modifications, and deletions -- into units that appear to the database as an atomic operation. That is, if you commit() a transaction, all changes to the database within the transaction take effect; whereas if you rollback() a transaction, no changes to the database within the transaction take effect.

Actually, there is a fourth event that can terminate a transaction. Suppose the system crashes during the transaction session. In that case, when the database application restarts and the database is re-opened, db4o detects the interrupted transaction and returns the database to it's non-corrupted state prior to the start of the transaction session. Consequently, db4o databases are (barring catastrophic destruction of the media on which the database file resides) crash-proof, thanks to the database engine's use of silent transactions.

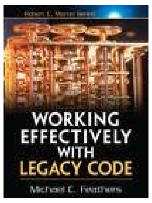
Objective Benefits

Best of all, db4o is easy to incorporate into an object-oriented application. The database engine's behavior "conforms" to the behavior of the rest of the application; that is, objects act like objects, in or out of the database. This is true regardless of which query mechanism you choose. And, once more, it is open source. That connects db4o to the growing inertia of the open- source movement, which is filling the programmer's toolbox with more and more high-quality equipment.



Rick Grehan is a QA Engineer for Compuware/NuMega Labs. His articles have appeared in BYTE Magazine, Dr. Dobbs, Embedded Systems Journal, JavaPro, and other publications. In addition, Rick has co-authored three books on a range of programming topics.

The Testing Lever: Making Progress in Legacy Code



Michael Feathers ***discusses how automated testing can help when working with legacy code…***

Sometimes, I feel like the boy in 'The Emperor's New Clothes.' I'm unduly sensitive to cases where what people describe and what I see are different. One of the cases that hits me the hardest is the difference between code quality as it's often presented in books, and code quality out in the field. It's not that there aren't examples of good code out there in industry, but the fact is I don't get to see them much as I'd like to. I'm a consultant. People call me in when they need help, so it is easy to believe that less than ideal code and broken design are the norm. Many teams I visit have code bases that are large sprawls. Their classes are huge; they often have fifty to a hundred methods, and the median size of the methods? Well, let's just say that it isn't five lines or ten lines – or twenty.

Most code bases (at least the ones I see) are a hodge-podge. And, this isn't just an aesthetic concern. Companies spend significant time and money trying to get work done in these swamps, and it is usually obvious that if the code was structured better, many tasks that might take four days or five days could take one or two instead. Poor code quality is just a way of throwing away money. It doesn't make anyone's life easier, least of all programmers who have to spend most of the time mired in it. What can we do?

Well, when we encounter bad code, we could rewrite it. Sometimes that's the right choice, but large scale rewrites can be hazardous. You have to make sure that your new code does exactly the same thing as your old code, and that can be difficult. No, refactoring is often the better choice: systematically making the code better by transforming it piece by piece. But if we are going to refactor, we have to be very careful. We have to make sure that we work in a way which keeps us from making silly mistakes.

Let's take a look at an example. Here's a piece of ugly code in C++:

```
void Scheduler::perform_consistency_check(string& message)
{
   for(std::vector<Event *>::iterator it = events.begin();
       it != events.end();
       ++it) {
        Event *e = *it;
        if (e->getSlot() > Time6PM
        && dynamic_cast<Meeting *>(e)) {
                report_scheduling_violation(e->getSlot());
                message += "::No meetings after 6PM";
        }
        if (e->getSlot() > Time8PM
        && dynamic_cast<ClientAppointment *>(e))
            message += "::No appointments after 8PM";
        if (e->getSlot() < Time9AM || e->getSlot() > Time6PM
        && dynamic_cast<Flextime *>(e))
            message +=
                "::No Flextime outside of working hours";
        if (e->getSlot() == Time12PM
        && dynamic_cast<Flextime *>(e)
```

```
        && !get_meeting(e->getDate(), Time5PM))
            message += "::No deferred lunch without "
               "late scheduled meeting";
            if (e->getSlot() > Time12PM
           && dynamic_cast<Flextime *>(e)
            && get_meeting(e->getDate(), Time12PM)
            && get_meeting(e->getDate(), Time1PM)
            && get_meeting(e->getDate(), Time2PM)
            && get_meeting(e->getDate(), Time3PM)
            && get_meeting(e->getDate(), Time4PM)
            && get_meeting(e->getDate(), Time5PM))
                message += "::No flextime on afternoons "
                           "of scheduled meetings";
            if (e->getSlot() == Time12PM
            && dynamic_cast<Meeting*>(e)){
                report_scheduling_violation(e->getSlot());
                message += "::No meetings during lunch";
            }
            if (e->getSlot() > Time12PM
            && dynamic_cast<ClientAppointment *>(e)
            && get_meeting(e->getDate(), Time12PM)
            && get_meeting(e->getDate(), Time1PM)
            && get_meeting(e->getDate(), Time2PM)
            && get_meeting(e->getDate(), Time3PM)
            && get_meeting(e->getDate(), Time4PM)
            && get_meeting(e->getDate(), Time5PM))
                message += "::No client appointments on "
                    "afternoons of scheduled meetings";
    }
    dispatch(message);
}
```

I hope you'll agree with me that it isn't the clearest function in the world, and, obviously, we've all seen worse code, but let's list a few of its problems:

1. It's long. Not terribly long, but long enough to make us scroll.
2. It's ill-defined. What exactly is `perform_consistency_check` supposed to do? It does at least three things for the caller. Can you see them?
3. It repeatedly uses reflection (`dynamic_cast`) to make decisions based upon the type of an Event. There is probably a cleaner way of doing the same work.

Suppose that wanted to make this function better. Where would we start? Well, my answer is a little different than what you might expect. My answer is to say that the first thing we should do is figure out *why* want to make it better.

I can imagine what some of you are thinking right now. You're thinking that I just gave a list of reasons a few paragraphs ago: the design is bad so we should just fix it. Well, I'd love to do that most of the time, but the cold brutal truth is that you could literally spend your entire life making the typical legacy code base arbitrarily better, but you wouldn't have time for anything else. If you're going to go through the trouble to make things better you should have a reason. Here are some of the best ones:

1. You have to add a feature to a piece of code, and you don't understand it well enough to make the change confidently.
2. A piece of code is so unclear that it impedes understanding of surrounding areas.
3. You have to fix a bug and you really don't want to go through the trouble of trying to understanding the code again later, and you don't want to inflect that burden on anyone else coming after you either.

Isn't it interesting that all of these reasons have something to do with ease of understanding? I think it's more than interesting, it's significant. Understandability is one of the most important qualities that code can have. When it disappears, the work just gets harder and harder without bound.

What can we do to make legacy code more understandable?

For me, the answer is: testing. We can write tests for existing code that help us when we refactor it. The tests will fail if we change the code in a bad way; but the tests will do much more for us than that. They will allow us to build up the net level of understanding in the system.

Here. I'll show you what I mean in the context of the function we saw earlier.

Our function has this signature:

```
void Scheduler::perform_consistency_check(string& message);
```

It accepts a string called `message`, by reference, and it modifies it. We can write tests which show us how the message string is changed under various conditions:

```
void test_meetings_allowed_upto_six() {
    Scheduler scheduler;
    scheduler.add_event(new Meeting("", Time6AM));
    scheduler.add_event(new Meeting("", Time9AM));
    scheduler.add_event(new Meeting("", Time3PM));
    scheduler.add_event(new Meeting("", Time6PM));

    assert(message == "");
}

void test_meetings_disallowed_after_six() {
    Scheduler scheduler;
    scheduler.add_event(
        new Meeting("Meeting with Jim", Time7PM));
    assert(message == "::No meetings after 6PM");
}

void test_meetings_disallowed_during_lunch() {
    Scheduler scheduler;
    scheduler.add_event(
        new Meeting("Meeting with Jim", Time12PM));
    assert(message == "::No meetings during lunch");
}
```

Here we have a few test cases which show that meetings are allowed until 6PM, but they are forbidden afterwards and during lunch. Were we able to see this in the original code? Yes, but the logic was spread around and surrounded by unrelated conditions. The tests we've written make the logic explicit. Moreover, they are not just documentation. We can execute them and really see whether those statements about the logic in `perform_consistency_check` are true.

Now that we have some tests, we can refactor. We can go into the `perform_consistency_check` function and regroup the logic for meetings. We can also extract the logic into its own function and call it from `perform_consistency_check`:

```
bool Scheduler::all_meetings_are_valid(string& message)
{
    bool result = true;
    for(std::vector<Event *>::iterator it = events.begin();
    it != events.end();
    ++it) {
        Event *e = *it;
        if (e->getSlot() > Time6PM
        && dynamic_cast<Meeting *>(e)) {
            message += "::No meetings after 6PM";
            result = false;
        }
        if (e->getSlot() == Time12PM
        && dynamic_cast<Meeting*>(e)){
            message += "::No meetings during lunch";
            result = false;
        }
    }
    return result;
}

void Scheduler::perform_consistency_check(string& message)
{
    if (all_meetings_are_valid(message) && …
    …
}
```

Now, we can use the tests we've written for `perform_consistency_check` to make sure that it uses `all_meetings_are_valid` to do what it used to do.

The refactoring we've done here is minor, but it is a step forward. Eventually, we'll probably want to move toward a scheme which separates message generation from checking logic. However, I want to make a point about the testing: In a typical legacy code base, I would consider it a significant improvement just to get the three tests in place that we started with.

Why? I consider those tests an improvement, because they increase the total understanding in the system. When we wrote them, they were actually a better than `perform_consistency_check` at explaining its functionality. It's sad, but true. The tests improved the system simply because they clarified some unclear logic.

The fact that tests provide this incremental benefit is very powerful. It means that we can make systems better progressively, simply by adding tests and refactoring as we can. We don't have to solve all problems to make progress; we just have to resolve to make code better every time we touch it. And as silly as it is to say, better is *better*; it's not worse. The tests that you write to just to add a little bit of understanding to a system are powerful leverage. You can use them to progressively spread the understanding from the tests to the code as you refactor, and, in the process, make your work easier, regardless of how bad things were when you started.

I do see signs of improvement in the industry. People are writing more tests and learning more about design. They are using test-driven development to develop fresh code which is easier to refactor and easier to change. However, there's still a lot of low quality code out there. It's important for each of us to know how to pull ourselves out and make things better. Tests are the most direct lever we have.

Michael Feathers is a consultant with Object Mentor and the author of 'Working Effectively with Legacy Code' (Prentice Hall 2005).

A Z C H G T X U R A E Z C F G T X U R A
U X P T X C F T X U R A G T X U R A X U
A Z C F G T X U R A A Z C F G T X U R A
U X P T X C F T X U R A G F T X U R A G
R A G T X U R A X U U X P T X C F T X U
X U R A A Z A Z C F G T C F G T X U R A

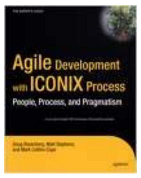
Starting next issue we will be publishing letters from readers.
Send your LETTERS to:
oveditor@objectiveviewmagazine.com

Letters on software development related issues and/or comments or discussion of articles are welcomed.

ObjectiveView Training
Application Architecture 1 or 2 days
Agile Project Management 1 day
Use Case Driven Object Modelling (Iconix Process)
* Jumpstart (work on your project) 5 days
* Public Schedule * 2 days
C # 4 days
C ++ 4 days
Ruby/Rails 3 days
AJAX 2 days
Java 4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

Product Spotlight • Enterprise Architect

Model/Code Synchronicity: The UML Holy Grail — found at last?


Agile Development with ICONIX Process
People, Process, and Pragmatism

Doug Rosenberg ***takes a look at UML tool Enterprise Architect's round-trip engineering***

Since the beginning of modeling time, the gap (sometimes a chasm) between models and code has always been problematic. Models, the argument goes, don't represent reality…only the code represents reality…therefore the model must be worthless, and we should just skip modeling and jump straight to code. Those who have used this argument to avoid modeling probably felt quite safe in doing so because nobody has ever managed to make "reverse engineering" or "round-trip engineering" a very seamless process…until now. The innocuously named "MDG Integration" product from Sparx Systems (a companion product to the Enterprise Architect modeling tool) changes the whole equation.

Bringing Mohammed to the mountain

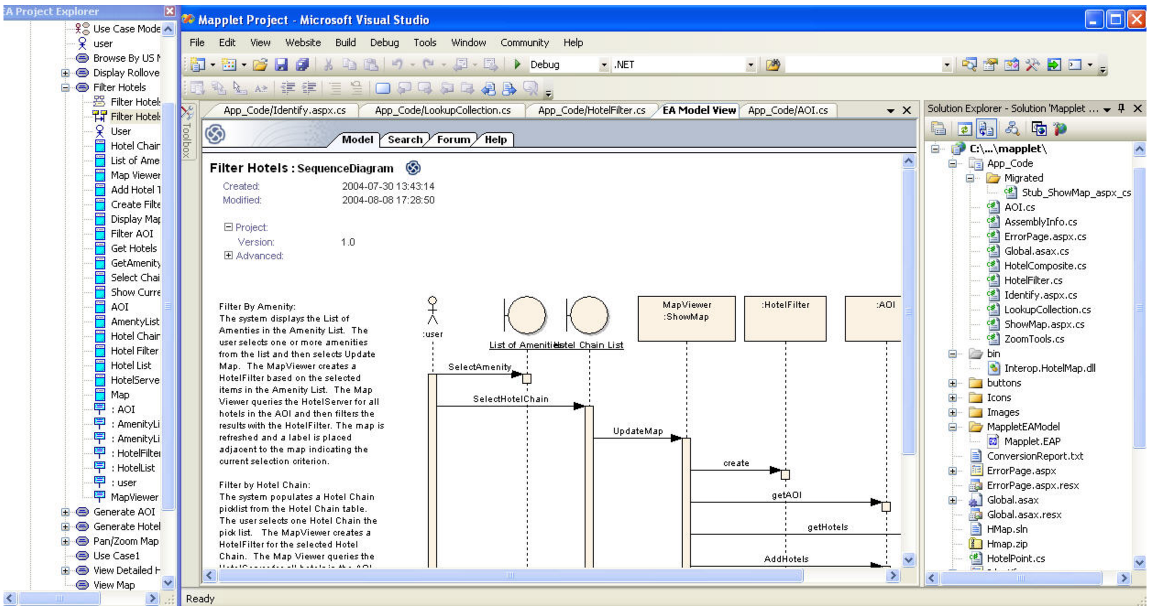
You can lead some programmers to UML, but you can't make them embrace modeling. The ever-present gap between models and code is one of the reasons for this. Modeling introduces another environment, another tools interface, another user interface to learn, and forces the programmer to leave the familiar confines of their coding environment, where they have all the comforts of home. In short, it's often viewed as a pain-in-the-ass.

But what would happen if the UML model was brought inside of the programming environment? Let's say if you could open your project, right click a menu and say something like "Attach UML Model". So you can browse your use cases, sequence diagrams, classes, etc. from within, let's say, Visual Studio (Eclipse is coming in a few months, but not ready yet). OK so far? Then let's suppose you could "hot link" a package of classes from the UML window to the source code. Nice, but not compelling yet? How's this? You can double-click on an operation on a class in the UML window and instantly browse to the source code for that method, and you can edit the code as you normally would in Visual Studio and update the UML model by right-clicking on the class and choosing "Synchronize".

Suddenly, instead of the UML model being a pain-in-the-ass, the model is actually helping you to navigate through your code, you can click to see the use cases and sequence diagrams that are using the classes you're building, and you can re-synch the models effortlessly. Suddenly your UML model is the asset which it was supposed to be all along.

Gosh, it sounds so…agile...is it really that easy?

We don't blame you for being skeptical, so we'd like to use the remainder of this article to show you an example. If you've seen our book, "Agile Development with ICONIX Process" (Apress, 2005), or if you've been to one of our open-enrollment public classes, the example might be familiar to you. It's a C# /.Net application, modeled in Enterprise Architect using ICONIX Process, and developed in Visual Studio.



Filter Hotels : SequenceDiagram

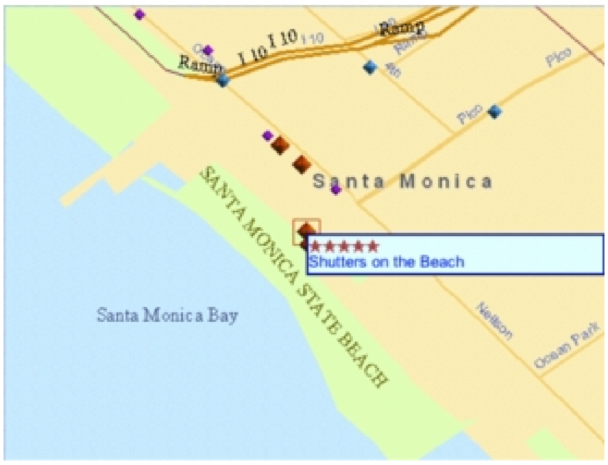
Created: 2004-07-30 13:43:14
Modified: 2004-08-08 17:28:50

Project:
Version: 1.0
Advanced:

Filter By Amenity:
The system displays the List of Amenties in the Amenity List. The user selects one or more amenities from the list and then selects Update Map. The MapViewer creates a HotelFilter based on the selected items in the Amenity List. The Map Viewer queries the HotelServer for all hotels in the AOI and then filters the results with the HotelFilter. The map is refreshed and a label is placed adjacent to the map indicating the current selection criterion.

Filter by Hotel Chain:
The system populates a Hotel Chain picklist from the Hotel Chain table. The user selects one Hotel Chain the pick list. The MapViewer creates a HotelFilter for the selected Hotel Chain. The Map Viewer queries the

The application is a map-based hotel finder (we call it the “mapplet”) that’s in production use on the VResorts.com travel website (http://smartmaps.vresorts.com), and the design, from use cases through C# code, is presented in the Agile/ICONIX book.



You can compare the use cases in the book to the running application, live on the web, and you can look at the C# code that makes it work, either in the book, or inside Visual Studio. In fact you can browse the C# code using the UML model.

A quick example of driving a use case to code

This example is borrowed from our book “Agile Development with ICONIX Process”. It shows a use case for filtering the hotel display by amenities and by hotel chain, it’s robustness diagram (you can see the sequence diagram at the top of this article) and some classes which are needed to implement the use case.

Use Case: “Filter Hotels”

Filter By Amenity:

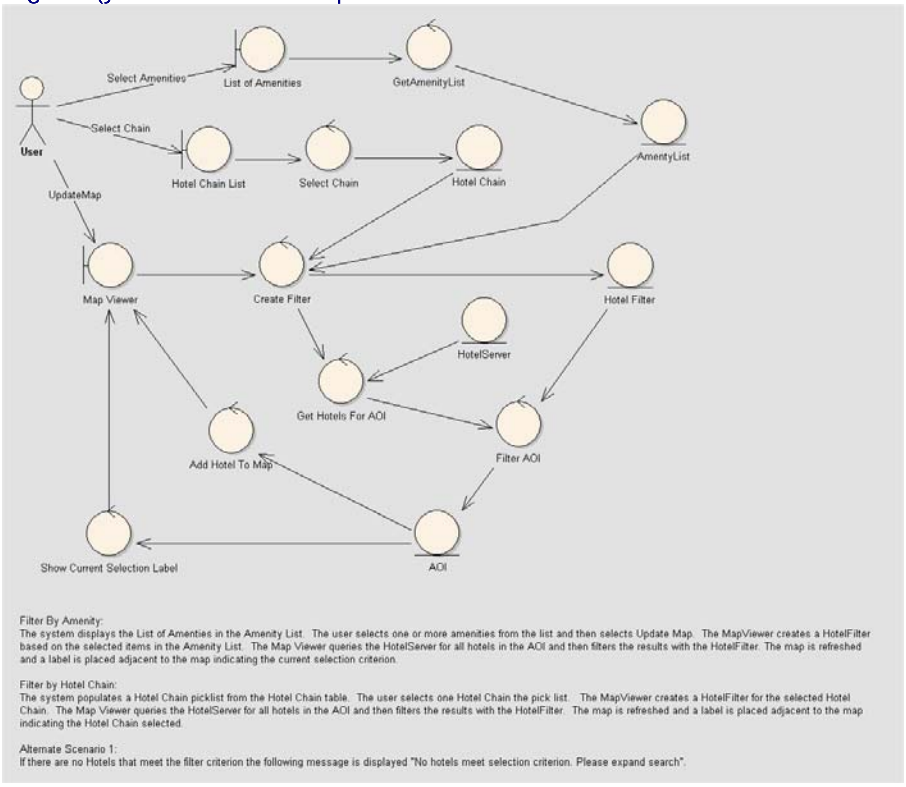
The system displays the List of Amenities in the Amenity List. The user selects one or more amenities from the list and then selects Update Map. The MapViewer creates a HotelFilter based on the selected items in the Amenity List. The MapViewer queries the HotelServer for all hotels in the AOI and then filters the results with the HotelFilter. The map is refreshed and a label is placed adjacent to the map indicating the current selection criterion.

Filter by Hotel Chain:

The system populates a Hotel Chain pick list from the Hotel Chain table. The user selects one Hotel Chain from the pick list. The MapViewer creates a HotelFilter for the selected Hotel Chain. The MapViewer queries the HotelServer for all hotels in the AOI and then filters the results with the HotelFilter. The map is refreshed and a label is placed adjacent to the map indicating the Hotel Chain selected.
Alternate Scenario 1:

If there are no Hotels that meet the filter criterion, the following message is displayed: “No hotels meet selection criterion. Please expand search.”

Here’s the robustness diagram for this use case:

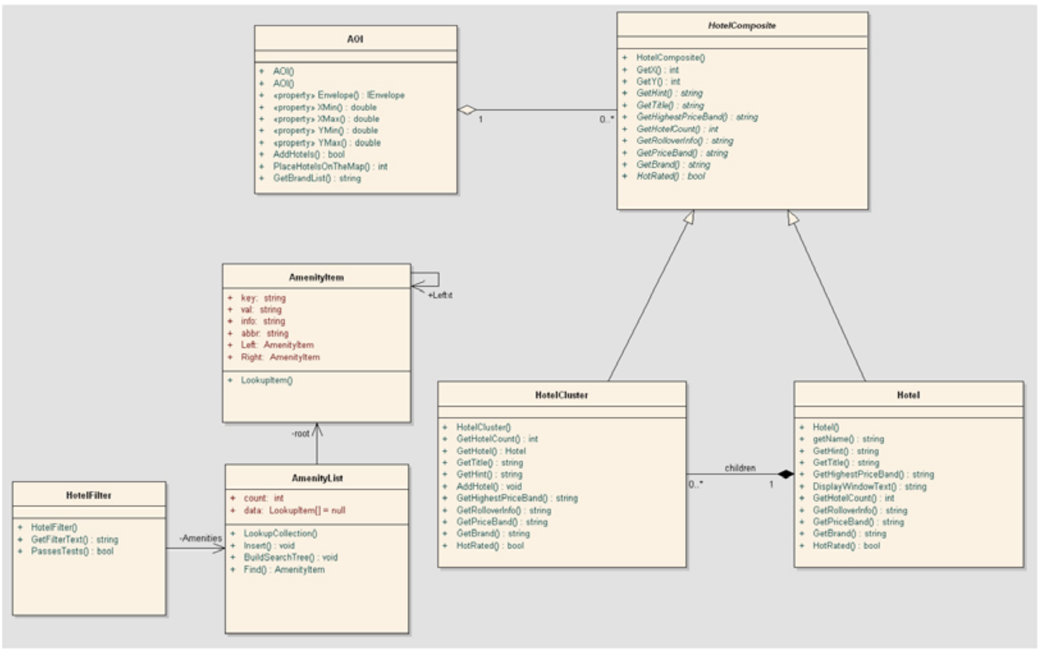


Filter By Amenity:
The system displays the List of Amenties in the Amenity List. The user selects one or more amenities from the list and then selects Update Map. The MapViewer creates a HotelFilter based on the selected items in the Amenity List. The Map Viewer queries the HotelServer for all hotels in the AOI and then filters the results with the HotelFilter. The map is refreshed and a label is placed adjacent to the map indicating the current selection criterion.

Filter by Hotel Chain:
The system populates a Hotel Chain picklist from the Hotel Chain table. The user selects one Hotel Chain the pick list. The MapViewer creates a HotelFilter for the selected Hotel Chain. The Map Viewer queries the HotelServer for all hotels in the AOI and then filters the results with the HotelFilter. The map is refreshed and a label is placed adjacent to the map indicating the Hotel Chain selected.

Alternate Scenario 1:
If there are no Hotels that meet the filter criterion the following message is displayed "No hotels meet selection criterion. Please expand search".

And here are some classes:



Finally, here's some C# code for the HotelFilter class:

```
public class HotelFilter
{
    private string AmenityFilter;
    private string HotelChainFilter;
    private string HotelChainName;
    private AmenityList Amenities;
    private char [] delimiter;

    public HotelFilter(AmenityList aAmenities,
                           string aAmenityFilter,
                           string aHotelChainFilter,
                           string aHotelChainName)
    {
        Amenities = aAmenities;
        AmenityFilter = aAmenityFilter;
        HotelChainFilter = aHotelChainFilter;
        HotelChainName = aHotelChainName;
        string delimStr = " ";
        delimiter = delimStr.ToCharArray();
    }

    public string GetFilterText ()
    {
        if (HotelChainFilter.Length > 0)
        return "Currently displaying " + HotelChainName;
        if (AmenityFilter.Length > 0)
        {
            string res = "";
            AmenityItem p;
            for (int j = 0; j < Amenities.count; j++)
            {
                p = Amenities.data [j];
                if (p.abbr != null)
                if (AmenityFilter.IndexOf (p.abbr) >= 0)
                {
                    if (res.Length > 120)
                    {
                        res = res + ", ...";
                        break;
                    }
                    if (res.Length > 0) res = res + ", ";
                    res = res + p.val;
                }
            }
            return "Currently displaying hotels with " + res;
        }
        return "";
    }

    public bool FilterHotel (string aHotelChain,
                                 string Amenity,
                                 ref string hotelAmenities)
    {
```

```
        string [] sp;
        AmenityItem p;

        if (HotelChainFilter.Length > 0)
           return HotelChainFilter.ToUpper ().
               Equals (aHotelChain.ToUpper ());
        else if (AmenityFilter.Length > 0)
        {
            sp = Amenity.Split (delimiter);
            hotelAmenities = "";
            for (int j = 0; j < sp.Length; j++)
            {
                p = Amenities.Find (sp [j]);
                if (p != null)
                    hotelAmenities = hotelAmenities + p.abbr;
            }
            for (int j = 0; j < AmenityFilter.Length; j++)
            {
                if (hotelAmenities.IndexOf
                    (AmenityFilter.Substring (j, 1)) < 0)
                    return false;
            }
        }
        return true;
    }
}
```

Nice and simple so far. Any child could do it. As one of my old Electrical Engineering professors used to say (I think we were studying Maxwell's Laws at the time): "*It's intuitively obvious to the casual observer*".

But…here's the six million dollar question: how do we keep the model and the code synchronized over the lifetime of the project?

Five Simple Steps to Modeling Nirvana – without chanting OMMMMM

We wrote a whole chapter in **Agile Development with ICONIX Process** about how to synchronize models and code, and the reasons why it's important. It's still just as important, but the folks at Sparx Systems have obsoleted the "how-to" from that chapter. Now it's absurdly simple.

So simple that an old tool-builder like me wonders "why the heck didn't I think of that?"

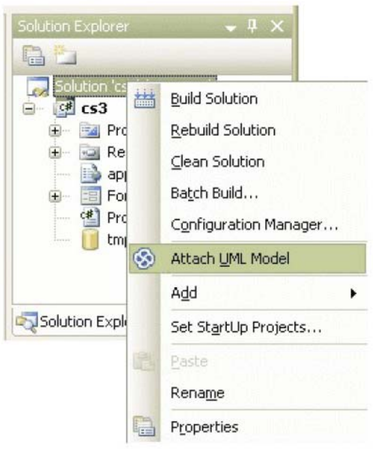
Here's how it works:

1. Connect UML model to VS Project
2. Link package in model to VS project
3. Browse source code by clicking on operations on classes
4. Edit source code in VS
5. Right-click on class and choose Code Services -> Synchronize

Let's take these one at a time:

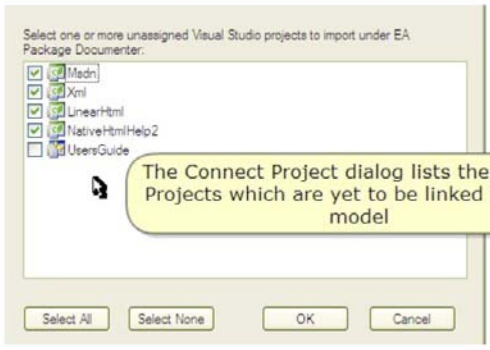
Connect the UML Model to the Visual Studio Project

When MDG Integration is installed, Visual Studio grows another brain…whoops, I mean it gains the ability to have a UML model attached to it. You do this by selecting "Attach UML Model" from the Visual Studio Solution Explorer:



Link Package in UML model to the Visual Studio Project

Visual Studio and Enterprise Architect are advised that the classes within a certain package should be hot-linked to source code files in VS.


Select one or more unassigned Visual Studio projects to import under EA Package Documenter:

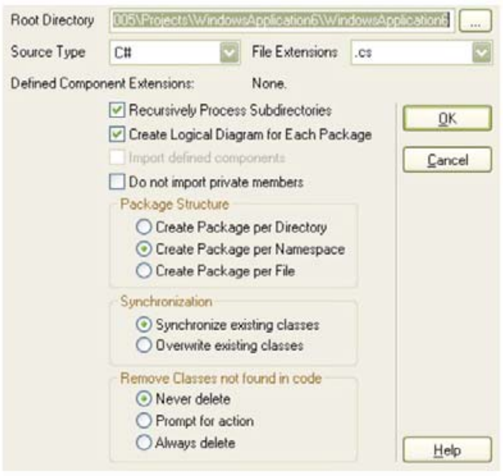
- [✓] Msdn
- [✓] Xml
- [✓] LinearHtml
- [✓] NativeHtmlHelp2
- [ ] UsersGuide

The Connect Project dialog lists the Solution Projects which are yet to be linked to a UML model

Select All | Select None | OK | Cancel

Clicking on Ok will then reverse engineer the project(s).

Enterprise Architect then reverse engineers the code for you, automagically.


Root Directory: 005\Projects\WindowsApplication6\WindowsApplication6

Source Type: C# | File Extensions: .cs

Defined Component Extensions: None.

- [✓] Recursively Process Subdirectories
- [✓] Create Logical Diagram for Each Package
- [ ] Import defined components
- [ ] Do not import private members

Package Structure
- ○ Create Package per Directory
- ● Create Package per Namespace
- ○ Create Package per File

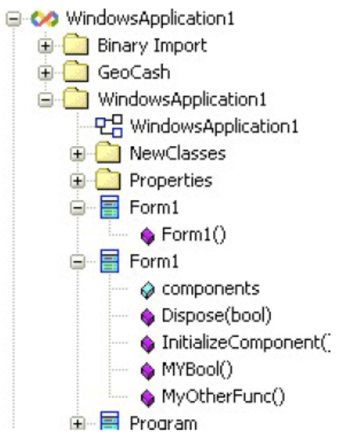
Synchronization
- ● Synchronize existing classes
- ○ Overwrite existing classes

Remove Classes not found in code
- ● Never delete
- ○ Prompt for action
- ○ Always delete

OK | Cancel | Help

Browse source code by clicking on operations on classes

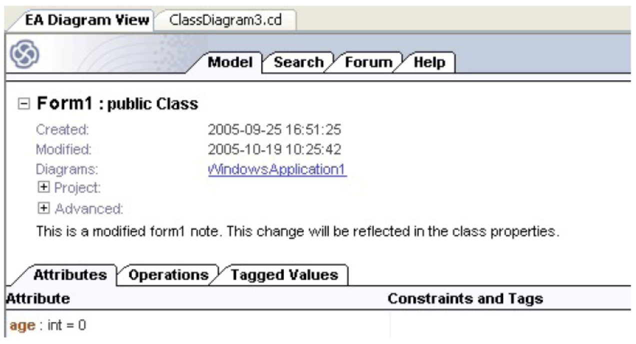
Your UML model should provide high-level guidance and help you to understand how the code is structured. Hopefully, modeling the system in UML resulted in cleaner, better organized code. What could be a more natural way to leverage the investment in the UML model than to simply click an operation on a class in the UML browser window, and have Visual Studio pop up the associated code?



If there's a more natural way to browse your source code and maintain a high-level view of the code's organization, we haven't seen it.

Edit source code in Visual Studio

Actually you can either edit the source in Visual Studio, or edit the operations in the UML browser…

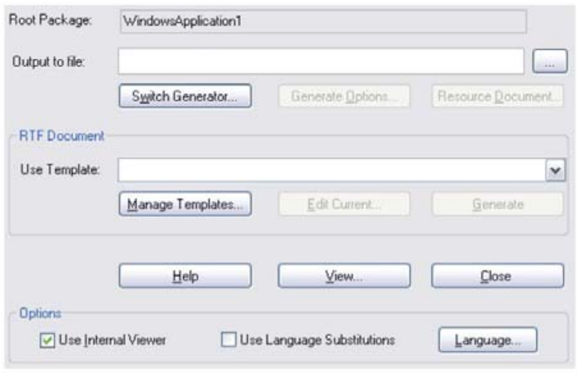


Form1 : public Class

Created: 2005-09-25 16:51:25
Modified: 2005-10-19 10:25:42
Diagrams: WindowsApplication1
Project:
Advanced:

This is a modified form1 note. This change will be reflected in the class properties.

Attribute	Constraints and Tags
age : int = 0	


Root Package: WindowsApplication1

Output to file:

☑ Use Internal Viewer ☐ Use Language Substitutions

Right-click on a class and choose Code Services -> Synchronize

You've added some detail (revised parameter list, maybe) to a class in EA, or you've edited the code in Visual Studio. Now you need to confront the ever-so-painful task of synchronizing the model and the code. Ever the good agilist, you've read Scott Ambler's writings on the subject and for years have practiced "Update only when it hurts" because this task has been so difficult. Then your models and your code have gotten out of synch and you've gradually disregarded the models. But, you think to yourself, what if the UML and the code were really one and the same. Just two representations of the same thing…the UML classes and the C# classes united by their inner-oneness and sameness of purpose. What would it be like? So you right click on the class you just modified and select "Code-Services -> Synchronize". Ahhhhhh. Feel the waves of bliss sweeping over your keyboard. You feel your inner self floating out of your body and looking down at the room. Nirvana at last.

Would you like some project documentation to go with that code?

Oh, we almost forgot. Do you have annoying managers who ask you to document your stuff? EA handles that for you, too…in your choice of RTF or HTML.

Yes! The documentation gets automagically done for you! Life is good.

For more info…

You can spend two days (with me) working through the entire example "mapplet" project from use cases through C# code using Enterprise Architect and Visual Studio at our open enrollment public training workshops: (http://www.iconixsw.com/EA/PublicClasses.html).

Some of the material in this article was borrowed from a book I wrote with Matt Stephens and Mark Collins-Cope called **"Agile Development with ICONIX Process"** (http://www.softwarereality.com/AgileDevelopment.jsp)

Additional material was borrowed from a Sparx Systems white paper on MDG Integration called "MDG Integration for Visual Studio 2005" (available from http://www.sparxsystems.com/products/mdg_integrate.html) You can use the "mapplet" to find hotels in any city in the US on the VResorts.com travel website (http://smartmaps.vresorts.com).

Oh, I almost forgot. You can buy a nice little bundle which includes a copy of EA Corporate Edition, a copy of MDG Integration for Visual Studio, and two multimedia tutorials: "Enterprise Architect for Power Users" and "Mastering UML with Enterprise Architect and the ICONIX Process" from the ICONIX website. We call it "EA PowerPack VS.Net 2005" Just click this link: http://www.iconixsw.com/EA/PowerPack.html

Doug Rosenberg is President of Iconix Software.

Opinion • Kevlin Henney • Getting Over the Waterfall

Sequential development, as typified by the waterfall style of development, is the common whipping boy of anyone supporting a more iterative approach to development. However, in spite of the increased popularity of iterative and incremental approaches over the last decade, waterfalls still reflect the most popular formal approach to managing projects -- rivalled only in popularity by the most popular informal approaches: unmanaged and mismanaged. Neither the metaphor of a stream of water crashing inexorably and forcefully onto the rocks beneath nor the impressively poor track record of the approach appear to have prevented the continued popularity of this way of development.

However, it is all too easy to mock the waterfall approach without understanding its motivation and attraction; without such understanding, advocacy and adoption of agile development can be superficial, dogmatic and propaganda based, which misses much of the deeper rationale and actual value of agile approaches. It is not my intention to defend, let alone advocate, sequential development as a general approach, but any honest criticism of it in favour of something else needs to begin with a more even-handed appraisal.

The organising principle of sequential development can be summarised quite simply: strictly align development activities with phases in development. This principle is tidy, clear and easy to explain: given a number of different activities that occur in development, combined with the recognition that a development effort follows a lifecycle that can be characterised differently at each point in time, define each phase of time in terms of an activity and run the phases in sequence. Such a model of development is easy to lay out and present visually. At any given point in time it is clear what activity is being performed. The sequence of activities seems reasonably organised so that problem discovery and reasoning comes before problem solution and execution, which in turn precedes final confirmation that the right thing was built right before proceeding onto deployment.

This model is entirely logical if you make some important assumptions: the problem being addressed is stable and fully understood by all parties (in the same way); the approach to the solution is well defined and fully understood by developers; the technology to be used is fully understood by developers and its use is guaranteed to be free of surprises. If you can guarantee those assumptions, you can ride the waterfall and keep your head above water. But if you can't, the raft of assumptions quickly unravels into something far less watertight.

Software development is typically a multi-variable problem with few guaranteed constants. Treating a dynamic situation with a static plan is a recipe in risk, easily upset by the slightest change. The waterfall approach has the laudable intent of attempting to derisk unknowns by exploring the problem in detail at the start of the lifecycle, with a final check on things at the end. Unfortunately, in practice the effect can be quite different: instead of derisking at the earliest possible opportunity, this approach pushes and accumulates risk towards the back end of the lifecycle. Significant decisions are made at the start, the point of least knowledge of what is involved in developing a system, both in terms of tangible requirements and technical requirements.

At the point of greatest knowledge - the end - the chance for effective change has all but vanished. This is not to say that there is no merit in emphasising problem discovery and architectural foundation early in development, just that these activities are not the exclusive preserve of the front end of the lifecycle and their results are not set in stone. It is precisely because of the uncertainty surrounding these issues that you want to start them early. The reason you engage in a development lifecycle with repeating feedback loops is to give yourself the opportunity to clarify and converge as you go, replanning and redesigning as you learn rather than being caught off guard when you're supposed to be done.

It is perhaps telling that, in response to the publicity surrounding agile development, some advocates of waterfall-style development have favoured a rebranding of sequential process models under the heading "plan-driven development". The intended implication being that agile development is unplanned. However, a more accurate reading is that an agile development lifecycle is not driven by *a* plan, and it turns out this is not wrong: agile approaches tend to be highly planned or, more accurately, "planning driven", but not "plan driven" -- a subtle but important distinction. For the reasons examined, being driven a plan is a fragile and risky approach when what are assumed to be constants are actually variables. The role of the term "plan" in agile processes is that of a verb rather than a noun: planning is an activity that is pervasive and continuous, not a static artefact produced at an early stage as input to later phases. It is planning rather than the plan that takes centre stage in agile development. The metaphorical entailment of "plan-driven development" is perhaps closer to the idea of "planned economy" than anything else -- and makes the majestic, natural imagery of a waterfall somehow more attractive.

That said, it would be disingenuous to say that plan-driven approaches exclude the possibility of modification or revision to an initial plan in the light of new information, changed circumstances and measures of progress. The attitude to such changes, however, is that they are corrections, irritations and exceptions rather than the normal state of affairs. In spite of much published wisdom to the contrary, estimates are still often treated as predictions rather than as forecasts. Some amount of change and uncertainty is acknowledged, but it is a grudging and partial acceptance that does not inform the overall mindset or nature of the development process.

Given the typical human response to change and uncertainty, this is hardly surprising: this trait is within each of us to a greater or lesser extent.

All this suggests that the management of incremental development, particularly processes intended to be streamlined and responsive, is not necessarily the path of least resistance. It sounds hard: it is continuous; it is constantly buffeted by change; it can come into conflict with human nature. The apparent alternative of following a plan that lays everything out in a predictive and tidy sequence, abstracting out interference from change, discovery and human nature, does indeed look simpler. But appearances can be deceptive. Quoting from the last column ("Down on the Upside", ObjectiveView #9): "A good abstraction is one that allows us to develop a piece of software more effectively; a poor abstraction is one that misleads us by omitting or including the wrong kind of detail".

In this case, the appeal of the highly planned model comes from abstracting away some fairly critical details -- details that if taken into account would, by necessity, change the needs and nature of the development process. So yes, the management of an agile process sounds hard and it is, but that's a property of software development rather than specifically of agile approaches; using a mismatched model of software development makes the challenge of management even harder. Where jumping a waterfall requires a leap of faith, agility is more openly feedback driven and evidence based, using smaller steps to ensure footing and gauge the next step.

A Z C H G T X U R A E Z C F G T X U R A
U X P T X C F T X U R A G T X U R A X U
A Z C F G T X U R A A Z C F G T X U R A
U X P T X C F T X U R A G F T X U R A G
R A G T X U R A X U U X P T X C F T X U
X U R A A Z A Z C F G T C F G T X U R A

Starting next issue we will be publishing letters from readers.
Send your LETTERS to:

oveditor@objectiveviewmagazine.com

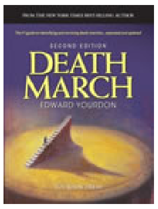
Letters on software development related issues and/or comments or discussion of articles are welcomed.

Subscribe to ObjectiveView
email: objective.view@objectiveviewmagazine.com with subject: subscribe

Distribute ObjectiveView
It's easy. You link to ObjectiveView using our linkgif, and your logo will appear on *all* copies of the magazine.
Contact oveditor@objectiveviewmagazine.com for more info.

Historical Perspectives • Ed Yourdon • Structured Analysis

Ed Yourdon ***takes a retrospective look at Structured Analysis …***



A long time ago, in a galaxy far, far away, a growing number of software engineers began worrying about an emerging software "crisis." Computers were being used in more and more safety-critical applications (including missile guidance systems, and process control systems in manufacturing plants); computer systems were getting larger and more complex, with software playing an increasingly dominant role.

And the number of bugs, project failures, schedule slippages, and budget over-runs was becoming increasingly embarrassing. This may sound like today's news, but it was actually discussed as far back as 1969, when NATO (yes, the very same North Atlantic Treaty Organization) sponsored a historic conference on "software engineering."

The widespread consensus at the time was that the nascent software industry was faced with a *programming* problem: *programs* were time-consuming and expensive to code, *programs* were difficult to test, and *programs* had bugs.

Fortunately, a solution was at hand: by the late 1960s, the industry was already experimenting with a fairly radical concept known as "structured programming" – articulated by Edsger Dijkstra, and based on theoretical work published by two Italian computer scientists (Bohm and Jacopini).

Alas, structured programming didn't make the problem go away: we ended up with well coded (and usually GOTO-less) programs that still suffered from weak architectures, subtle interdependencies, and expensive maintenance budgets.

By the mid-1970s, a new solution to the crisis emerged: *structured design*, based on formative work by Larry Constantine, and an influential 1974 *IBM Systems Journal* article by Constantine, Myers, and Stevens. A series of structured design textbooks were published in the mid-1970s, one of which (*Structured Design*, by Larry Constantine and yours truly) had the intriguing experience of being discussed in its 30th year of publication at the 2005 OOPSLA conference in San Diego.

Alas, structured design didn't solve the problem either: at best, it merely produced brilliant solutions to the wrong problem. Or, to put it more directly, it didn't really address a more fundamental problem: if we don't understand the user's requirement for an automated system, it doesn't really matter how good our design, and how well-structured our code, turns out to be. At the end of the day, we still haven't solved the user's problem.

Enter *structured analysis*. It turns out that, even in the early-to-mid 1970s, some very smart, and very clever, people at a Boston-based company called Softech had been thinking about better ways of identifying and documenting user requirements. Their approach, known as SADT, was quite powerful, and enjoyed some impressive successes; however, it was carefully guarded as "intellectual property" by Softech, and was generally provided only to large IT organizations by means of a very expensive license. Most of the civilized world was blissfully unaware of its existence.

Meanwhile, several of the people who had been heavily involved in structured design – including people like Tom DeMarco, Chris Gane, Trish Sarson, and several colleagues of mine at YOURDON, Inc. – were thinking about extending the concepts and tools of structured design into the world of systems analysis, requirements modeling, and requirements documentation. As mentioned above, we were already seeing, first-hand in several consulting projects, that "programming" wasn't really the problem, nor was "design." The real problem was trying to communicate with intelligent, but non-technical, business people to understand their requirements in a way that we could "feed back" to them for confirmation.

At the time, the business of "understanding" a user's requirements typically involved interviews, surveys, brainstorming sessions, and various other forms of person-to-person communication. And that still remains true, to a surprising extent, 30 years later: a system analyst will often sit across the table from a business person and say, "Let me interview you to find out what you do, and what you'd like to see in the new system we're going to build for you."

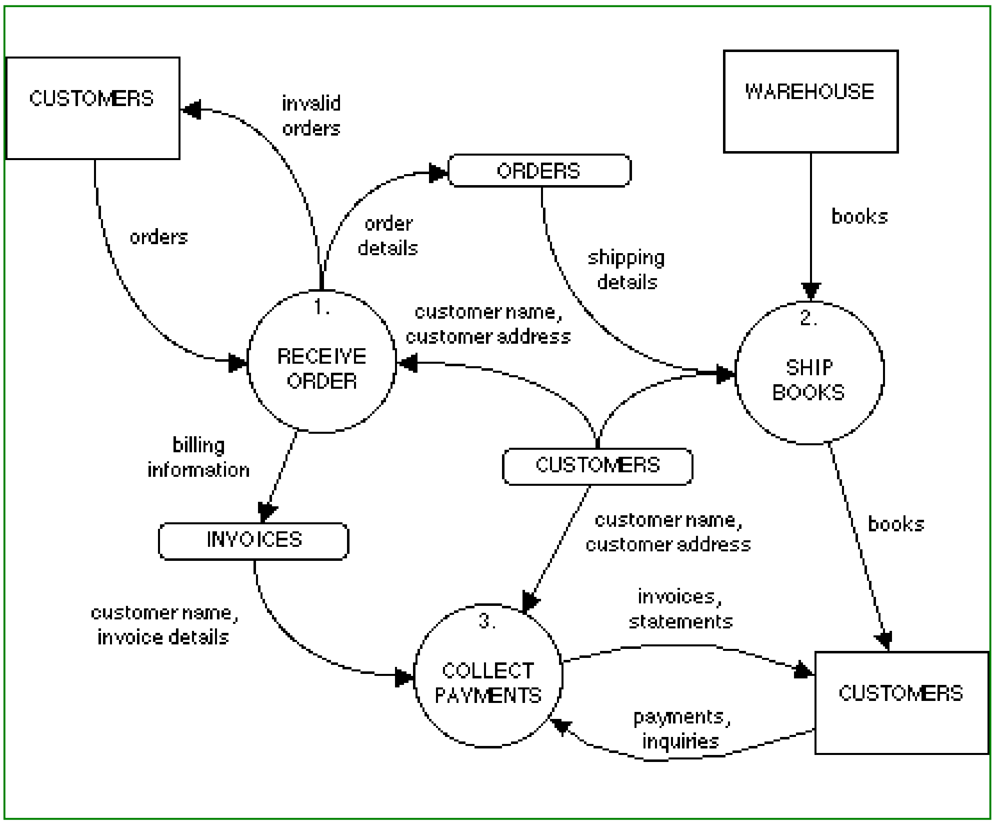
What we didn't have at the time – and we *do* have today, in a large number of situations – is the ability to prototype pieces of a "straw man" system very quickly, for immediate feedback from the user. Remember: all of this was taking place shortly after the invention of fire and electricity: we didn't have PC's, we didn't have Excel or Access or Java or Visual Basic. We didn't have fourth generation languages of any kind, and we didn't even have word processing tools to edit the massive, monolithic "Victorian novel" specifications that usually resulted from the series of interviewing sessions.

So the interviews and surveys and specification-writing process went on for weeks, months, and sometimes even years; the result was typically a massive pile of paper, hand-typed on ancient electric typewriters, and delivered to the business users *en masse* for their review and approval. In the worst case, the users had completely lost interest in the system; in the best case, the users desperately tried to read, understand, and provide feedback on an overwhelming document.

In the best case, the specification document would be revised and delivered to the users for another round of reviews and approvals – until they agreed that it was "right," or simply ran out of energy to keep making corrections. In the worst case, the systems analysts discovered that the cost of changing the document was too much to bear: I consulted on more than one project in the 1970s where senior management decreed that any changes to the specs would be incorporated in the software – because code was easier to change than English documents!

So, if there's one good thing that structured analysis did for the software industry, it was encouraging both analysts and end-users to break out of the "text-only" mode of describing requirements. In its place, we substituted the notion of graphical (pictorial) *models* of a system. And we insisted that such models be organized and presented to the users in a top-down, hierarchical, partitioned fashion – so they could review an overview of the entire system on one page, or any lower-level set of requirements in isolation from the others.

That concept – essentially that a picture is better than a thousand words of requirements documentation – has survived for 30 years, even if the notational details (and the organizing principles of that notation) has changed. In the 1970s, we envisioned the key graphical model of structured analysis as a *data flow diagram*, of which this is a simple example:



Today, it's more likely that we would present an *object-oriented* model of requirements to a user, based on the notation of popular methodologies such as UML. But it would still be a picture, and we would probably discover that business users would prefer to look at a simple picture than a thousand pages of mind-numbing text.

Of course, what they would *really* like to look at is a "real" system, with real input forms and real output displays (whether in the form of a Web page or a printed report). And with today's prototyping tools, we can obviously do that; but if the system is large enough and complex enough, there is still great benefit in providing a pictorial overview of various aspects of the system – so that the user can focus on the "big picture" without being overwhelmed by the details, and also so the user can focus on a particular *perspective* of the system.

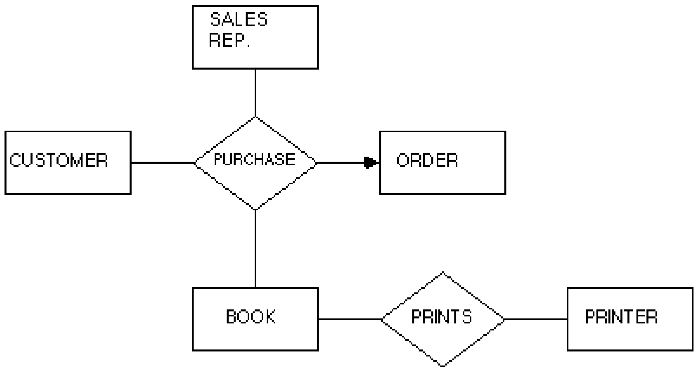
The data flow diagram (or DFD) shown above emphasizes a *functional* perspective – i.e., it draws the reader's attention to the fact that the proposed system (in this case, a publishing system) has three major functions represented by the three "bubbles." Each of the three bubbles can then be "partitioned" into a separate DFD showing the functional decomposition into lower-level sub-functions. And the partitioning process can continue as long as necessary; early structured analysis practitioners often spoke of "bubbling on down to the bottom" of a large complex set of requirements.

The DFD also provides a pictorial reference to three databases (customers, orders, and invoices), but it doesn't say anything about the relationship between those databases. And it shows something about the flow of data between the various functions, without bothering to say anything about details of the data elements flowing back and forth.

It soon became apparent that in many business-oriented systems, the *data relationships* were as important as the functional, if not even more important. In any case, the data relationships represented a different *perspective* on the requirements, and often demanded a separate series of discussions and interviews with the business user. This was accomplished with a separate modeling notion known as an *entity-relationship* diagram (or ERD), an example of which is shown here:

The ERD shows nothing about the functions being performed by the system, but instead concentrates on the *entities* (or what we would now consider to be the attributes of an object, without the associated methods).

Additional graphical models (e.g., *state transition diagrams*, or STDs)) were eventually



added to the core structured analysis notation, and detailed documentation techniques (known as "structured English mini-specs") were introduced to describe the business requirements of "bottom-level bubbles" in the DFD, as well as the detailed description of attributes in the ERD, and the data flows between the DFD bubbles.

It all worked reasonably well, and the whole approach gained increasing popularity throughout the 1980s. But as with any new idea, problems eventually emerged and grew more and more annoying:

- While vastly superior to the monolithic Victorian novels they replaced, the combination of DFDs, ERDs, STDs, detailed data dictionaries, and mini-specs grew overwhelming for large systems; for some large government systems, there were as many as 10,000 "bubbles" in the bottom-level DFDs. Remember, all of this was being done manually in the era before PCs and graphical modeling tools like Visio. IT organizations began to be overwhelmed by the burden of drafting all of the diagrams.
- Because the structured analysis models *were* created and maintained manually, it became more and more tedious to modify them as requirements changed throughout the project. As a result, we were cursed with the same problem we experienced with Victorian novel specifications: the structured analysis models would only be created once, and all of the "new" requirements could only be found in the code that resulted from the development work.
- Because the structured analysis model involved different kinds of diagrams (DFD, ERD, and STD), along with various kinds of low-level documentation (data dictionaries and mini-specs), consistency checking *between* the models became more and more cumbersome. By the late 1980s and early 1990s, this problem was ameliorated somewhat by so-called Computer-Aided Software Engineeering (CASE) tools, but some organizations had already given up.
- Like many other methodologies and good ideas in the software field, structured analysis was sometimes over-sold as the panacea that would cure all ills. The ultimate example of this problem came with the CASE tools – which, in their early incarnations, often cost as much as $10,000 per software engineer. CASE vendors sometimes sold their products as "model-to-code" engines that would "automatically" convert user requirements into fully developed, working code. Alas, the reality never lived up to the promise.
- The separation of *functions* and *data* (as represented by the DFD and ERD) proved to be a fatal flaw in many projects – for, among other things, it meant that the work was divided into two separate groups of IT professionals. One group tended to worry about the functions (which the business users were sometimes equally interested in, but sometimes less so); and the other group worrying about the nuances of data relationships (with occasional forays into premature discussions of *normalizing* the entities they had created), with little or no concern about the functions.

This "functions-vs-data" schizophrenia was eventually resolved with the introduction of an object-oriented (OO) approach to requirements analysis in the early 1990s, after OO had gained a "critical mass" of popularity as a programming and design approach. This eventually led to the 1995 "unified method" representing the convergence of OOA methodologies formulated by Grady Booch, James Rumbaugh, and Ivar Jacobsen; and the notation for that unified methodology was eventually formalized into today's Unified Modeling Language, or UML.

Meanwhile, CASE tools faded away, and were replaced by Integrated Development Environments (IDEs). And the cost of both the workstations and the tools dropped dramatically, to the point where the tools often cost less than $1,000 (and can sometimes be acquired as open-source products, for free), and will run on the PC that software engineers already have on their desktop.

Interestingly, while some things have changed drastically in the past 35-40 years, some things have not changed at all. We still have projects that are late, over budget, full of bugs, and sources of intense frustration by the business users. All too often, business users and/or IT professionals still document their requirements in Victorian novel tomes, and they refuse to change the requirements documentation after their first iteration. And IT professionals are still subject to hype and over-selling: they will often believe a vendor's frenzied pitch about prototyping, without remembering that a prototype doesn't provide the detailed documentation that is often necessary for final acceptance and ongoing maintenance of a system.

Equally interesting, structured analysis never disappeared completely from the computer science/software engineering curricula of universities around the year; I still get frantic questions every year from desperate students about to face a final exam on the subject.

Not only that, but structured analysis has enjoyed a new birth of popularity: organizations who are focusing on *business process improvement* may or may not be interested in automating/ computerizing an existing business process; but in order to understand whether it *can* be improved, they need a simple way of modeling the existing process, as well as the "to-be" process. Though object-oriented analysis and UML could certainly be used for that activity, I've been surprised to see how many organizations have "re-discovered" data flow diagrams and the other structured analysis modeling notations.

Whether structured analysis will still be around 30 years from now is something I won't even attempt to predict. After all, I was one of those people who firmly believed that COBOL would vanish from the earth by 1990 … and it will probably still be here long after I'm dead. As for structured analysis: well, perhaps it will provide a modestly useful activity to keep me out of trouble after I've become a doddering old retiree.

There are, after all, worse fates. I could still be writing COBOL programs…

Ed Yourdon is an internationally-recognized computer consultant, as well as the author of more than two dozen books, including Byte Wars, Managing High-Intensity Internet Projects, Death March, Rise and Resurrection of the American Programmer, and Decline and Fall of the American Programmer. His latest book, Outsource: competing in the global productivity race, discusses both current and future trends in offshore outsourcing, and provides practical strategies for individuals, small businesses, and the nation to cope with this unstoppable tidal wave.

A Z C H G T X U R A E Z C F G T X U R A
U X P T X C F T X U R A G T X U R A X U
A Z C F G T X U R A A Z C F G T X U R A
U X P T X C F T X U R A G F T X U R A G
R A G T X U R A X U U X P T X C F T X U
X U R A A Z A Z C F G T C F G T X U R A

Starting next issue we will be publishing letters from readers.
Send your LETTERS to:

oveditor@objectiveviewmagazine.com

Letters on software development related issues and/or comments or discussion of articles are welcomed.

Subscribe to ObjectiveView
email: objective.view@objectiveviewmagazine.com with subject: subscribe

Distribute ObjectiveView
It's easy. You link to ObjectiveView using our linkgif, and your logo will appear on *all* copies of the magazine. Contact oveditor@objectiveviewmagazine.com for more info.

Treating Tests as Software

Kevin P. Taylor ***explains why we must treat test code with the same respect as other source code …***

On a recent project, I was pair programming with a talented and experienced programmer. Talented and Experienced was relatively new to the project and this was my first opportunity to pair with him. When I was in the driver's seat, I noticed that Talented and Experienced was fidgeting and his obvious discomfort was growing as I continued typing. In the test that we were working on, I spotted a variable misleadingly named "tax." I opened Eclipse's rename dialog box and typed in "expectedTaxRate." Talented and Experienced exploded, "It is just a test! Let's spend our time on the real code."

A few weeks later, I attended an OpenSpace discussion session entitled "Are Tests Software?" Didn't all agile programmers consider unit tests an integral part of their software? In fact, I would argue that on an agile development team with comprehensive unit tests in place, tests must be treated with more care than the functional code they protect. When a team has a robust and flexible test harness around its functional code base, the team is liberated to refactor that code with confidence, knowing the tests will complain if the functional code breaks.

But, what gives confidence to developers when they refactor or modify the tests themselves? Since writing unit tests against unit tests would lead nowhere, instead, great care must be taken when tests are refactored or updated to handle new or changing functionality. In this article, we'll review current views of what software quality consists of and how these characteristics are reflected in unit tests. Then, we'll discuss specific things you can do to improve the quality of your unit tests.

Software Quality

Software quality is an elusive concept. Is software quality a measurement of how closely software fulfills its specification? Is it how well the software meets end user needs? Or, should software quality be defined as how well it is designed and written, i.e. how readable and maintainable the source code is?

Software quality becomes even more difficult to define when we consider the imbalance in skill and experience amongst different software teams and the varying external pressures different teams cope with. What is considered acceptable code in one shop may be considered defect-ridden spaghetti code in another. The first shop may be primarily concerned with quickly delivering adequately correct Web applications and may have a high tolerance for defects. The other shop may be working on safety-sensitive systems, so would certainly be much less tolerant of defects. However, they may still have very low quality standards regarding code design and maintainability.

Unit Testing Tips - Summary

- setUp() is for setting up
- Use one TestCase per fixture scenario
- Write reusable fixture logic
- Keep assertions simple
- Use meaningful identifiers
- Pair program
- Test your tests
- Run > 100 tests per second
- A test must never affect another test's outcome
- Ensure test failures are easy to debug
- Use stubs to temporarily help test-drive new code
- Use fakes liberally to isolate tests
- Use mocks sparingly to assert interactions

ISO 9126 Standard

ISO 9126 is an attempt to standardize the definition of software quality. According to ISO 9126, software can be evaluated against each of six quality characteristics.

Functionality

The *functionality* of software refers to what software does rather than how it does it. For unit tests, this reflects how accurately and completely the tests measure the correctness of the functional code. Do the unit tests test all the scenarios and execution paths? How thoroughly do they assert the expected behavior of the functional code?

Reliability

The *reliability* of software is associated with its capability to maintain its specified level of performance under specified conditions. *Maturity*, *fault tolerance*, and *recoverability* are the three elements of reliability. Unit test reliability is primarily a measure of its maturity, i.e. the presence or absence of logic and runtime errors.

Usability

The *usability* of software consists of three sub-characteristics. The first, *understandability*, is concerned with how easy it is to determine the purpose of software and whether the software is applicable to our needs. In other words, should we use it? The next is concerned with how *learnable* software is. Finally, *operability* is a measure of how easy is it to actually use software: What is the level of effort required? Usability of unit tests is of paramount importance to developers. Being able to quickly find the appropriate unit test, understand what scenario is being tested, and modify the unit test is at the heart of test quality.

Efficiency

The *efficiency* of software can be evaluated by considering how fast it is (CPU time and I/O throughput rates) and how resource intensive it is (memory, CPUs, socket connections). Unit test efficiency manifests itself by how fast the test runs and how isolated it is from external resources such as file systems and network connections.

Portability

The *portability* of software refers to how well it operates in different environments, such as different operating systems. A unit test's portability is primarily concerned with how well it runs on all targeted operating systems and how well it runs in different execution environments, such as via an IDE or a continuous integration tool such as CruiseControl or AntHill.

Maintainability

Software *maintainability* characterizes the design and clarity of the software's source code. Maintainability directly affects developers who must analyze and modify software. Indirectly, maintainability affects the owners and users of the software by influencing the costs involved in enhancing the software. Since unit tests are used by developers in source code format, maintainability of unit tests is logically equivalent to the usability characteristic of unit tests that we already looked at.

Treat Your Tests Well

According to Kent Beck, Ron Jeffries coined the phrase "Clean code that works" to describe the goal of test-driven development (TDD). Jeffries' concise description of quality code is not only applicable to your functional code base. Let's see how it can help guide us toward higher quality test code. Turning your unit tests into clean code that works requires keeping them simple and intentional. But, according to Jeffries, having clean code is only half the solution: code must also work. For unit tests, this means they must be correct, sufficiently complete, fast enough, independent, and isolated.

Simple

Simple unit tests keep fixture setup as simple as possible by only setting up a single set of closely related fixtures per test case. Use the setUp method of the test case and never use conditionals to get clever with the set up. Keep it simple. When you feel the need to add a conditional statement while setting up your fixtures, instead create a new test case with its own setUp method and fixtures.

Use an ObjectMother or Builder to remove complex, duplicate fixture code from your tests. This will make the code easier to understand and reduce the chance of errors.

Simple unit tests keep assertions and expectations as simple as possible while proving that the functional code is correct. For example, if asserting a value is null, use Assert.assertNull(value) instead of Assert.assertEquals(null, value).

Simple unit tests don't contain *equivalent duplication*. Equivalent duplication is duplication that is not coincidental. For instance, if a test is expecting 10 line items on an order and also a quantity of 10 widgets, the 10 is coincidental. It should be represented by two different identifiers. However, if 5:00 P.M. is asserted as the expected order cutoff time in multiple tests, 5:00 P.M. is equivalent duplication and should be represented by a single identifier.

A common source of duplication in unit tests is caused by overuse of JUnit's built-in assert methods. Use the *extract method* refactoring to pull reusable assertion logic into custom methods. You should have plenty of custom asserts in your unit tests, such as assertCollectionEquals, assertDateBefore, assertBeforeOrderCutoff(Date), etc.

Intentional

Along with being simple, clean unit tests must communicate to the reader what is important for her to know about the tests. Intentional unit tests are those that have scenarios that are easy to identify, have fixtures and assertions that are easy to understand, and clearly document the expected behavior of the functional software.

To make unit tests intentional, use the same techniques we have all become accustomed to applying to functional code. Use identifiers that are clear and meaningful. Use refactorings such as extract method, to document business logic. Use comments sparingly. Use your team's coding standards and common accepted idioms for the language that you are working in.

```
import junit.framework.TestCase;

public class HelloWorldTest
        extends TestCase {

    protected void setUp()
            throws Exception {
        super.setUp();
    }

    public void test_sayIt() {
        Person person = null;
        HelloWorld helloWorld = new HelloWorld(person);
        assertTrue(
            "Hello!".equals(
                helloWorld.sayIt()));
        assertTrue(person == helloWorld.getPerson());
    }

    public void test_sayIt_withName() {
        Person person = new Person();
        person.setName("Kevin");
        HelloWorld helloWorld = new HelloWorld(person);
        helloWorld = new HelloWorld(person);
        assertTrue(
            "Hello! Kevin is 0".equals(
                helloWorld.sayIt()));
        assertTrue(person == helloWorld.getPerson());
    }
    public void test_sayIt_withNameAndAge() {
        Person person = new Person();
        person.setName("Kevin");
        person.setAge(30);
        HelloWorld helloWorld = new HelloWorld(person);
        assertTrue(
            "Hello! Kevin is 30".equals(
                helloWorld.sayIt()));
        assertTrue(person == helloWorld.getPerson());
    }
}
```

Listing 1: A TestCase doing too much

Listing 1 has a TestCase with all the fixture set up code within the test methods. This TestCase contains three different scenarios. Each scenario is set up and asserted in a different test method. You'll notice there is a fair bit of duplication between test methods. Also, the test methods are difficult to follow because there is so much fixture code to wade through (not really, in this trivial example, but use your imagination).

Listing 1 is also using assertTrue() for every assertion. This further decreases readability.

```
import junit.framework.TestCase;

public class HelloWorldTest_withNullPerson
        extends TestCase {

    private Person person;
    private HelloWorld helloWorld;
    protected void setUp()
            throws Exception {
        super.setUp();
        person = null;
        helloWorld = new HelloWorld(person);
    }
    public void test_sayIt() {
        assertEquals(
            "Hello!", helloWorld.sayIt());
    }
    public void test_person() {
        assertNull(helloWorld.getPerson());
    }

}

import junit.framework.TestCase;

public class HelloWorldTest_withNameOnly
        extends TestCase {

    private Person person;
    private HelloWorld helloWorld;
    protected void setUp()
            throws Exception {
        super.setUp();
        person = new Person();
        person.setName("Kevin");
        helloWorld = new HelloWorld(person);
    }
    public void test_sayIt() {
        assertEquals(
            "Hello!, Kevin", helloWorld.sayIt());
    }
    public void test_person() {
        assertSame(person, helloWorld.getPerson());
    }
}

import junit.framework.TestCase;

public class HelloWorldTest_withNameAndAge
        extends TestCase {

    private Person person;
    private HelloWorld helloWorld;
    protected void setUp()
            throws Exception {
        super.setUp();
        person = new Person();
        person.setName("Kevin");
        helloWorld = new HelloWorld(person);
    }
    public void test_sayIt() {
        assertEquals(
            "Hello!, Kevin", helloWorld.sayIt());
    }
    public void test_person() {
        assertSame(person, helloWorld.getPerson());
    }
}
```

Listing 2: Now three TestCases

Listing 2 contains cleaned up versions of the tests. Since each method represented a different scenario, I moved each scenario to its own TestCase and moved the fixture code to setUp().

To assert HelloWorld behaviors, I dumped all the assertTrue() methods and replaced them with more specific assertions, including assertNull() and assertEquals().

There is room for improvement in listing 2, though. Notice the duplication between the setUp() methods. (Again, this may not be obvious in this trivial example, but imagine that Person took 10 lines of code to set up.) This can be improved by extracting the set up of Person into a reusable Builder or ObjectMother. I will leave this as an exercise.

Correct

Of obvious importance, unit tests must be correct. This is not always easy to achieve. Before TDD, a developer had to devote his attention to ensuring that his functional code was correct. Now he has unit tests to give him positive feedback that his functional code is correct (or not correct!). No such luck with unit tests: Good, old-fashioned logic must be relied upon.

Don't rely only on your own über-programming skills. Whenever possible use pair programming. It provides an effective safety net when working on unit tests. Likewise, always remember to wear two hats when programming: a coding hat and a refactoring hat. Code when you have a red bar. Refactor when you have a green bar. Don't mix the two.

Sufficiently Complete

In addition to being correct, unit tests must also be sufficiently complete. Very few code bases have 100% test coverage and each team must determine their target coverage level. Use code coverage tools such as Emma, Coverlipse, and Jester. They can help you measure how much of your code base is covered by tests, find those gaps in coverage, and evaluate the semantic quality of your unit tests (how well the assertions are written).

Fast Enough

Finally, in addition to being correct and sufficiently complete, unit tests should run fast enough to be convenient. Fast tests encourage developers to run the entire test suite frequently throughout the day. Fast enough is subjective, but a good rule of thumb is that 100 unit tests should run in less than one second (much faster, if possible) In a current project of mine, our team has 2800 unit tests that run in 15 seconds.

That is almost all there is to turning your unit tests into "clean code that works." Two additional qualities are specific to unit test code, though: independence and isolation.

Independent

Firstly, unit tests must run *independently* of other tests. This ensures that one test's side effects will not affect the outcome of another test. This usually occurs when fixtures are not properly torn down between test runs. Bad fixtures could manifest themselves as external resources that retain some state or static class variables that are not reset between tests.

Isolated

Lastly, unit tests should exercise a specific cohesive unit of your functional code base. *Isolating* the unit you are testing has two advantages. Most importantly, isolating the code you are testing makes it easy to figure out why a test is failing. If you are debugging into multiple levels of an object graph or call stack trying to figure out why a test is failing, you need to further isolate the unit of code being tested from its collaborating objects. This is where stubs, fakes, or mocks come in handy. (Beware over-mocking, though, which can make tests brittle, i.e. cause internal refactorings to break tests.)

In addition to making a test failure easier to track down, isolated units take less fixture set up. This makes the tests easier to read and digest, as you don't have to understand as much set up logic to use the tests.

Conclusion

As teams move along the continuum from no test coverage to comprehensive test coverage, the value of their test suites increase. How valuable the tests ultimately become depends on two factors: How well the tests document the behavior of the system; and, how much flexibility the tests provide for the team when refactoring existing logic. To maximize the value of your team's test suites, treat the tests with the same care and consideration that you treat functional code.

Kevin P. Taylor is a Principal Consultant at Obtiva (http://www.obtiva.com), a firm that specializes in helping development teams transition to Agile methodologies. Kevin has written courses on test-driven development and Agile Project Management. Kevin is also the editor of http://java.about.com and the treasurer of Chicago Java Users Group-West.

CODE GENERATION 2007

MAY 18TH - 20TH, CAMBRIDGE UK

Industry experts agree that Code Generation is an essential tool in your development toolbox:

Dave Thomas: *"The leverage of code generators is incredibly important if you are to engineer accurate and maintainable systems."*
Andrew Watson: *"Our data shows that an MDA approach yields noticable savings in all but the very smallest projects."*

Code Generation 2007 is a new event for practising software developers. By taking part you will find out why these and other industry experts are so excited about the possibilities offered by emerging tools and technologies in this area. Come to Code Generation 2007 and improve your understanding of this important field and find out how to improve your day-to-day development work using these tools and technologies. Our aim is to draw the best practitioners from around the world to create a high quality learning experience for all participants.

http://www.codegeneration.net/cg2007/