

# goto; conference

Magazine vol.2, issue 1, 2012

www.gotocn.com



Polygot Persistence

Four Principles of Low-Risk Software Releases

Making Sense of Connected Data with Neo4J

Portfolio Kanban

Grace: an open-source, object-oriented programming language for education





# con

## goto magazine 2012

*Dear GOTO Community,*

*We can't stand the wait until GOTO Aarhus 2012, and we know that our speakers have a lot more to offer than the talks they give at the conferences, so why wait until October? Thus we decided to launch the second volume of the GOTO Conference Magazine to unveil insights from our GOTO Speaker community.*

*When we asked for input from speakers for this magazine, we were overwhelmed with the quality and the breadth of the material we received. But why? It just reflects what GOTO Conferences are all about. We strive to create breadth in our technical program at the conferences, not just one technology, never just one side of things.*

*Along with the breadth of technology, many of the case stories at the conference show how technologies and processes can be combined and used simultaneously. The logo of this year's GOTO Aarhus Conference visualizes this perfectly; like a machine that cannot work if one gear is missing, and this is in line with our vision of today's developers. If you choose to focus on only one aspect, you lose a lot of the strength you could have as a well-rounded developer.*

*We hope you enjoy reading the articles and viewing the interviews in this version of the GOTO Conference Magazine and we will release the next version in the fall*



# tents;

**01. Welcome to the goto magazine vol2, Issue 1, 2012**

**04. Polygot Persistence** / Martin Fowler, Pramod Sadalage

**10. Four Principles of Low-Risk Software Releases** / Jez Humble

**16. Backstage talks with GOTO speakers** / Interview

**18. Making Sense of Connected Data with Neo4J** / Jim Webber

**24. Portfolio Kanban** / Karl Scotland

**30. Grace: an open-source, object-oriented programming language for education**  
/ James Nobler, Kim.B.Bruce, Andrew P.Black, Michael Homer

## contributors





# Polyglot Pers



Martin Fowler,  
Author, speaker, consultant with  
ThoughtWorks and general loud-  
mouth on software development



Pramod Sadalage  
Principal Consultant, ThoughtWorks Inc

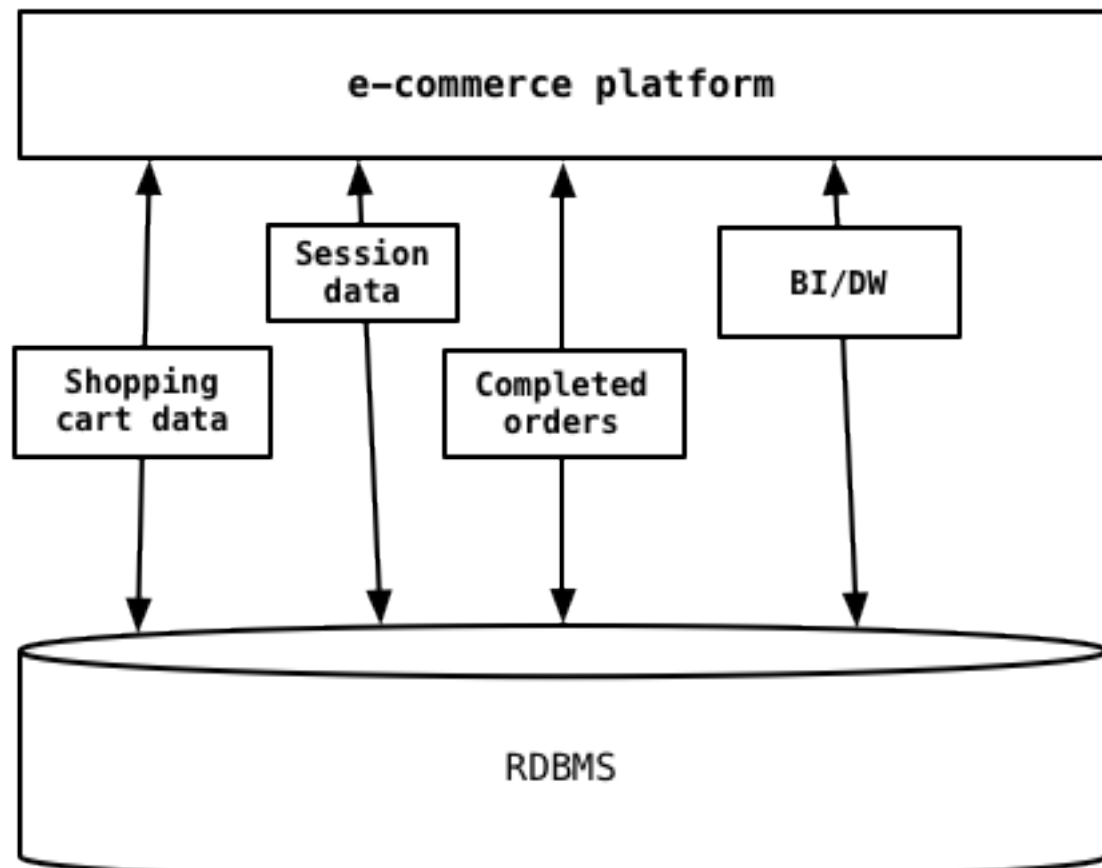
**Different databases are designed to solve different problems. Using a single database engine for all of your requirements, usually leads to non-performant solutions. Storing transactional data, caching session information, and traversing a graph of customers and the products their friends bought are essentially different problems. Even in the RDBMS space the requirements of an OLAP and OLTP system are very different, but they are often forced into the same schema.**

Consider the relationship between data. RDBMS are good at enforcing that relationships exist. If you want to discover relationships, however, or have to find data from different tables that belong to the same object, then using RDBMS is difficult.

Database engines are designed to perform certain operations on certain data structures and data amounts very well, such as operating on sets of data or storing and retrieving keys and their values really fast or storing rich documents or storing complex graphs of information.

## Disparate data storage needs

Many enterprises use the same database engine to store business transactions and session management data and also for other storage needs such as reporting, BI, data warehousing, and logging information.



*Use of RDBMS for every aspect of storage for the application*



# sistence

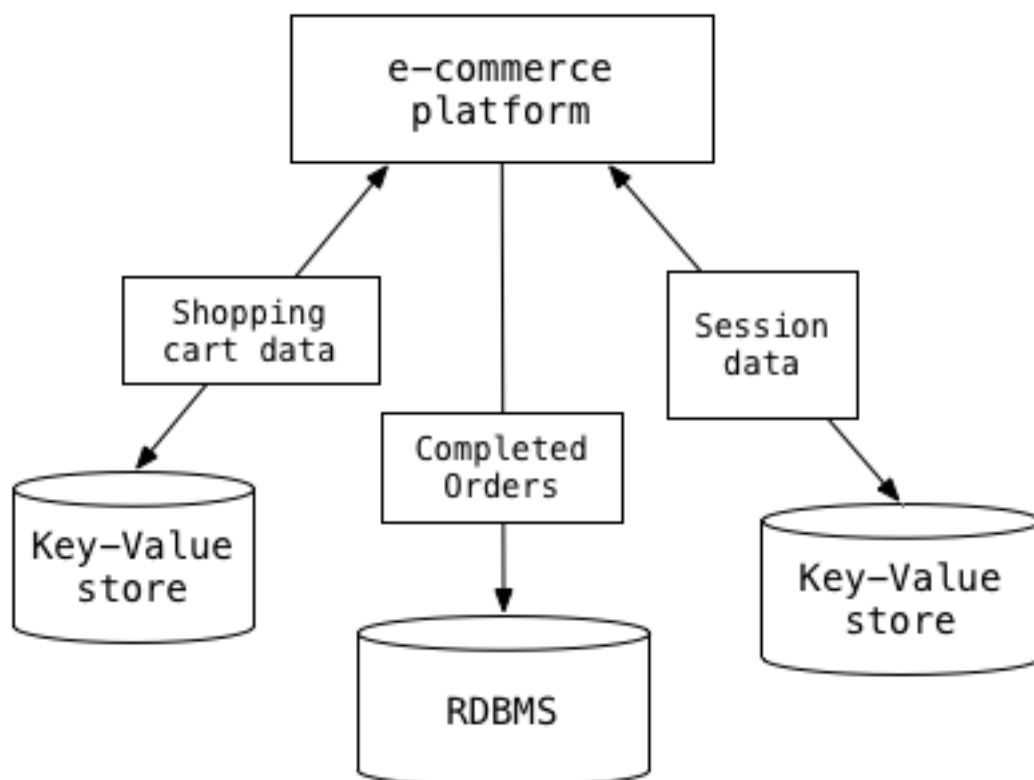
Session, shopping cart or order data, however, do not need the same availability or consistency properties or the same backup requirements of the data storage. Does session management storage need the same rigorous backup/recovery strategy as the e-commerce orders data? Does session management storage need more availability of an instance of the database engine to write/read session data?

In 2006 Neal Ford coined the term Polyglot Programming to express the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for tackling different problems. Complex applications combine different types of problems, so picking the right language for each aspect of the job may be more productive than trying to fit all aspects into a single language.

Similarly, when working on an e-commerce business problem, it is important to use a datastore that is highly available and scalable for storing the shopping cart. The same datastore, however, cannot help you find products bought by a customer's friends, which is a totally different question that the datastore needs to answer. We use the term Polyglot Persistence to define this hybrid approach to persistence.

## Polyglot datastore usage

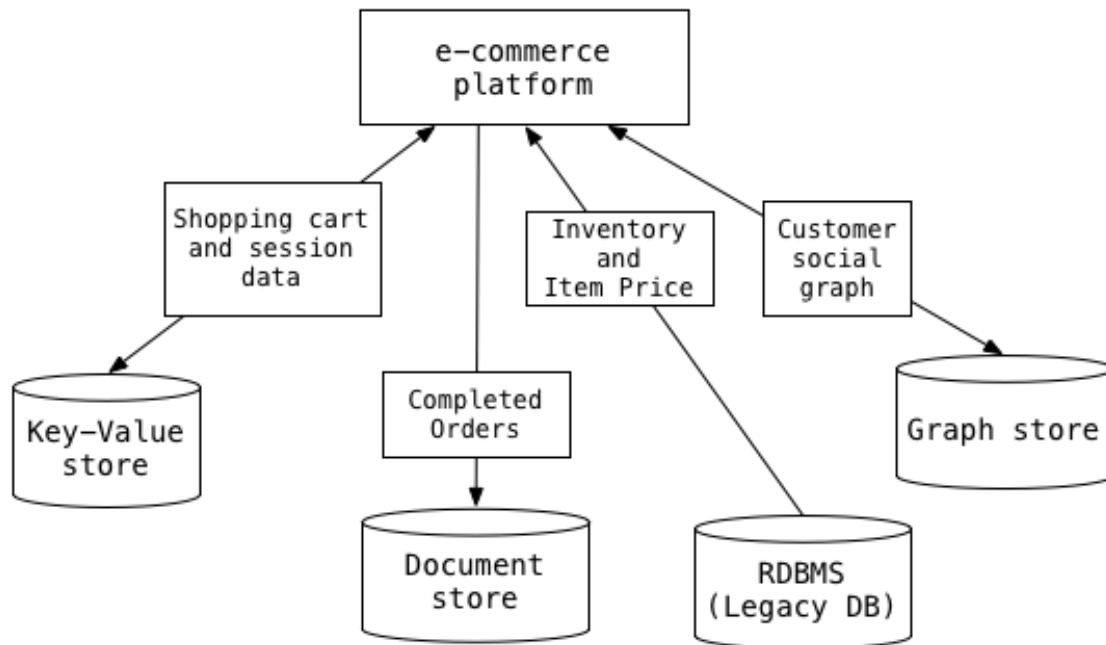
Let's review our e-commerce example using the polyglot persistence approach in our architecture and see how some of these different datastores can be applied. In this example, a key-value datastore is used to store the shopping cart data before the order is confirmed by the customer and also used to store the session data, but now the RDBMS is not used to store this transient data. Key-value stores make sense here because the shopping cart is usually accessed by userId and once it is confirmed and the order is paid for by the customer, it can be saved in the RDBMS. Similarly session data is keyed by the sessionId.



*Use of key-value stores to offload session and shopping cart data storage*

If you need to recommend products to customers when they put products into their shopping carts, with messages such as *Your friends also bought these products* or *Your friends bought these accessories for this product*, then introducing a graph datastore into the mix becomes relevant.





*Example implementation of Polyglot Persistence*

It is not necessary for the application to use a single datastore for all of its needs, since different databases are built for different purposes and not all problems can be elegantly solved by one single database.

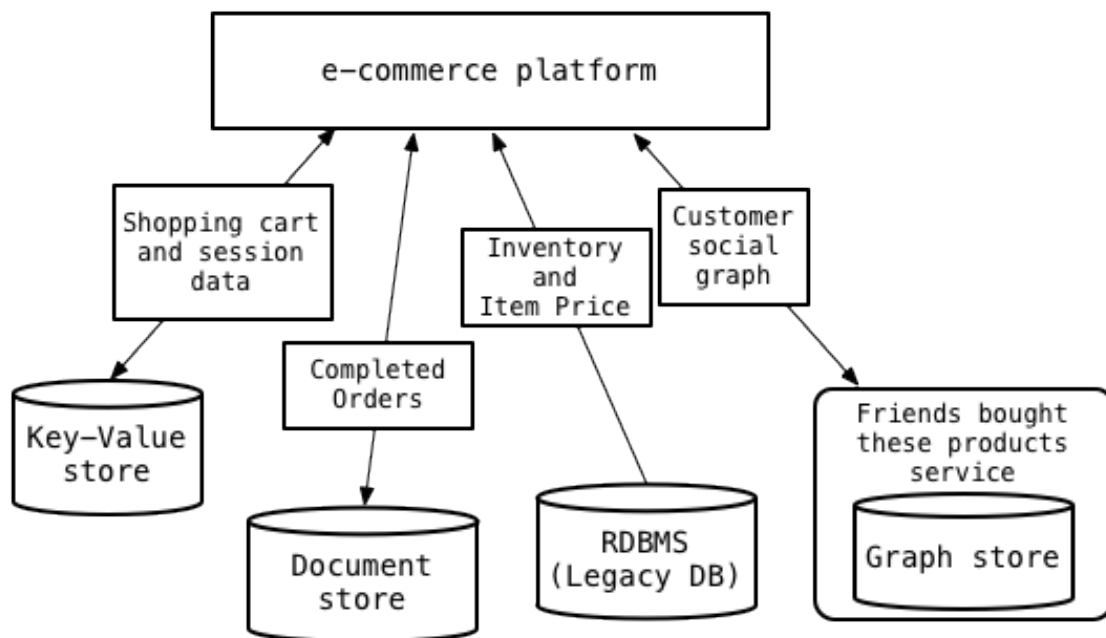
Even using specialized relational databases for different purposes within the same application, such as data warehousing appliances or analytics appliances, can be considered an example of polyglot persistence.

## Service usage over direct datastore usage

As we move towards using multiple datastores in an application, there may be other applications in the enterprise that could benefit from the use of those datastores or the data stored in the datastores. In our previous example, the graph datastore can serve data to other applications that need to understand a question like “which products are being bought by a certain segment of the customer base” or similar questions.

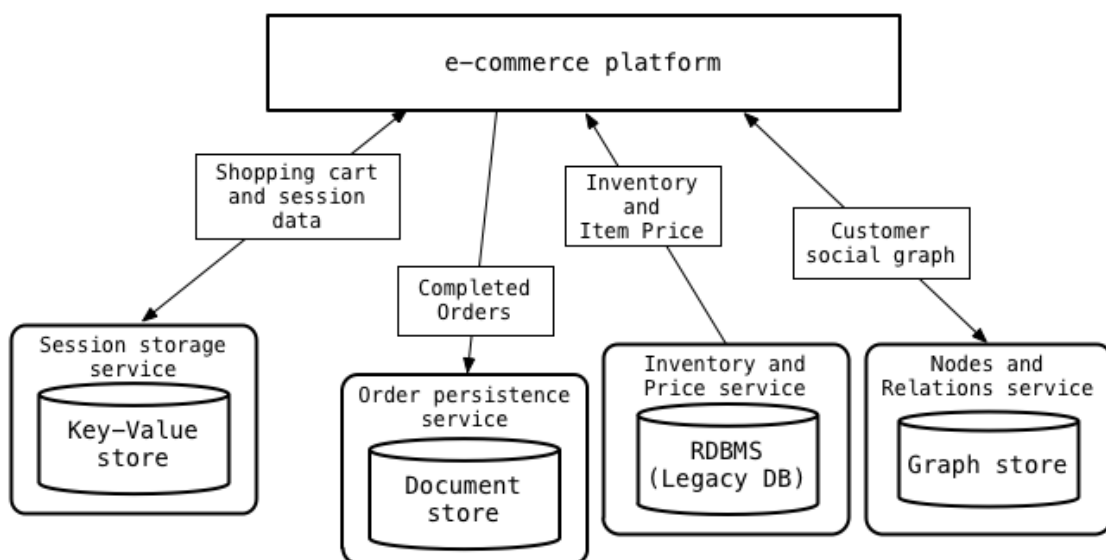
Instead of having each application talk independently to the graph database, wrapping the graph database with a service allows all relationships between the nodes to be saved in one place and queried by all the applications. The data ownership and the API’s provided by the service are more useful than a single application talking to multiple databases.





Example implementation wrapping data stores with services

This philosophy of service wrapping can be taken further. You could, for example, wrap all databases with services so that the application is basically just talking to a bunch of services. This allows for the databases inside the services to evolve without having to change the application.



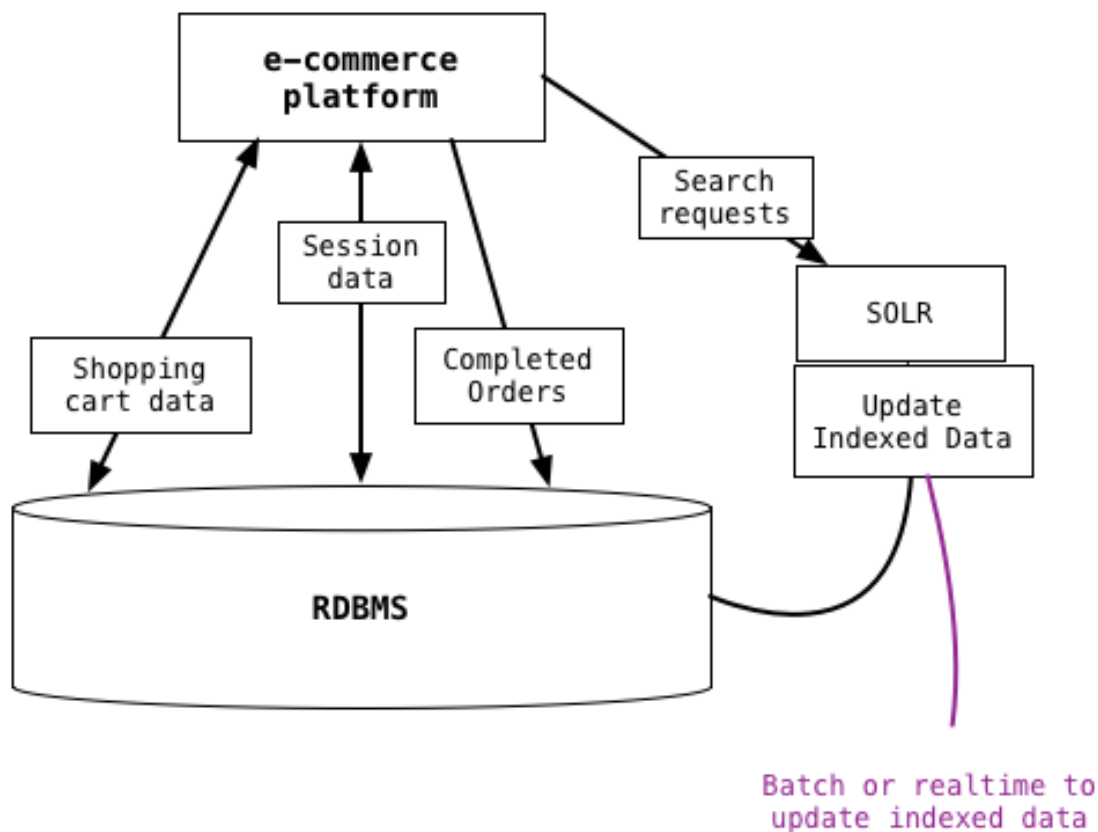
Using services instead of talking to databases

Many NoSQL datastore products actually provide out of the box REST APIs such as Riak and Neo4J

## Expanding for better functionality

Because of existing legacy applications and their dependency on existing data storage, many times we cannot easily move to new data storage systems to add functionality such as caching for better performance or to use indexing engines such as Solr so that search can be better served. When new technologies like Solr are introduced you have to make sure that data is synchronized between the data storage for the application and the indexing engine.

When there is a need to update the indexed data, as the data in the application database changes, the process to update the data can be real time or batch, as long as you ensure that the application can deal with stale data in the index/search engine. The event sourcing pattern described earlier can be used to update the index.



*Using supplemental storage to enhance legacy storage*

## Choosing the right technology

The various data storage technologies available today give you a rich choice of data storage solutions. The pendulum initially swung away from specialty databases to a single database like the RDBMS that allow all types of data models to be stored--although with some abstraction. The trend is now shifting back to using the data storage technology that supports the implementation of solutions natively.

Consider this, you may have to implement a feature that recommends additional products to customers based on what is in their shopping carts. Therefore you will need to know which additional products were purchased



by earlier customers who also bought the products that are in the current customer's shopping cart. This feature can be implemented in any of the data stores by persisting the data with the correct attributes to answer these questions. The trick is to use the right technology so that when the questions change, the answers can be asked of the datastore without losing existing data or changing all the existing data into new formats.

Going back to this new feature you may need, you could use a RDBMS to solve this problem using hierarchical query and model the tables accordingly. When you need to change the traversal, however, you will have to refactor the database, migrate the data, and then start persisting new data. Instead, if you had used datastores that track relations between nodes, we could have just programmed new relations and started using the relationships data store with minimal changes.

## Key Points

**Polyglot persistence is about using different data storage technologies to handle varying data storage needs.**

**Polyglot persistence can occur across an enterprise or within a single application.**

**Encapsulating data access through services reduces the impact of data storage choices on other parts of a system.**

*This excerpt is from Pramod Sadalage and Martin Fowler's upcoming book "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" that will be published in August.*

*Interested in learning more about NoSQL from Martin Fowler? At GOTO Aarhus 2012 Martin will present an "Introduction to NoSQL" in the "Apps of NoSQL" track.*

The GOTO team organizes year round GOTO Nights to support local communities and networks in the industry.

On May 31 we had the privilege of welcoming Pramod Sadalage from ThoughtWorks in Hamburg, Germany.

Pramod gave a talk about "NoSQL Distilled", we had a great night with an engaged audience.

Check out the presentation!



# Four Principles of Low-Risk Software

*Is your style of delivery high-risk, ‘big bang’ deployment? Unless you’re an adrenaline junkie, you’re just risking spectacular failure with your company’s money and your sanity. Jez Humble, coauthor of “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, provides detailed examples of how four simple principles can reduce your risk from high to low and increase your chances of success from low to high.*



Jez Humble  
Principal Consultant, ThoughtWorks



One key goal of continuous deployment is to reduce the risk of releasing software. Counter-intuitively, increased throughput and increased production stability are not a zero-sum game, and effective continuous deployment actually *reduces* the risk of any individual release. In the course of teaching continuous delivery and talking to people who implement it, I’ve come to see that “doing it right” can be reduced to four principles:

- Low-risk releases are incremental.
- Decouple deployment and release.
- Focus on reducing batch size.
- Optimize for resilience.

## NOTE

In this article, I’m focusing on release practices. Developers and testers can do many other things to reduce release risk, such as continuous integration and automating tests, but I won’t deal with those topics here. I’ve also limited the scope of the discussion to hosted services such as websites and “Software as a Service” systems, although the principles can certainly be applied more widely.

## Principle 1: Low-Risk Releases Are Incremental

Any organization of reasonable maturity will have production systems composed of several interlinked components or services, with dependencies between those components. For example, my application might depend on some static content, a database, and some services provided by other systems. Upgrading all of those components in one big-bang release is the highest-risk way to roll out new functionality.

Instead, deploy components independently, in a side-by-side configuration wherever possible, as shown in Figure 1. For example, if you need to roll out new static content, don’t overwrite the old content. Instead, deploy that content in a new directory so it’s accessible via a different URI—before you deploy

# are Releases

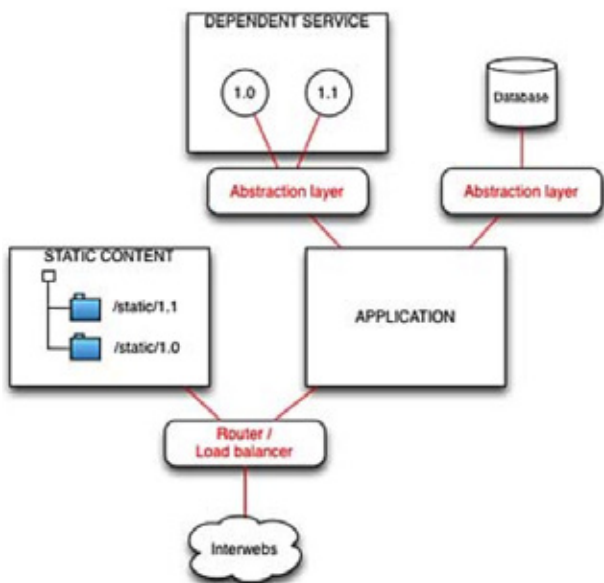


Figure 1 Upgrading incrementally.

Database changes can also be rolled out incrementally. Even organizations like Flickr, which deploy multiple times a day, don't roll out database changes that frequently. Instead, they use the expand/contract pattern. The rule is that you never change existing objects all at once. Instead, divide the changes into reversible steps:

1. Before the release goes out, add new objects to the database that will be required by that new release.
2. Release the new version of the app, which writes to the new objects, but reads from the old objects if necessary so as to migrate data "lazily." If you need to roll back at this point, you can do so without having to roll back the database changes.
3. Finally, once the new version of the app is stable and you're sure you won't need to roll back, apply the contract script to finish migrating any old data and remove any old objects.

Similarly, if the new version of your application requires a new version of some service, you should have the new version of that service up, running, and tested *before* you deploy the new version of your app that depends on it. One way to do this is to write the new version of your service so that it can handle clients that expect the old version. (How easy this is depends

a lot on your platform and design.) If this is impossible, you'll need to be able to run multiple versions of that service side by side. Either way, your service needs to be able to support older clients. For example, when accessing Amazon's EC2 API over HTTP, you must specify the API version number to use. When Amazon releases a new version of the API, the old versions carry on working.

Designing services to support clients that expect older versions comes with costs—most seriously in maintenance and compatibility testing. But it means that the consumers of your service can upgrade at their convenience, while you can get on with developing new functionality. And of course if the consumers need to roll back to an older version of their app that requires an older version of your service, they can do that.

Of course, you must consider lots of edge cases when using these techniques, and they require careful planning and some extra development work, but ultimately they're just applications of the branch-by-abstraction pattern.

Finally, how do we release the new version of the application incrementally? This is the purpose of the blue-green deployment pattern. Using this pattern, we deploy the new version of the application side by side with the old version. To cut over to the new version—and roll back to the old version—we change the load balancer or router setting (see Figure 2).



Figure 2 Blue-green deployment.

A variation on blue-green deployment, applicable when running a cluster of servers, is *canary releasing*. With this pattern, rather than upgrading a whole cluster to the latest version all at once, you do it incrementally. For example, as described in an excellent talk by Facebook's release manager, Chuck Rossi, Facebook pushes new builds to production in three phases (see Figure 3):

1. First the build goes to A1—a small set of production boxes to which only employees are routed.
2. If the A1 deployment looks good, the build goes to A2, a "cou-



ple of thousand” boxes to which only a small percentage of users are routed.

3. A1 and A2 are like canaries in a coal mine—if a problem is discovered at these stages, the build goes no further. Only when no problems occur is the build promoted to A

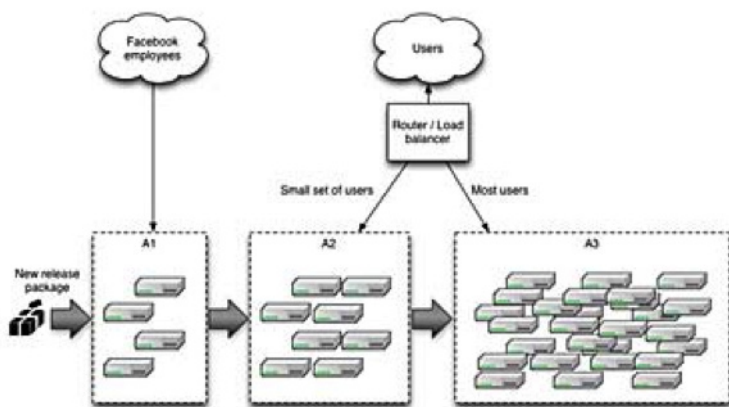


Figure 3 Facebook’s three phases for pushing new builds to production.

An interesting extension of this technique is the cluster immune system. Developed by the engineers at IMVU, this system monitors business metrics as a new version is being rolled out through a canary releasing system. It automatically rolls back the deployment if any parameters exceed tolerance limits, emailing everyone who checked in since the last deployment so that they can fix the problem.

### Principle 2: Decouple Deployment and Release

Blue-green deployments and canary releasing are examples of applying the second of my four principles: decoupling deployment and release. *Deployment* is what happens when you install some version of your software into a particular environment (the production environment is often implied). *Release* is when you make a system or some part of it (for example, a feature) available to users.

You can—and should—deploy your software to its production environment before you make it available to users, so that you can perform smoke testing and any other tasks such as waiting for caches to warm up. The job of smoke tests is to make sure that the deployment was successful, and in particular to test that the configuration settings (such as database connection strings) for the production environment are correct.

*Dark launching* is the practice of deploying the very first ver-

sion of a service into its production environment, well before release, so that you can soak test it and find any bugs before you make its functionality available to users. The term was coined by Facebook to describe its technique for proving out its chat service: “Facebook pages would make connections to the chat servers, query for presence information and simulate message sends without a single UI element drawn on the page.” When they were ready to release the chat service, they simply changed the HTML to point to the JavaScript which held the real UI. “Rolling back” would have involved simply changing back to the previous JavaScript.

However, as our systems evolve, it would be nice to have a way to decouple the deployment of a new version of our software from the release of the features within it. In this way, we can deploy new versions of our software continuously, completely independently of the decision as to which features are available to which users. Feature toggles can perform this function. As Chuck Rossi described, Facebook developed a tool called “Gatekeeper” that works with Facebook’s feature toggles to control who can see which features at runtime. For example, they can roll out a particular feature to only 10% of users, or only women under 25. This design allows them to test features on small groups of users and get feedback before a more general rollout. Similar techniques can be used for A/B testing.

Feature toggles also enable you to degrade your service under load gracefully—as Facebook did when launching usernames—and to switch off problematic new features if bugs are discovered in them, rather than rolling back the release by redeploying the previous version.

### Principle 3: Focus on Reducing Batch Size

Another essential component of decreasing the risk of releases is to reduce batch size. In general, reducing batch size is one of the most powerful techniques available for improving the flow of features from brains to users. Donald G. Reinertsen spends a whole chapter in his excellent book “The Principles of Product Development Flow: Second Generation Lean Product Development” (Celeritas, 2009) discussing a whole constellation of benefits generated by reducing batch size, from reducing cycle time (without changing capacity or demand) and preventing scope creep to increasing team motivation and reducing risk.

We particularly care about that last benefit—reducing risk. When we reduce batch size we can deploy more frequently, because reducing batch size drives down cycle time. Why does this reduce risk? When a release engineering team spends a week-end in a data center deploying the last three months’ work, the last thing anybody wants to do is deploy again any time soon.

But, as Dave Farley and I explain in our book “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, when something hurts, the solution is to do it more often and bring the pain forward. Figure 4 shows a slide from John Allspaw’s excellent presentation “Ops Meta-Metrics: The Currency You Use to Pay for Change,” which should help to illustrate the following discussion on how reducing batch size helps decrease deployment risk.

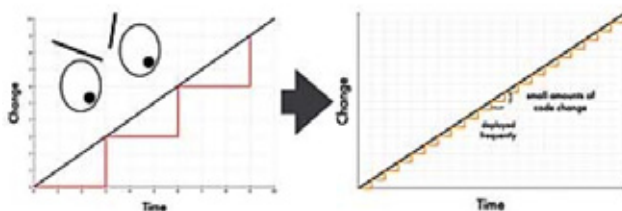


Figure 4 Reducing batch size reduces risk.

Deploying to production more often helps to reduce the risk of any individual release for three reasons:

- When you deploy to production more often, you’re practicing the deployment process more often. Therefore, you’ll find and fix problems earlier (and hopefully in deployments to preproduction environments), and the deployment process itself will change less between deployments.

The other reasons have to do with optimizing the process of fixing incidents. It’s often the case that a deployment gone wrong causes an incident. Incidents occur in three phases:

1. Finding out that an incident has in fact occurred (which is why monitoring is so important).
2. Finding out enough about the root causes to be able to work out how to get the system back up again.
3. Getting the system back up, followed by root-cause analysis and prioritizing work to prevent the incident from happening again.

Deploying more frequently helps with the second and third steps of the incident-resolution process.



Figure 5 Lifecycle of an incident.

- When you’re deploying more frequently, working out what went wrong is much easier because the amount of change is much smaller. It’s going to take you a very long time to find what went wrong if you have several months’ worth of changes to search—probably you’ll end up rolling back the release if you have a critical issue. But if you’re deploying multiple times a week, the changes between releases are small, and they’re likely to be a good place to start when looking for the root causes of the incident.

- Finally, rolling back a small change is much easier than rolling back several months’ worth of stuff. On the technical front, the number of components affected is much smaller; on the business front, it’s usually a much easier conversation to persuade the team to roll back one small feature than twenty big features the marketing team is relying on as part of a launch.

If your deployment pipeline is really efficient, it can actually be quicker to check in a patch (whether that’s a change to the code or a configuration setting) and roll forward to the new version. This is also safer than rolling back to a previous version, because you’re using the same deployment process you always use, rather than a rollback process that’s not as well tested.

As my colleague Ilias Bartolini points out, this capability depends on two conditions:

- Having a small lead time between check-in and release, since often multiple commits are required to fix a problem. (You might first want to add some logging to help with root-cause analysis.)

- Your organization must be set up to support a highly optimized deployment process. Developers must be able to get changes through to production without having to wait for out-of-band approvals or tickets to be raised.

## Principle 4: Optimize for Resilience

As Allspaw points out, there are two fundamental approaches to designing a system. You can optimize for *mean time between failures (MTBF)*, or for *mean time to restore service (MTRS)*. For example, a BMW is optimized for MTBF, whereas a Jeep is optimized for MTRS (see Figure 6). You pay more for a BMW up front, because failure is rare—but when it happens, fixing that car is going to cost you. Meanwhile, Jeeps notoriously break down all the time, but it’s possible to disassemble and reassemble one in under four minutes.



Figure 6 Two different approaches to system design.

Like a Jeep, it should be possible to provision a running production system from bare metal hardware—or via a virtualization API—to a baseline (“known good”) state in a predictable time. You should be able to do this in a fully automated way by using configuration information stored in version control and known good packages (in ITIL-world, these come from your definitive media library).

This ability to restore your system to a baseline state in a predictable time is vital not just when a deployment goes wrong, but also as part of your disaster-recovery strategy. When Netflix moved its infrastructure to Amazon Web Services, building resilience into the system was so important that the developers created a system called “Chaos Monkey,” which randomly killed parts of the infrastructure. Chaos Monkey was essential both to verify that the system worked effectively in degraded mode—a key attribute of resilient systems—and to test Netflix’ automated monitoring and provisioning systems.

The biggest enemy in creating resilient system is what the *Visible Ops Handbook* (Kevin Behr, Gene Kim, and George Spafford; Information Technology Process Institute, 2004) calls “works of art”: components of your system that have been hand-crafted over the years and which, if they failed, would be impossible to reproduce in a predictable time. When dealing with such components, you must find a way to create a copy of that component by using virtualization technology—both for testing purposes, and so you can create new instances of it in the event of a disaster.

But the most important element in creating resilient systems is human, as Richard I Cook’s short and excellent paper “How Complex Systems Fail” points out. This is one of the reasons that the DevOps movement focuses so much on culture. When a service goes down, it’s imperative both that everyone knows what procedures to follow to diagnose the problem and get the system up and running again, and also that all the roles and skills necessary to perform these tasks be available and able

to work together well. Training and effective collaboration are key here—issues discussed at more length in John Allspaw and Jesse Robbins’ book *Web Operations: Keeping the Data on Time* (O’Reilly, 2010).

## Conclusions

At many of my early gigs, any change you wanted to put out had to include a rollback procedure in case the change went wrong. Often rollback meant redeployment of the previous version, along with rolling back database changes—processes that (rather like restoring from backups) had not been tested nearly as well as the deployment process, if at all.

One of the concepts introduced by ITIL is remediation, defined as “recovery to a known state after a failed change or release.” The patterns and practices described here provide a way to remediate in a low-risk way—perhaps by changing a router setting or switching off a problematic feature—without resorting to rollback to a previous version of your system.

With these techniques, you can dramatically reduce the risk of releasing to users. However, they come with an added development cost and require some upfront planning, so you pay a certain amount in advance in order to achieve this lowered risk. Often these kinds of costs are hard to justify, partly because people have a tendency to undervalue a reward that some way exists in the future. (This is a behavioral bias known as temporal discounting.) This is one of the reasons why reducing batch size—and thus decreasing lead time—is important: You also get feedback much sooner on the benefits of changing your delivery process, which increases motivation.

These practices also depend on having good foundations in place—effective monitoring, comprehensive configuration management, a deployment pipeline, and an automated deployment process. If you’re lacking in any of these areas, you’ll need to address them as part of implementing a more reliable release process.

Operations teams often resist change, on the basis that any change carries risk. While this is true, it doesn’t follow that we should attempt to reduce the frequency of changes, since this in turn leads to high-risk “big bang” deployments. Instead, create more stable and reliable services by building resilience into systems and working to minimize and mitigate the risk of each individual change.

*Thanks to Max Lincoln, Mark Needham, Ilias Bartolini, Peter Gillard-Moss, and Joanne Malesky for feedback on an earlier version of this article. Thanks also to Martin Fowler and John Allspaw for permission to reproduce their diagrams.*

*The GOTO Conference Magazine wants to thank InformIT for granting the permission to use this article*

*Jez Humble will run the “Continuous Delivery” Training Course at GOTO Aarhus 2012. He takes the unique approach of moving from release back through testing to development practices, analyzing at each stage how to improve collaboration and increase feedback so as to make the delivery process as fast and efficient as possible. Furthermore Jez will present in both the “Agile Perspectives” and the “Continuous Delivery” track at the conference.*





**Vikings at GOTO Aarhus 19:00 Oct. 1.th**

**Join the welcome party. See you there...**

# Conversations

## with a few of our GOTO Aarhus

*In May 2012 at GOTO Copenhagen, we had a chance to do a series of interviews with some of the speakers. These interviews are conversations with speakers around their particular area of expertise where they provide their own insights, interest and lessons learned. You can also look forward to meeting these speakers in person at GOTO Aarhus in October 2012.*

**Kasper Lund** is one of the lead developers behind V8 at Google. After implementing JS with V8, the team decided to create a language which would be easier to optimize, and to work with for developers, now known as Dart. Essentially, Dart is about enabling more people to do web programming. Hear directly from one of the lead-Dart developers, Kasper, on why the world needs Dart.

*Kasper Lund is the host of the "JavaScript" track in which he will also present about "Translating Dart to efficient JavaScript"*



<http://www.youtube.com/watch?v=HOSXv45QEEM>



**Brian Leroux** who leads open source PhoneGap project team at Adobe shares with us the history behind PhoneGap along with his thoughts on what the future holds in this candid video interview.

*Brian Leroux will present in both the "Mobile cross-platform, testing and tools" and the "Modern Client App Architecture" track at GOTO Aarhus 2012*



[http://www.youtube.com/watch?v=4tjXcD9TH\\_I](http://www.youtube.com/watch?v=4tjXcD9TH_I)

# 2012 Speakers

**Simon Brown**, Founder of Coding the Architecture and either a software architect who codes or a software developer who understands architecture, talks about what it takes to be an architect, a bit about why he became interested in software architecture himself and what is needed in terms of modeling to support architecture design.

*Learn in practice what Simon Brown talks about in this interview. Sign up for his “Effective architecture sketches” training session, where you’ll be asked to design a software system and practice communicating your vision through a series of effective architecture sketches.*



<http://www.youtube.com/watch?v=6t0EOHy6OBA>



**Michael Nygard**, the man behind the book: “Release IT” shares his story of how he cheated his way into Ops, where he learned enough to write his book, and why it can be hard to introduce continuous delivery in your organization. This is also the first time Michael talks about NoOps on record, hear what he has to say on this in the video. *Michael Nygard hosts the “Continuous Delivery” track at in which he also a speaker. Furthermore does Michael will run a training session on “Production ready software”.*



<http://youtu.be/qkblZoRzEZo>

**Steve Freeman** and **Nat Pryce**, the authors behind “Growing Object-Oriented Software, guided by tests” talk in this video about how fear of adding more code makes developers create huge ugly classes. Also, they invite developers to be lazy so they stop spending time doing stupid things and instead write the tests that are needed to save work.

*Steve Freeman and Nat Pryce will present in the “Makers” track and teach the hand-on training session “TDD at the system scale”, about techniques for test-driven development at large scales, starting development with end-to-end tests at the system or system-of-systems level.*



<http://www.youtube.com/watch?v=KTraKQ9KOnY>





# Portfolio Kanban

## Steering the Agile Enterprise with Kanban Thinking



Karl Scotland: Software Systems Thinker, Practitioner and Agile Coach at Rally Software Development



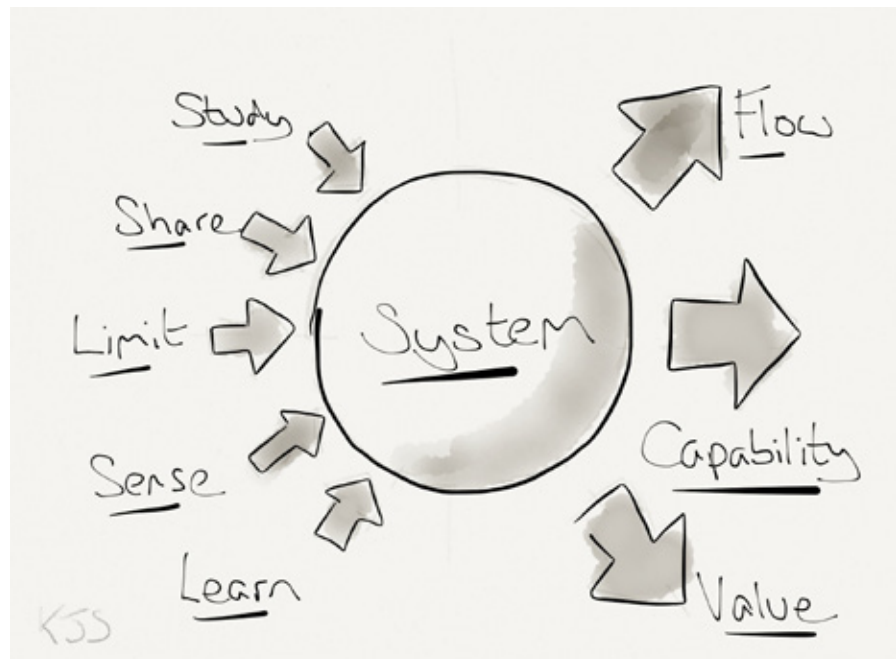
*Large scale, enterprise organisations consist of many initiatives, programs, projects and development teams. While it is well understood how each individual team can achieve the benefits of agility, it is more difficult for the enterprise as a whole to achieve those same benefits. This article will look at how Kanban Thinking can inform the design of a Portfolio Kanban System, with a view to steering the Agile Enterprise to attain those benefits in order to respond to today's competitive environments and unpredictable change.*

### KANBAN THINKING

Kanban Thinking is the framework I use when approaching the design of a Kanban System within a given context. The central concept is that the design approach is based on principles of Systems Thinking.

On the right side are the leverage activities for evolving the system design; study, share, limit, sense and learn. Studying the current system helps understanding of the existing context. Sharing that understanding gives everyone knowledge about what is happening. Putting limits in place bounds the system to help stabilise it. Sensing how the system is performing gives an understanding of current capability. Finally, learning from the systems performance allows its capability to be continually improved.

On the right side are the anticipated impacts the system design should have; flow, value and capability. Achieving flow involves a smooth, regular and frequent progress of work. Kanban Thinking looks to eliminate delays rather than eliminate waste. Delivering value involves focussing on doing the right thing in order to delight the customer (and other stakeholders). Kanban Thinking looks to maximise value rather than minimise cost. Building capability involves developing people and knowledge as a foundation for business success. Kanban Thinking looks to develop people as problem solvers rather than their tools to solve problems.



## THE PORTFOLIO AS A SYSTEM

As a system, a portfolio is more than the sum of its parts - that is the initiatives, projects, features, teams and people - but is a product of the interactions of those parts with a particular tendency. Managing a portfolio, and as a result steering the enterprise, is the job of designing and guiding the portfolio such that its tendency is to have a positive impact.

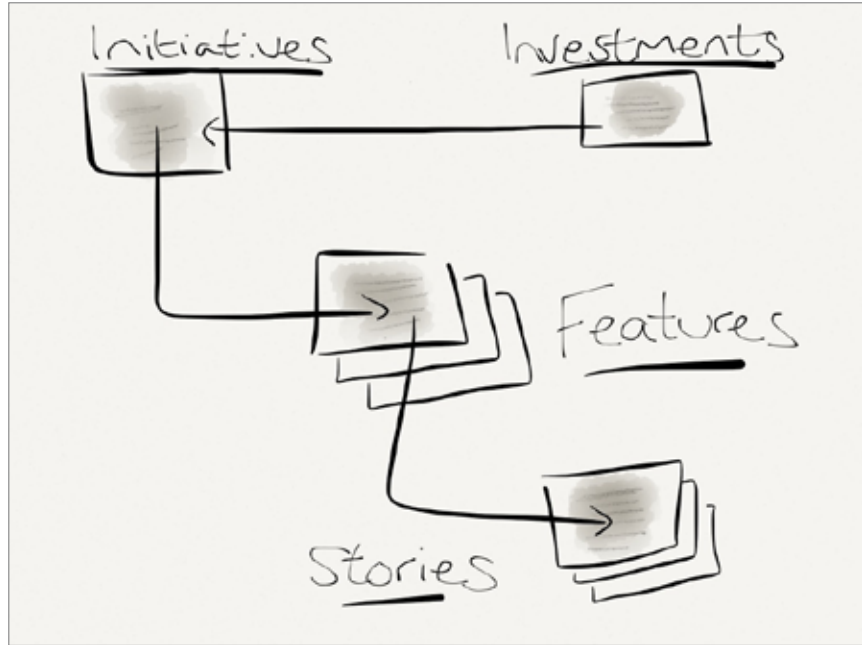
First I will describe how the levers can be used to discover a Portfolio Kanban System design, and then show how to understand the impact of that design.

### Portfolio Levers

#### Studying

When studying an existing portfolio, it is useful to begin by identifying all the work that is currently known about, from early ideation, to already in production and being maintained. That work can then be clustered and arranged into themes based on similarity or relatedness.

Common patterns which emerge are often based around investment and work item types, hierarchies and their governance workflow.



Examples of investment types might be areas such as architecture, urgent customer requests, current market segments, expanding market segments or future opportunities. Those investment types could be made up of initiatives, which are progressed through the development of features, which are iterated on by breaking down into stories.

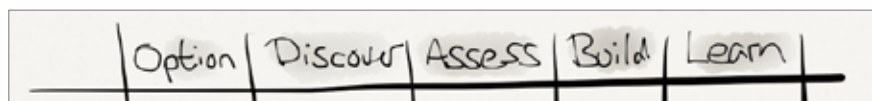
*An important question to ask when designing the mechanism to share the portfolio is what you want to understand in order to learn.*

*The TIP (Token, Inscription, Placement) heuristic is one I find useful to think about how to amplify the important signals, and dampen any noise.*

#### Sharing

Sharing a portfolio involves creating a model of the work which everyone can easily access and understand. Visual kanban boards provide a powerful mechanism for achieving this.

The most common approach is to create columns for the various stages of workflow that work items go through. For example, initiatives might begin as options, then have some discovery work done, then be assessed for suitability, then built and released before the results are reviewed for learning.



An important question to ask when designing the mechanism to share the portfolio is what you want to understand in order to learn. The TIP (Token, Inscription, Placement) heuristic is one I find useful to think about how to amplify the important signals, and dampen any noise.

#### Limiting

Limiting a portfolio is the means by which it can be effectively steered. By limiting the number of initiatives or features being worked on, they can be completed sooner, and with greater predictability, allowing organisations to get earlier feedback and respond better to new information and changing market conditions.

Work can be limited at a number of levels. Investment allocation can help keep the portfolio focussed on the right mix of work. The number of initiatives and features can be limited to focus on finishing work before new work is started. Flowing work through

stable teams can be used to balance demand against the capacity and capability of those teams.

As well as just limiting work in process through a Portfolio Kanban System, another form of limit is explicit policies. Creating transparency of the boundaries of the system design mean that the system can be stable within those constraints and the policies can be evolved to allow the system to evolve. A simple approach to policies is to add a checklist of exit criteria to each stage of the workflow.

	Option	Discover	Assess	Build	Learn
Exit Policy	== ==	== == ==	==	== ==	== == ==
Limit	5	3	2	3	4

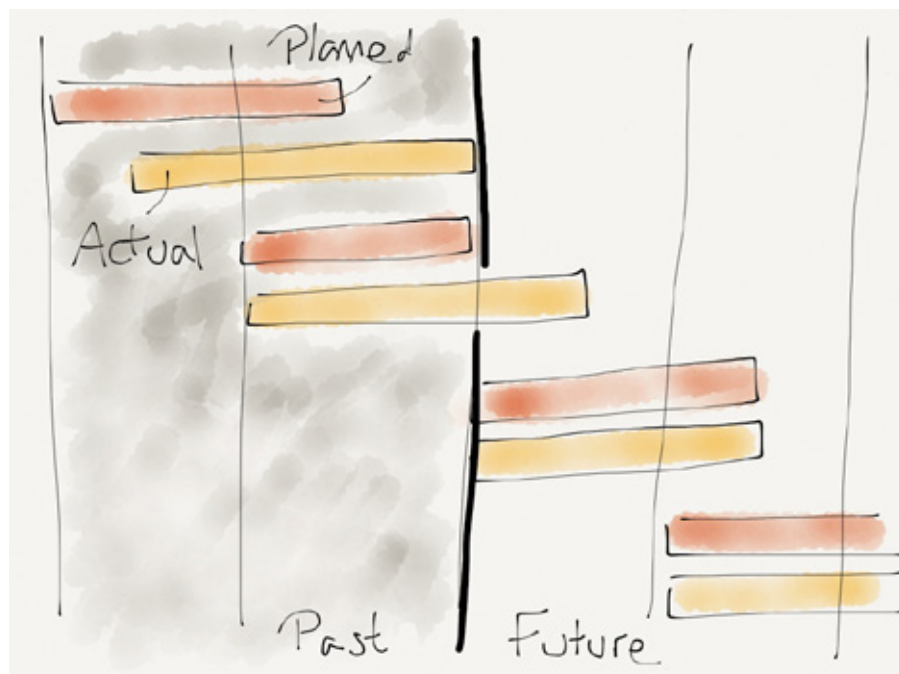
Sensing

Sensing the performance of the portfolio is what tells us how well it is being steered, so that direction can be adjusted effectively. There are generally two elements to sensing; establishing a cadence and measuring outcomes.

Establishing a cadence creates a sense of rhythm, and helps lessen the co-ordination cost of getting people together. Setting up a regular meeting to plan and review the portfolio enables a forum for gathering new information and feedback. A Portfolio Council is then able to respond by scheduling and re-prioritizing work in a timely manner and ensure focus is on the most important work.

Establishing appropriate measures generates insights which can aid decision making about what can be done to enable better outcomes. For a Portfolio Kanban System, financial measure seems to be appropriate. An economic model, which uses high level estimates of the costs and benefits of initiative or features, along with an understanding of the run-rate of teams, allows effective trade-off to be made between portfolio items. A timeline of planned and actual progress, for example, provides the basis for a rolling forecast as an alternative to annual plans.





### Learning

Whatever choices you make, the design of your Portfolio Kanban System, and the work within it, it will be wrong! Learning, the detection and correction of those errors, is key to evolving the portfolio to have a greater impact.

Steering a portfolio is about updating the portfolio to match the reality of the current situation, rather than managing the portfolio towards a future situation. By sensing current performance with a regular cadence, work can be advanced, delayed or even killed to keep focus on the most important work.

Evolving a Portfolio Kanban System is about running deliberate experiments, with prediction and validation of how changes to the system design will affect its impact. Again, by sensing current performance with a regular cadence, visualisations, WIP limits and other explicit policies can be adjusted as knowledge is gained about what a better design should be.

## Portfolio Impact

### Flow

Achieving flow across a Portfolio Kanban System means that the initiatives, programs, projects or features in the system progress with as few delays as possible. As a result the enterprise will be more responsive to the competitive landscape, so that when new information surfaces it can be reacted to effectively. Further, when flow is smooth across a portfolio, the work delivered by the system becomes more reliable as variability of lead times is reduced to within understood ranges.

### Value

Delivering value from a Portfolio Kanban System means the initiatives, programs, projects or features in the system are the most important things that could be worked on at that time. As a result stakeholder satisfaction will increase as customers get their needs met. Maintaining stakeholder satisfaction sustainably means that quality will also increase.

### Capability

Building capability with a Portfolio Kanban System means that the right initiatives, programs, projects or features can be effectively delivered over the long term, and not just the short term. As a result employee satisfaction will increase, leading to greater retention of people and their knowledge. This long term building of people, teams and their skills will also lead to an increase in overall productivity.

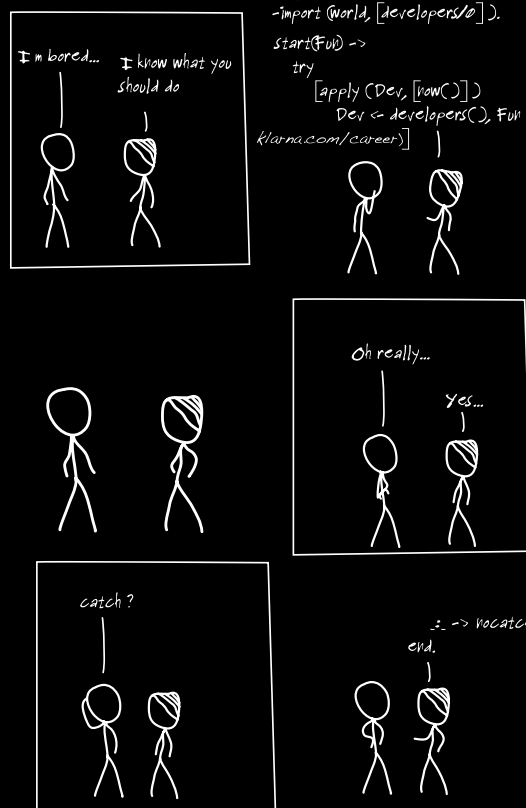


# mongoDB UK

..... 20 June London .....



MongoDB UK is an annual one-day conference dedicated to the open-source, non-relational database MongoDB. For more details, visit [mongoUK.com](http://mongoUK.com).




```

-module(job).
-export([start/1]).
-import(World, [developers/0]).
start(Fun) ->
  try
    [apply(Dev, [row()])
     Dev <- developers(), Fun(Dev,
                             klarna.com/career)]
  catch _ -> nocatch
end.
  
```

Oh really... yes...

catch? ... -> nocatch end.




## When it comes to managing high velocity data,

# ONE SIZE DOES **NOT** FIT ALL

## IMAGINE MILLIONS OF DATABASE OPERATIONS PER SECOND

- ✓ ACID transactions
- ✓ Realtime analytics
- ✓ “Five nines” availability
- ✓ Multi-level durability
- ✓ Cloud ready


[voltdb.com](http://voltdb.com)

[@voltdb](https://twitter.com/voltdb) 

Tame your data tsunamis with VoltDB




The NewSQL database for high velocity applications



[ree-ahk] **-noun**

1. The most powerful open-source, distributed database you'll ever put into production.
2. The feeling you get when disaster strikes and you realize you haven't lost any data.

(See also: ‘paradise,’ ‘utopia’)



# Making Sense of Connected Data with Neo4j



Jim Webber  
Chief Scientist at Neo Technology  
and Co-Author of “Rest in Practice”



## Making Sense of Connected Data with Neo4j

In order to provide better availability, scale and simplicity, the NOSQL movement has pushed data storage towards simpler models with more sophisticated computation infrastructure compared to traditional RDBMS. In fact, aggregate-oriented NOSQL stores<sup>1</sup> utilizing external map/reduce processing are commonplace today.

In contrast, graph databases like Neo4j actually provide a far richer data model than a traditional RDBMS and a search-centric rather than compute-intensive method for processing data. Instead of reifying links between data by processing that data in batches, Neo4j uses graphs as an expressive means of understanding, storing, and rapidly querying rich domain models.

Although the graph data model is the most expressive supported by the NOSQL stores, graphs can be difficult to understand amid the general clamor of the simpler-is-better NOSQL mantra. But what makes this doubly strange is that “simple” NOSQL databases have avowed their capabilities for graph processing too.

This strange duality where non-graphs stores can be used for (limited) graph applications was the subject many insightful mailing list posts from the graphistas in the Neo4j community<sup>2</sup>.

Community members have variously discussed the value of using non-graph stores for managing graph data, particularly since prominent online services have made popular graph processing (like Twitter's FlockDB).

As it happens the use-case for those graphs tends to be relatively shallow - "friend" and "follow" kinds of relationships. In those situations, it might be a passable solution to hold information in your values, document properties, columns, or even rows in a relational database to indicate a shallow relation as we can see in Figure 1:

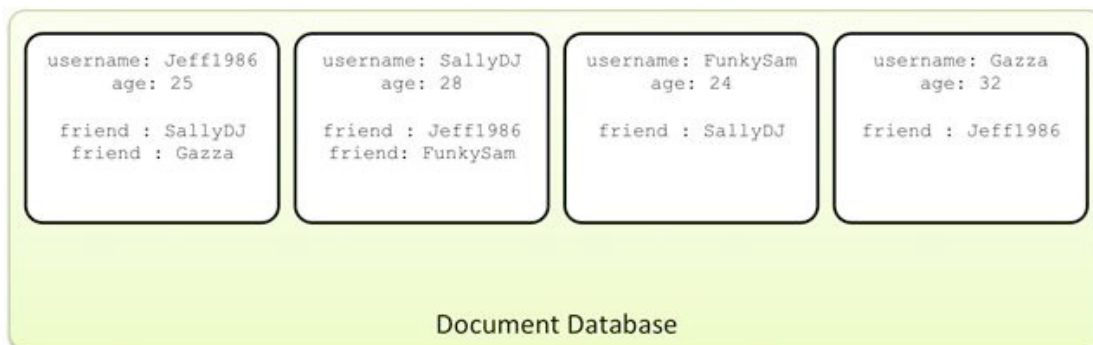


Figure 1 Using convention to imply links in flat data

At runtime, the application using the data store (remember: that's code we have to write and maintain) will try to process the implied foreign keys between stored documents to create a logical graph representation effectively reifying the business information out of the flat data source as we see in Figure 2. This means our application code needs to understand how to create a graph representation from those unlinked documents by enforcing some convention (and doing so consistently at scale).

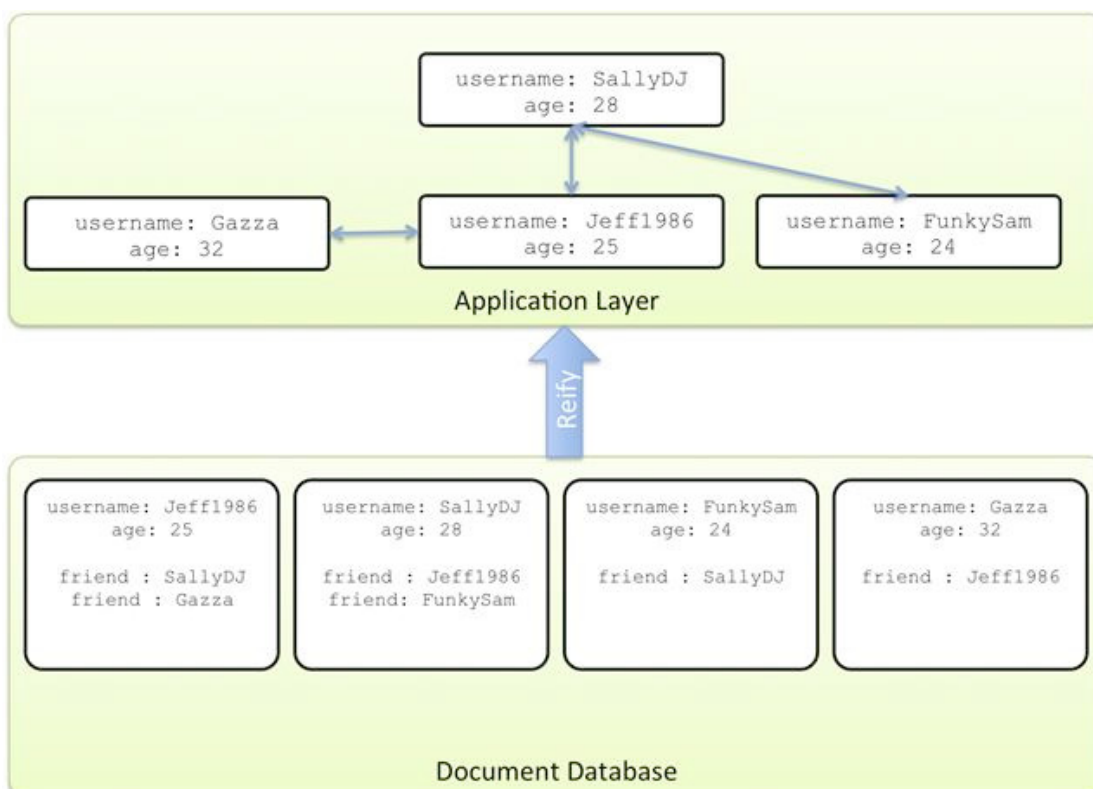


Figure 2 Reifying connected data out of flat data is hard work!



If the graphs are shallow, this approach might work. Twitter’s FlockDB is an existential proof of such. But as relationships between data become deeper, more intricate and valuable, this is an approach that runs out of steam.

Using the documents and foreign keys convention requires implicit graphs to be structured early on in the system lifecycle (at design time), which in turn means a particular topology from limited understanding of the data model of the system at that time is baked into the data store *and* into the application layer. This implies tight coupling between the code that reifies the graphs and the mechanism through which they’re flattened into and retrieved from the aggregate store. Any structural changes to the graph now require changes both to the stored data and the logic that reifies the data.

Neo4j takes a different approach: it stores graphs natively and so separates application and storage concerns. Simply put, where your application declares interrelated documents, that’s they way they’re stored, searched, and processed in Neo4j even if those relationships are very deep.

In our social example, the logical graph that we reified from the document store can be natively and efficiently persisted in Neo4j while remaining true to the domain that it supports as shown in Figure 3.

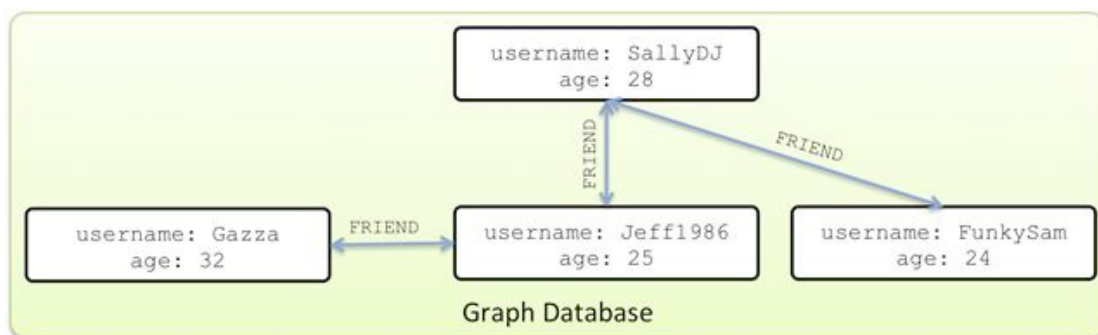


Figure 3 Storing connected data natively in Neo4j

It’s often deceptively easy in some use cases to project a graph from an aggregate store in the beginning. In fact it might seem that using an aggregate store – with its comfortable key/value semantics – is more productive.

For example, we might develop an e-commerce application with customers and items they have purchased. In an aggregate-oriented database we could store the identifiers of products our customers had bought inside the customer entity (or create separate purchase entities linked again by convention only). However making sense of that storage plan again requires application-level code that we have to build and maintain.

In Neo4j however we simply add relationships named PURCHASED between customer nodes and the product nodes they’d bought. Since Neo4j is schema-less, adding these relationships doesn’t require migrations, nor would it affect any existing code using the data. Figure 4 shows this contrast: the graph structure is explicit in the graph database, but implicit in a document store.

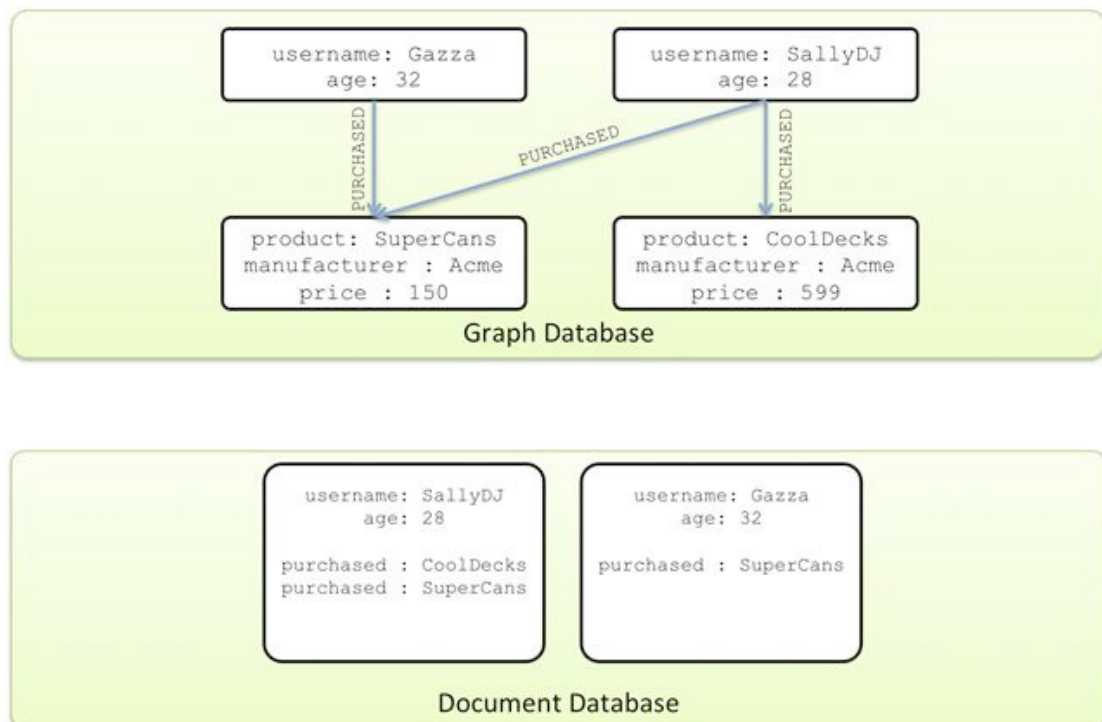


Figure 4 Comparing natively connected data with flat disconnected data

Even at this stage, the graph shows its flexibility. Imagine that a number of customers bought a product that had to be recalled. In the document case we'd run a query (typically using a map/reduce framework) that grabs the document for each customer and checks whether a customer has the identifier for the defective product in their purchase history. This is a big undertaking if each customer has to be checked (though thankfully because it's an embarrassingly parallel operation we can throw hardware at the problem). We could also design a clever indexing scheme, provided we can tolerate the write latency and space costs that indexing implies.

With Neo4j, all we need to do is locate the product (by graph traversal or index lookup) and look for incoming `PURCHASED` relations to determine immediately which customers need to be informed about the product recall. Easy!

Of course system requirements are rarely static and it's a simple step from a basic e-commerce application to evolve a social aspect to shopping so that customers can receive buying recommendations based on what their social group has purchased (and by extension people like them, people who live in their area, and people whose purchase history is similar) as we see in Figure 5.

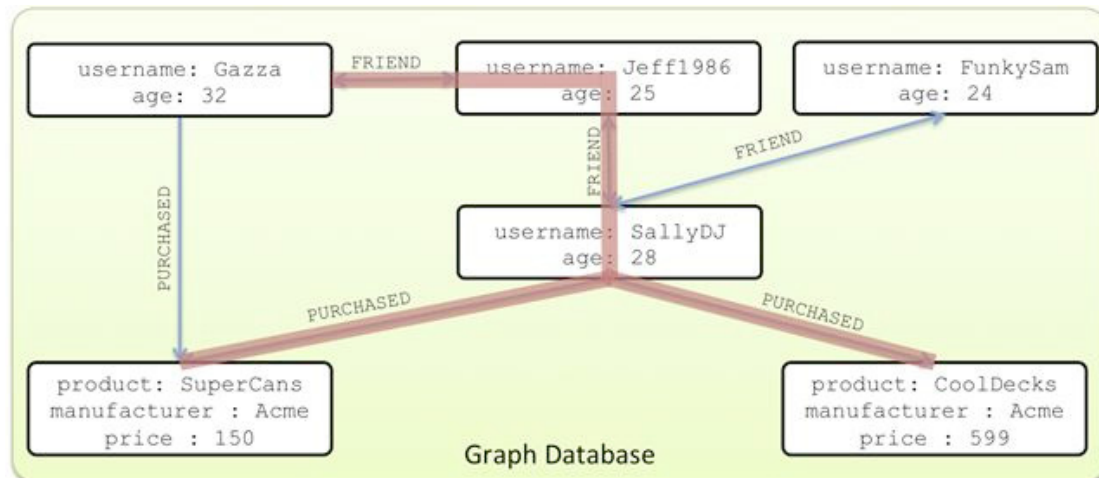


Figure 5 Social recommendations and selling are simple with graphs

In the aggregate store, we now have to encode the notion of friends and even friends of friends into the store and into the business logic that reifies the graph. This is where things start to get tricky since now we have a deeper traversal from a customer to customers (friends) to customers (friends of friends) and then into purchases. What initially seemed simple is now starting to look dauntingly like a fully-fledged graph store, albeit one we have to build and maintain.

Conversely with Neo4j we simply use the FRIEND relationships between customers, and for recommendations we simply traverse the graph across all outgoing FRIEND relationships (limited to depth 1 for immediate friends, or depth 2 for friends-of-friends), and for outgoing PURCHASED relationships to see what they've bought. What's important here is that it's Neo4j that handles the hard work of traversing the graph, not the application code, and Neo4j is capable of traversing millions of relationships per second on commodity hardware.

But there's much more value the e-commerce site can drive from this data. Not only can social recommendations be implemented by the activities of close friends, but the e-commerce site can also start to look for trends and base recommendations on them. This is precisely the kind of thing that supermarket loyalty schemes do with big iron and long-running SQL queries – yet Neo4j can achieve better results on commodity hardware (even down to the point of sale terminal itself).

For example, one set of customers that we might want to incentivize are those people who we think are young performers. These are customers that perhaps have told us something about their age, and we've noticed a particular buying pattern surrounding them - they buy DJ-quality headphones. Often those same customers buy DJ-quality decks too, but there's a potentially profitable set of those customers that - shockingly - don't yet own decks (much to the gratitude of their roommates and neighbors I suspect).

With an aggregate-oriented database, looking for this pattern by trawling through all customer documents and projecting a graph is laborious, even with map/reduce frameworks to automate the plumbing code and parallelize the work. But matching these patterns in a graph is quite straightforward and efficient – simply by specifying a prototype to match against and then by efficiently traversing the graph structure looking for matches<sup>3</sup> as we see in Figure 6.

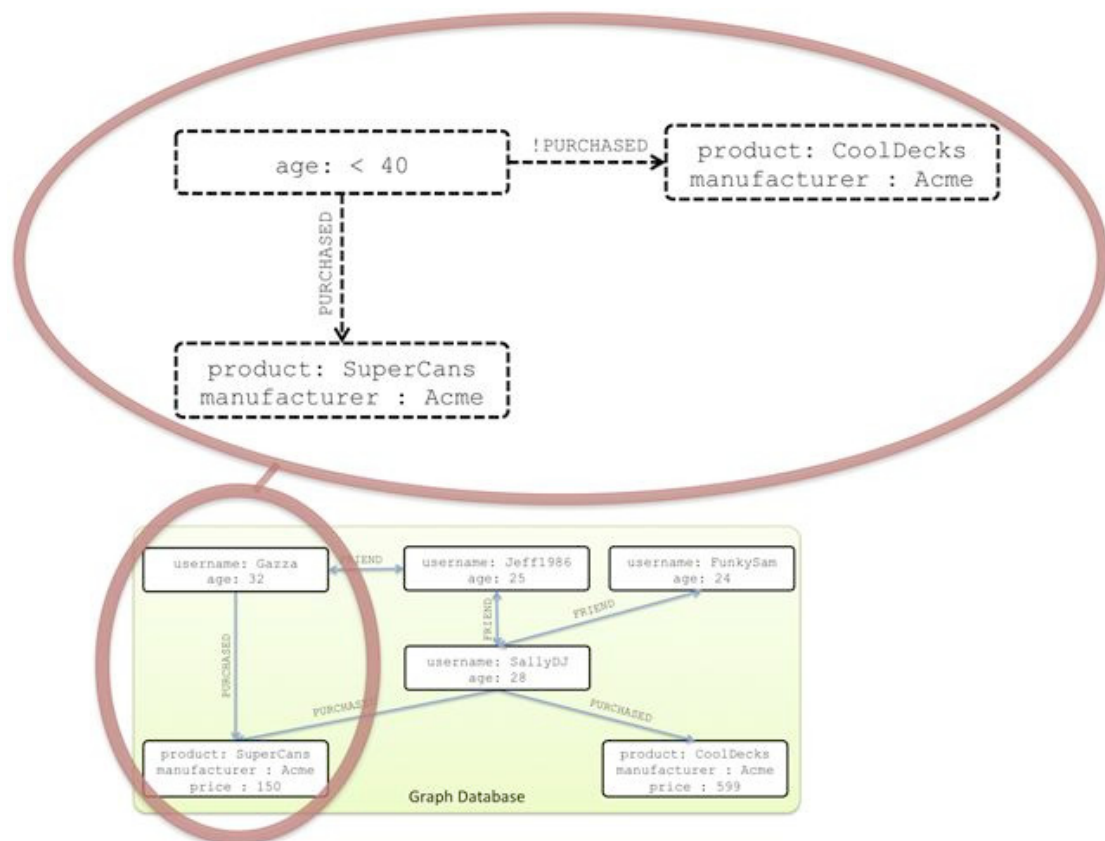


Figure 6 Sophisticated analytics patterns are made simple with graphs

This shows a wonderful emergent property of graphs - simply store all the data you like as nodes and relationships in Neo4j and later you'll be able to extract useful business information that perhaps you can't imagine today, without the performance penalties associated with joins on large datasets or latency associated with batch processing on external map/reduce frameworks.

In these kinds of situations, choosing a non-graph store for storing graphs is a gamble. You may find that you've designed your graph topology far too early in the system lifecycle and lose the ability to evolve the structure and perform business intelligence on your data. While in some edge cases non-functional concerns to drive you towards such a solution (e.g. taking advantage of the write throughput of Apache Cassandra or operational scale of Riak), in general you're better off with a graph database.

That's why Neo4j is cool - it keeps graph and application concerns separate, and allows you to defer data modeling decisions to more responsible points throughout the lifetime of your application. So if you're looking to drive value from your graph data then try Neo4j.

At GOTO Aarhus 2012 Jim Webber will run a training course, "A Programmatic Introduction to Neo4j ". It's a full day training course that covers the core functionality from the Neo4j graph database, providing a mixture of theory and accompanying practical sessions to demonstrate the capabilities of graph data and the Neo4j database.





# Grace: an open-source, object-oriented programming language for education



James Noble, Victoria University of Wellington  
kjj@ecs.vuw.ac.nz



Kim B. Bruce, Pomona College, CA  
kim@cs.pomona.edu



Andrew P. Black, Portland State University  
black@cs.pdx.edu



Michael Homer, Victoria University of Wellington  
mwh@ecs.vuw.ac.nz



Although object-oriented programming is widely taught in introductory computer science courses, no existing object-oriented programming language is the obvious choice for a teaching language. While Java was the *de facto* standard throughout the 2000's, in the last few years a range of newer languages such as Python, Ruby, Scala, C#, F#, Processing, and JavaScript have begun to make their way into classrooms — and so to research labs, offices, and, eventually, large software systems.

During ECOOP 2010, a group of language researchers and educators concluded that the time was ripe for an effort to design a language focussed on teaching. A “design manifesto” was presented at SPLASH 2010, in which we attempted to lay out design principles for a suitable a language. Since then three of us (Black, Bruce and Noble) have been meeting weekly to pursue the design of the language, which we have named

“Grace”, in honor of Admiral Grace Hopper, and in the hope that the name would serve as an admonition not to settle for less-than-graceful solutions.

## GRACE IN A NUTSHELL

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features. Our design choices have been guided by the desire to make Grace look as familiar as possible to programmers who know contemporary object-oriented languages such as Java, C#, Ruby, Scala, and Python. We have also been motivated by the need to give instructors and text-book authors the freedom to choose their own teaching sequence. Thus, in Grace it is possible to start teaching using types, to introduce types later, or not to use types at all. It is also possible to begin with objects, or with classes, or with functions. Importantly,

# oriented for

instructors can move from one approach to another while staying within the same language.

The traditional first program in a new language is “Hello, World”. In Grace, we kept this program as simple as we possibly could:

```
print "Hello, World"
```

We think that “Hello, World” needs to be especially simple, for a number of reasons. Not only is “Hello World” the first program many experienced programmers write in a new language, it is also the first programmer beginners will write in any language. We want those first programs to be easy because programming is hard enough as it is: especially for novices, we don’t want the programming language to get in the way of teaching the fundamental ideas. The less syntax that is required, the less syntax there is for novices to get wrong. Our experiences teaching Java, where it is necessary to have the whole class chant incantations like “`public static void main(String arg[])`” before the students could even print “Hello” — and then having to explain the resulting error messages — have convinced us that avoiding such accidental complexity is important.

## OBJECTS AND CLASSES

Grace can be regarded as either a class-based or an object-based language, with single inheritance. A Grace class is an object with a single factory method that returns a new object:

```
class aCat.named(n : String) {  
    def name = n  
    method meow { print "Meow" }  
}
```

Here the class is called `aCat` and the factory method is called `named`. We can create an instance of that class — a new cat object — and store it in a variable:

```
var theFirstCat := aCat.named "Timothy"
```

After executing this code sequence, `theFirstCat` is bound to an object with two attributes: a constant field (`name`), and a method `meow`. The method request `theFirstCat.name` answers the string object “Timothy” and `theFirstCat.meow` has the effect of printing *Meow*.

An object can also be constructed using an object literal — a particular form of Grace expression that creates a new object when it is executed. For example:

```
var theSecondCat := object {  
    def name = "Timothy"  
    method meow { print "Meow" }  
}
```

This code binds the variable `theSecondCat` to a newly created object, which happens to be operationally equivalent to `theFirstCat`.

## FIELDS AND VARIABLES

Mutable and immutable bindings are distinguished by keyword: `var` defines a name with a variable binding, which can be changed using the `:=` operator, whereas `def` defines a constant binding, initialized using `=`, as shown here.

```
var currentWord := "hello"  
def world = "world"  
  
currentWord := "new"
```

The keywords `var` and `def` are used to declare both local bindings and fields inside objects. Like Java — but unlike JavaScript — fields and methods cannot be added to an object after it is created. A field that is declared with `def` is constant. Each constant field declaration creates an accessor method

on the object. Declaring a field with `var` creates two accessor methods, one for fetching the currently bound object and one for changing it. So, after the declaration

```
def car = object {  
  def numberOfSeats = 4  
  var speed: Number := 0.  
}
```

the object `car` will have three methods called `numberOfSeats`, `speed`, and `speed:=()`. When we use `()` in the name of a method, it indicates the need to supply arguments. So, the last method might be used by writing `car.speed := 30`.

## REQUESTING METHODS

Grace method names may consist of multiple parts (“mix-fix notation”) as in Smalltalk or Objective-C. Separate lists of arguments are interleaved between the parts of the name, allowing them to be clearly labeled with their purpose. Thus we might define on `Number` objects

```
method between (l:Number) and (u:Number)  
{  
  return (l < self) && (self < u)  
}
```

The syntax of a method request is similar to that used in Java, C++, and many other object-oriented languages: `obj.meth(arg1, arg2)`, but extended to allow the name of the method to have multiple parts. We could request the above method `between()and()` on 7 by writing

```
7.between(5)and(9)
```

Single arguments that are literals do not require parentheses, so alternatively we could write

```
7.between 5 and 9
```

Following many other languages, the receiver `self` can be omitted. We have already seen several messages requested of an omitted receiver; for example, `print “Meow”` is short for `self.print “Meow”`.

## BLOCKS AND CONTROL STRUCTURES

Like Ruby, C#, and Scala, Grace includes first-class blocks (lambda expressions). A block is written between braces and contains some piece of code for deferred execution. A block

may have arguments, which are separated from the code by `->`, so the successor function is `{x -> 1+x}`. A block can refer to names bound in its surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

Control structures are designed to look familiar to users of other languages. However, as in Smalltalk and Self, control structures in Grace are just methods that take blocks as arguments.

```
if (x > 5) then {  
  print “Greater than five”  
} else {  
  print “Too small”  
}  
  
for (node.children) do {  
  child -> process(child)  
}
```

Notice that the use of braces and parentheses is not arbitrary: parenthesized expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may thus be evaluated zero, one, or many times. A `return` statement inside a block terminates the *method* that lexically encloses the block, so it is possible to program *quick exits* from a method by returning from the *then* block of an `if()then()` or the *do* block of a `while()do()`.

## TYPES

Types and classes are strictly separated in Grace. A Grace class is not a type, nor does a Grace class or object implicitly define a type. When programmers need types they must define them explicitly. We hope this separation will help us teach the concepts of types independently from classes. To this end, Grace supports both statically and dynamically typed code: omitted types of local variables and constants are inferred (as e.g. in Scala or C#), but omitted argument types are treated as the predefined type `Dynamic`. Messages requested on `type Dynamic` will be checked dynamically.

Grace types are structural: they describe properties of objects. A type is a set of method requests; a type declaration gives a name to a type.

```
type Vehicle = {  
  numberOfSeats -> Number  
  speed -> Number  
  speed:=(n : Number) -> Nothing  
}
```

An object has a type if it has the appropriate methods, and if

the signatures of those methods conform to the signatures in the type. No inheritance relationships or implements declarations are necessary. The `car` object defined above has the `Vehicle` type, but also has the smaller type `{ speed -> Number}`.

Within dynamically typed code, types need not be mentioned at all, and so the introduction of the concept of type can be delayed until late in the teaching sequence. When instructors do introduce types, they may do so in the language they are already using, as opposed to, for example, starting teaching in Python and then transitioning to Java. A static type checker will support instructors who wish to require that all student programs be fully typed.

## HOW CAN YOU CONTRIBUTE?

The Grace Project maintains a website at <http://gracelang.org>, including the project diary (as a blog), the evolving language specification, an early prototype

implementation, and other documents and papers about Grace. We are actively interested in comments and feedback about the language design, and as the project goes on, about APIs, libraries, and implementations. We would really appreciate programmers trying out prototypes as they are released, and ultimately testing the specification by building alternative implementations.

## References

AP Black, KB Bruce, M Homer, J Noble. *Grace: the absence of (inessential) difficulty*. Accessible from [gracelang.org/documents](http://gracelang.org/documents), April 2012.

AP Black, KB Bruce, J Noble. *The Grace Programming Language Draft Specification Version 0.353*. Accessible from [gracelang.org/documents](http://gracelang.org/documents). April 2012.

## Award-Winning Vendor of Developer Productivity Tools



[jetbrains.com](http://jetbrains.com)

# AARHUS

INTERNATIONAL  
SOFTWARE DEVELOPMENT  
CONFERENCE 2012

Conference: Oct 1 - 3  
Training: Sept. 30, Oct 4 - 5

**goto;**  
conference



# goto; conference

INTERNATIONAL  
SOFTWARE DEVELOPMENT  
CONFERENCE

- made for developers by developers...proudly presenting the best speakers & brightest attendees

www.gotocon.com



## AARHUS

Training : Sept 30, Oct 4-5 // Conference : Oct 1-3, 2012



## ZÜRICH

Conference : April 10-11, 2013



## CHICAGO

Conference : April 23-24 // Training : April 25-26, 2013



## AMSTERDAM

Conference : June 18-19 // Training : June 20, 2013



## STAY IN TOUCH...

Follow us on  
twitter

Sign up for newsletter  
Next issue in August '12