What do I think about Conway's Law now?

Conclusions of a EuroPLoP 2005 Focus Group

Authors and focus group leaders:

- Lise Hvatum - hvatum1@sugar-land.oilfield.slb.com
- Allan Kelly - allan@allankelly.net

EuroPLoP is the annual European Patterns conference - more information here, http://hillside.net/europlop/

As well as reviewing Patterns the conference hosts a number of "Focus Groups" where participants discuss topics concerned with writing software, software development in general and related topics.

In July 2005 Lise Hvatum and myself hosted over four hours of discussion over two days on the subject of "Conway's Law." This paper describes the groups conclusions and our own thinking on the subject at the end of the session and after some reflection.

Conway's Law is still open to debate.

We have proposed a second focus group for EuroPLoP 2006 on "Socio-Economic forces effecting software development."

Some names have been replaced by pseudonyms.

Allan Kelly, January 2006.

Background to this paper

This work captures the (possibly confused) thinking from the two focus group leaders at EuroPLoP 2005 after the focus group was over and we had summarized the outcome. We both went into the focus group thinking of Conway's Law as inevitable. Unless you heeded the organizational structure when developing software you would be in trouble.

Thanks to a fascinating and vital group of people attending the focus group, we were quickly and resolutely kicked out of our sandbox and for a period felt we were flying in outer space. Not in control (we could not really stick to the prepared model for how the focus group was to proceed, but then who cares about pre-made plans anyway…) but completely fascinated with the new discoveries the group was doing.

Defining Conway's Law

When discussing Conway's Law we need to be clear on which version we are analysing. In addition to the original definition from the article of Melvin Conway (Conway, 1968) there are a number of paraphrased versions of Conway's Law in use. In this document we will discuss the following three versions:

- Raymond's version:

  > "Conway's Law *prov*. The rule that organization of the software and the organization of the software team will be congruent; originally stated as 'If you have four groups working on a compiler, you'll get a 4-pass compiler'." (Raymond, 1996)

  This version is the easiest to consider. It deals with a micro-situation. Unless a conscious effort is made to avoid this situation it will happen. The default position of many developers seems to be to divide the problem into a number of chunks equal to the number of developers. Although this may seem easy to manage these chunks are unlikely to be even sizes and the resulting design is unlikely to be particularly good. Sometimes this design is even a "make work" exercise because the problem could be solved by a better design using fewer modules and fewer programmers. This version of the law may be the best known and the easiest to understand but it is actually the least interesting.

- Coplien and Harrison's version:

  > "If the parts of an organization (e.g. teams, departments, or subdivisions) do not closely reflect the essential parts of the product, or if the relationship between organizations do not reflect the relationships between product parts, then the project will be in trouble.
  > ...
  >
  > Therefore: Make sure the organizations is compatible with the product architecture." (Coplien and Harrison, 2004)

  The pattern applies to the macro environment, and in short tells you to "align architecture with organization". System architecture will be influenced by social and economic factors, of which the true (not formal) communication structure in the organization is the strongest. To apply the pattern, you are faced with a number of questions on how to implement it; what exactly to build, where to start, and not the least how to change an organization to support the product architecture.

- Conway's original form of the law:

  > "organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations." (Conway, 1968)

  This version is the most interesting version and is compatible with Parnas' "module as a responsibility assignment" (Parnas, 2001) - if you are responsible for a module you should communicate with those who you affect and should be talked to by those who use your code. If you have a high communication bandwidth (e.g. sit next to each other) your interface will be informal, if you are physically separate and communicate little you may publish a formal API.

In writing up these notes the authors also became aware of the work of Herbsleb and Grinter (1999). This qualitative study considers a case in which developers have problems with modules with which they need to interface and for which they have no contact with the developer. Their case study exposes the fallacy of written documentation for this purpose.

Therefore, it is a foolish organization that attempts to limit communication between those who need to interface. However, it may not be obvious to an organization that two individuals or groups need to communicate so they may break this link without realising it.

A new case study

One of the session participants, Keith Braithwaite, introduced the group to a case study in which he claimed Conway's Law had been broken. The full case study is contained as an appendix.

In this case a UK based company had expanded overseas and established local development teams. Initially these teams localised the software but over time they each maintained their own version of the software and the shared, single, code base started to break down as each team operated with their own copy of the source.

This scenario is completely compatible with Conway's Law. Communication within each team was better than between teams so the system architecture fragmented into multiple versions of the code base - one per team.

As the fragmentation increased so did costs and eventually the senior management decided to break the cycle. An active decision was made to support only one code base. The company decided to adopt a variant of *Extreme Programming*, which would be adopted by all teams. Some teams where disbanded altogether and those that remained where brought together to create new communication channels.

Once the team members had all met and been trained in the new methodology they returned to their development centres - now reduced to three, UK, West Coast USA and Singapore. These teams adopted the new working style and created a new communication path by participating in thrice daily hand-over meetings on video link up - once at the start of each teams work day and once at the end.

To date the new way of working has been successful and there is a single, shared, code-base.

At least superficially this study presents a case where Conway's Law was broken. The teams had been overcome by communication paths that created an architecture. Management had intervened and changed the architecture to overcome the power of Conway's Law.

Herein lies a lesson for organizations outsourcing and/or off-shoring: you are inserting a barrier between two groups, you will need to replace informal communication structures either with formal structures or technology which allows new communications paths to replace the old.

Lesson #1: Informal communication is important in developing software. If a barrier to informal communication is created (e.g. outsourcing or off-shoring) it is necessary to compensate for the barrier. Failure to do so will affect the software architecture.

The system concept

Conway does give us a "get out of jail free" card:

> "the need for communication depends on the system concept in effect at the time"

By changing the system concept we can change our system and re-direct our communications. For example, a team focused on performance will communicate with different people to a team focused on deadline.

If we return to the ABC case study we see we can also explain the case study in terms of the system concept. Management decided to intervene, they changed the system concept - in part through the adoption of a new methodology - and remade the communication channels - some where removed completely, others where improved (personal contact as in the Herbsleb and Grinter study) and new ones introduced (e.g. video handovers).

One question remains: *Does ABC still obey Conway's Law?*

On the one hand it broke the law because it escaped the tyranny of the law. On the other hand, the new situation still obeys Conway's Law, architecture is still following communication but we have changed the communication path.

Lesson #2: We are not powerless; we can intervene and change the system concept. Removing barriers requires an active intervention.

1960's Context

It became clear during the focus group discussion that the organizations that Conway was thinking of were typical for the 1960's. They had a strong hierarchy, and the communication structure was basically a mirror of the reporting structure. Of course these organizations still exist today. The defence industry, public offices, and those who insist on sticking to waterfall development methodologies are good examples.

But many modern software development organizations allow for completely free communication independent of reporting lines. Developers are actually expected to contact whoever they need to for their work. Working across geographic locations and time zone do impose some boundaries.

At the same time communication costs have fallen, not only does every developer have a phone on their desks but they probably have a cell-phone, instant-messenger, global e-mail, conference calls and video links. Not only has the variety of mediums increased but the cost has fallen by an order of magnitude.

The proliferation of plentiful - and cheap - communication channels does not guarantee communication but they are a pre-requisite for good communication. As both the *ABC* and Herbsleb and Grinter studies show these channels are more frequently used when individuals meet and know the other individuals.

Lesson #3: Give developers good communication links and ensure they meet one another - especially those who are based remotely.

This echo's Coplien and Harrison's *Face to Face Before Working Remotely* (Coplien and Harrison, 2004).

The Homomorphic force

Conway also gives us the *homomorphic* force:

> "This kind of structure-preserving relationship between two sets of things is called a homomorphism." (Conway, 1968)

The idea of homomorphism is established in mathematics (see http://en.wikipedia.org/wiki/Homomorphic) and the authors believe it can be seen in other fields (e.g. biology, social science) but further research is required to validate this claim.

This is the reason system designs become a copy of something else. We need to understand this force in more detail - this is an area of research.

The homomorphic force is in effect the thing that many people are actually referring to when they say "Conway's Law" - it is clear that Raymond's version of "Conway's Law" is actually describing the homomorphic force.

Once we accept this we can now understand Conway's 1968 piece in three layers:

- Lowest level: Conway introduces us to the *Homomorphic force* and describes some of its effects.
- Middle: Conway's broader argument encompassing the homomorphic force, division of a system along communication paths (akin to Parnas' responsibilities) and effect of the *system concept*.
- Top: Conway's entire article, which adds his 1968 context, his foreshadowing of Brooks Law (Brooks, 1995) and his solution in calling for "lean and flexible [organizations]."

It would also appear that the homomorphic force has an additional characteristic that makes it extremely powerful: it is self-reinforcing.

We know already that software architectures are difficult to change, as are organizations. We can reason that the two are mutually supporting. Once software takes on the characteristics of the wider organization it will serve to reinforce the organizational structures, which in turn will strengthen the software architecture.

Barriers that are created this way will be extremely difficult to remove. An attempt to change the architecture will conflict with the organizational structures and vice-versa.

Because of the self-reinforcing nature of the homomorphic force organizations will still copy themselves in code unless this and other forces are actively considered and managed (i.e. used or countered).

Beyond homomorphism and communication

Given the power of the homomorphic force it is important we seek to increase our understanding of these other forces and how they affect our designs.

Homomorphism is not the only force which effect system design, there are others. Nor are communication paths the only factors that may be reflected through the homomorphic force into system architecture. Unfortunately the homomorphic force is so powerful that other forces may be ignored and left unbalanced. This can cause problems later in the system history.

Other forces may include:

- Financial forces, e.g. in an attempt to keep its wage bill low a company may only hire recent graduates to develop software, consequently many practices of mature developers are absent. Use of outsourcing IT partners and/or global distribution of team members create new challenges and strongly affects the communication structures.
- Political forces, large corporate will often mandate a particular OS or language be used on all projects.
- Organizational forces, e.g. how democratic or bureaucratic the organizational culture is. Bureaucracy may interfere with communication, financial decision-making, recruitment, training and any number of other factors.
- Communication forces, where the reduced cost of communication has enabled quick and easy access to a much larger number of people involved on many levels and in numerous roles with a project, while the ease of communication means that it is hard to control who talks to who even if the desire to control is there from the management.
- Cultural forces, e.g. the relationship between younger engineers and management has changed considerably since 1968. This impacts communication but also decision making and the architectural freedom of the development team.

We could list more forces or go more in detail, but choose not to since it does not reflect the work of the focus group. One thing the group did agree on was that technical considerations play a relatively minor role in the design of a software system compared to external forces. What the discussion on other forces told us is that organizations have changed since 1968 (or at least some have, or maybe they are new and different organizations). This is changing the context and the relative strength and effect of the dominant forces.

It would seem that when Conway wrote his article in1968, a number of forces where all pushing in the same directions, for example, hierarchy, bureaucracy and expensive communication. Many of these forces have changed since 1968. Forces that were strong are now weak (e.g. hierarchy) while forces that where weak are now strong (e.g. corporate democracy.) Other forces have changed direction, e.g. communication as discussed above.

Of course these forces will vary from company to company and department to department. One may still find highly hierarchical organizations (e.g. the civil service) but they are no longer the rule.

We should also seek to increase our understanding of the user role in system design. This is particularly important in the contexts of ERP, organizational change and markets. The authors hope to explore this dimension in a future.

Homomorphic designs

Designs that are overwhelmed by the homomorphic force can be termed "Homomorphic designs."

Unless the organization is structured to support an optimal design and is willing to change as the architectural needs are changing (which we believe is rare), homomorphic designs are most likely not optimal. Better designs can be produced if the organization makes an effort to consider other forces - both in software and process design.

Lesson #4: If we do not choose to actively break homomorphism it will take control of the architecture, so we must actively choose to design systems that break homomorphism.

The problem is: homomorphism is a very strong force but it does not result in good solutions because it does overwhelm other forces.

(For a few (dysfunctional) organizations it may be that embracing the homomorphic design and allowing the organization and software to mirror each other may actually be an improvement over the current status quo.)

Structure is about barriers and balancing forces. Designing software is about choosing where to place barriers; homomorphism will reduce your freedom in making balanced choices in where to place these barriers. Once in place these barriers grow in strength, but some barriers will be in the wrong place and will become obstacles.

People will also create barriers themselves: to protect themselves, to protect their teams and avoid anxiety - sometimes called *social defences*. In a case study Watsell describes how software developers can create defences:

> "The argument is, thus, that methodology, although its influence may be benign, has the potential to operate as a 'social defence', i.e. as a set of organizational rituals with the primary function containing anxiety. The grandiose illusion of an all-powerful method allows practitioners to deny their feelings impotent in the face of the daunting technical and political challenges of systems development." (Wastell, 1996)

Obviously, such barriers often become obstacles.

We, as system designers, need to know when to create a barrier and when to remove one, when to reinforce one and when to weaken one. To do this we work in two domains: the technical and the social.

The *ABC* case demonstrates this. The geographically different groups built their own defensive barriers - in code. Eventually the manager decided to blow up the barricades - both social and technical behind which people were working. Simultaneously they introduced new communication channels that allowed technical changes.

The same is true in the Herbsleb and Grinter study: after engineers had visited the other site things improved because barriers had been removed.

In complex organizations it can be difficult to tell which is which. It can be difficult to tell which is a barrier preventing change and which is enhancing the design. Removing barriers can destroy designs and loose knowledge.

This is the mistake made by Business Process Re-engineering. Barriers were removed and new ones added and all the time knowledge (particularly tacit knowledge) was lost.

One way around this is to create a culture where change is accepted and understood. This means a culture that learns. Not just the individual but also the team, the organization. A learning team will be a flexible one, a team driven to learn and improve will be lean - we know this from Toyota (Kennedy, 2003, Womack et al., 1991, Cusumano and Nobeoka, 1998).

And this is what Conway calls for: organizations that are lean and flexible.

Conway was right but things are more complicated than he thought. In the twenty-first century we need a new understanding of his argument for a new context.

Appendix A: *ABC* Case study

ABC began as a British company based in the UK. They proceeded to own offices in the US, in Latin America (sales only), South Africa, Singapore and Australia.

The development organization was structured as follows: a "Global Team" group in the UK doing the core system; this was the oldest and most experienced development team, and they were close to experienced customers. The other sites had satellite development teams doing extensions, branding and adoptions for local markets. As an example, Singapore had to deal with several languages and became good at localization. South Africa had infrastructure problems and had to duplicate databases. The team in Australia used different technology and developed their own functionality.

After a few years, started to observe that work was duplicated, the cost was high, and it was difficult to upgrade to new core updates. As a reaction, the "Global Team" plus local team in the UK started to put up barriers to stop changes in the other locations. Feedback loops with the other teams going on to incorporate local changes into upgrades. Manager felt he was paying 5 times for the same feature, and other locations could not benefit from a feature developed at a certain location.

The manager asked the Head of the Global Team to "fix" it. The decision was made that there is no local ownership. There is one code base. Each market gets the advantage of all development independent of location. No cracks because of local development.

To implement, all developers were brought together for 6 weeks (about 30 people). A new architecture was put in place that was a better fit for the product. This architecture reflects the problem domain and not the structure of the development team.

Shortly afterwards the company found it necessary to reduce the size of the team with some positions becoming redundant. Preference was given to retain individuals who would fit best with the new collaborative environment. Of those who left the team some where retained on other projects in the development group.

Today all the teams are working on a shared code base. If one team wants to make changes:

They just do it – and if it is not good other teams will remove the change

At the end of each day there is a videoconference with other teams, due to the time difference this works like a sliding window around the world

For big changes they do a position paper on the team wiki. There is no lead architect. The global team (20+ people) will go on until they reach consensus in the discussion.

The system that is being built is deployed in live situations.

The top level of management is where the decisions meet.

Needed someone from the outside to do the change.

Appendix B: Further comments

When this report was complete the authors circulated it to the focus group participants and asked for additional comments for inclusion. We also asked Neil Harrison and Jim Coplien, authors of the *Conway's Law* pattern for comments.

Kevlin Henney and Keith Braithwaite, both of whom participated in the focus group, and Neil Harrison responded and we include their comments here.

Comments from Kevlin Henney

I first came across Conway's Law in the New Hacker's Dictionary(Raymond, 1996), where it was presented as "the rule that the organization of the software and the organization of the software team will be congruent". I later came across it again presented as a pattern (Coplien, 1995). And then just a couple of years ago I had the good fortune to read the original paper by Conway (Conway, 1968), followed by the more detailed pattern write up in Organizational Patterns of Agile Software Development (Coplien and Harrison, 2004).

There have been a few things that have troubled me about the various billings of Conway's Law, and many of them simply come down to the (mis)use of the word "law". One sense of the word "law" is a strong rule that is agreed upon in some social structure, with the implication of judgement and penalty for any violation. Another common sense is in the sense of physical law, for which there is no concept of violation. It is quite clearly possible to break Conway's Law both with and without penalty, depending on other factors. This suggests that the term "law" is quite the wrong one, and what is being described here is a force. It also means that it doesn't make as much sense to talk about breaking a force as it does a law. Admittedly, Conway's Force sounds less catchy, but does appear more accurate.

When characterised as a force many things fall into place. It is not the sole determinant of software architecture, so in shaping an architecture it is one consideration of many. Its overall effect will relate to the strength of the other forces at play. This seems more in keeping with how development projects are observed to unfold. If the force exerted by the organisational communication paths were the only force, then there would be nothing more to software architecture than the communication paths in the organization. Of course, this is not what is observed.

For example, in the limit, a one-person project would produce, according to a strict interpretation of the force as a law, no modular structure in its software architecture.

Another example that should result in a flat architecture would be a small, close-knit team where code is shared transparently and effectively. That this is not always the case demonstrates, by simple contradiction, that what is at

play here is no law. That this is sometimes the case does, however, highlight that there is something important here that cannot be ignored: organisational structure is a necessary (but not sole) consideration in software architecture. Hence why it makes more sense to look on it as a force rather than a law. It is a force that is not hard to observe in action, but it is also easy to see it come into conflict or sympathy with other forces.

From a pattern perspective, this idea of a force makes perfect sense: conflicting forces make up the tension in a problem that the proposed solution is intended to resolve. However, this perspective does mean that Conway's Law (Coplien and Harrison, 2004, Coplien, 1995) as a pattern is named after a problem force rather than a solution structure. This in turn highlights another issue: as a pattern, if what have come to know as Conway's Law is a force, what exactly is the solution?

The earlier version of the pattern (Coplien, 1995) was also documented with two synonyms: Organization Follows Architecture and Architecture Follows Organization. These are very descriptive, especially for someone who may be unfamiliar with the Conway connection, and I have found them helpful when reading and referring to the pattern. However, these two names also describe two quite distinct solutions with quite different characteristics. The first suggests that the architecture of the software dictate the structure of the organization around it, and so the organisation is considered to be subordinate to the software. The second suggests the converse. The advice is quite different in each case, although both cases pursue a common cause of architecture-organisation alignment. So is this perhaps two patterns rather than one?

The pattern's solution text is a little ambivalent, suggesting that the important feature is that the organisation and software architecture are aligned, and that a secondary consideration is that the software should probably, but not necessarily, drive the organisation. This description is also in part true of the later documentation of the pattern (Coplien and Harrison, 2004), but importantly this later description also states that "care should be taken to align the structure of the architecture with the structure of the organization by making piecemeal changes to one or the other". And in this statement lies an apparent resolution: there are (at least) three patterns at play here.

The root pattern we can name *Align Architecture and Organization*, and this captures many of the points that the original versions of the pattern focus on as issues and benefits.

To *Align Architecture and Organization* we can make changes so that *Organization Follows Architecture*, or so that *Architecture Follows Organization*, or both, responding to feedback from each previous change and other factors at play in the architecture and organization. This separation makes clearer the distinct and complementary approaches at work here, rather than including them in the slipstream of the root pattern. Hence, there is more scope for discussing the interaction and choice of pattern application in a given situation, and I believe that such clarity and generatively is no bad thing.

Comments from Neil Harrison

First, I have no feedback on the report itself; as far as I can tell, it is an accurate report of what went on in the workshop.

Naturally, I do have feedback on the content of the workshop. Whether you add or change your report based on my comments is entirely up to you.

(Authors note: We chose to leave the text unchanged.)

By nature, workshops tend to be an exchange of ideas. Opinions are often given without much evidence to back them up. This is appropriate. This workshop was no exception. In particular, one person presented a single case study which was supposed to call Conway's law into question.

Conway's law has been around for over 35 years. We have seen it over and over again in our studies. And we aren't alone. One counterexample does not invalidate a widely substantiated theorem. But even more to the point, when I read the case study, it didn't contradict Conway's law at all! In fact, it reinforced it. Note what happened:

> Separate teams worked on separate parts of the system (Conway's law), until the architecture deteriorated enough that they had to do something (note the lack of *Architect Controls Product*.) Then they redid the architecture by basically creating a single blob. And then - this is important - they redid the teams by bringing everyone together (*Lock Em Up Together*, *Face to Face Before Working Remotely*). After the team size was reduced they were down to 15 or so people. That's about the size of one team...

Note that *Lock Em Up Together* is about forging team unity as much as forging architectural unity.

Now it's hard to have a single team that is geographically distributed; there is a natural tendency to form local teams. And local teams will eventually cause architectural drift. The company knew this, so they took actions to prevent it. They have daily meetings, which enforce the single team concept. They are bucking natural tendencies, so they have to take overt, constant, action against it.

This will be hard for them to sustain.

By the way, note also that there is no lead architect, and the top level of management is where the decisions meet. This means either that the top manager is the de facto architect, or they again have lack of *Architect Controls Product*. Either way, that sounds potentially troublesome.

Other stuff:

There appeared to be general agreement in the workshop that things have changed since the 1960s; that teams have become less hierarchical, that communication channels have increases, and that Conway's Law was most appropriate for the old teams of the 1960s. So it needs rethinking in the modern world.

These are opinions. (Again, a workshop is all about discussing opinions, so this is perfectly all right.) My opinions on the above are that I disagree with basically all of them.

In the 1960's, many software developments were small; relatively few were large. Small developments had small teams with flat (or nonexistent) hierarchies. Now, many software developments are still small, in terms of people. Relatively few are large. Small developments have small teams, and large ones have large teams.

Communication has indeed increased. At the same time, we have lost much of the advantage by distributing teams, which decreases the quality of communication. We distribute teams in ways no sane organization in the 1960's would have. At best, it's a wash, but we may be even worse off than we were before, in terms of the quality of information flow.

So I guess I disagree with your last two sentences: I think that the changes that have happened in the last 35 years pale in comparison to the things that have remained the same...

Comments from Keith Braithwaite

I first came across Conway's Law while pursuing a Master's degree in Software Engineering, around a decade ago. During the following ten years I somehow did not notice that I had arrived at an opposite understanding of the "Law" from most observers--until attending Allan's and Lise's workshop where I discovered to my surprise that I was the only person there who considered that the Law described a failure mode. Rather, in my mind, after the fashion of the laws of Brooks and Parkinson. Yes, the phenomenon described by Conway's Law occurs, it even is widespread, but it's neither inevitable nor desirable. That was my view. Yet the common view amongst the workshop attendees was that the Law described an outcome so desirable that even if it did not arise naturally (as it was almost bound to do), then it should be induced. Puzzling, stimulating and challenging!

It so happens that my first work as a professional programmer was to do with compilers, and so the common shorthand form of the law, suggesting that four teams will build a four-pass compiler very obviously to me described a pathological situation, in which a far-reaching technical decision ("architecture" indeed) was driven by an artefact of staffing. And so with the Law more broadly understood: what are the chances that the reporting lines of a development organisation map onto the architecture of a system to be built in a way most beneficial to the customer? Why would a competent development organisation allow such a constraint to apply.

In preparing for the workshop, I read Conway's original paper. It asserts that there will be a 1:1 mapping between the org. chart of the organisation building a system and a graph of the information flows within the system. We did see this in operation at *ABC*: a hub-and-spoke organisation of the business lead to a system built from a core component with local additions.

This failed for us, so we chose to do something different. Amongst other things, we chose to decouple the architecture of the system from the structure of the business, in the hope that they could then both find their best respective forms.

In Conway's model a developer A working on a module can only execute work to agree an interface with another module worked on by developer B by appeal to the "least common manager" of A and B, C. And thus the homomorphism described arises, with the dataflow between the two modules corresponding to the reporting lines from A and B to C. Again: why would you put up with such overhead? At *ABC* we do not. If a developer working on a story (the implementation of which might cut across multiple modules) needs to agree an interface (or anything else) with another developer, then they simply do so, peer-to-peer. The least common manager likely won't, doesn't need and probably doesn't want to, know about it. The developers can, if they are in the same site, simply talk to each other. If they are in separate sites, they can communicate with one another freely via the (single) code base, via email, via the team wiki, via the twice-daily videoconference, via distributed pair-programming sessions. At the time of writing, this practice has been in place for almost two years, and continues to work well. Meanwhile, the system architecture is what(ever) it needs to be.

During the discussion of this case in the workshop it was suggested that this was indeed Conway's Law in action! One team, one code base, therefore homomorphism. If this is Conway's Law, then I would suggest that the law is content free. The suggestion that "architecture is still following communication" in the *ABC* case simply makes no sense to me.

Our single global team is not atomic, rather it has an internal structure--three regional sub-teams, each with a local line manager who reports to the group CTO. Communication between the three regional groups is structured around the twice daily videoconferences. Our code base similarly has structure. It is built out of several relational schemas, a J2EE container full of beans, a substantial library of POJO's, servlet engine and contents, Velocity templates (and not just for web presentation), an extensive MVP framework, O-R mapping layer, stand-alone Java applications, Ruby on Rails web apps, stand alone servlets, etc. etc. But there's no partition of anything in the code into three to match our three local groups of developers. And there is no hub-and-spoke (CTO<->regions), and there is no three layers (CTO<->Regional Head of Development<->Developer).

There continues to be no architect (and the CTO is most definitely not one). Team members around the world continue to spike, lobby their peers for, and execute architectural change as and when they see fit, to better serve their customers and users.

Our architecture does not follow our organization, nor vice versa other than in the most trivial ways. For instance, we have RDBMS's, and we have DBAs to run them, but a developer can institute a schema change, or introduce a new schema even, without consultation via the least common manager.

I invite anyone who maintains that we exhibit Conway's Law to come learn about the structure of our system (this week :) and the structure of our team, and then present back to us the homomorphism they find. I'd expect this to be a short presentation, but I'll be happy to be surprised.

We have chosen not to accommodate the homomorphic force that would tend to distort our architecture to match our organisation, or conversely our organisation to match our architecture and either way make both unreasonably hard to change (as noted in the workshop outcomes). This can be done, and based upon my experience at *ABC*, vs. experience elsewhere (particularly with other distributed teams) must be done. And I remain convinced that we have done it.

Bibliography

Brooks, F. 1995 *The mythical man month: essays on software engineering,* Addison-Wesley.

Conway, M. E. 1968 How do committees invent?, *Datamation***,** http://www.melconway.com/research/committees.html.

Coplien, J. 1995 *A Generative Development-Process Pattern Language* In *Pattern Languages of Program Design*(Eds, Coplien, J. O. and Schmidt, D. C.).

Coplien, J. O. and Harrison, N. B. 2004 *Organizational Patterns of Agile Software Development,* Pearson Prentice Hall, Upper Saddle River, NJ.

Cusumano, M. A. and Nobeoka, K. 1998 *Thinking Beyond Lean,* Free Press.

Herbsleb, J. D. and Grinter, R. E. 1999 Architectures, Coordination, and Distance: Conway's Law and Beyond, *IEEE Software,* 16, 63-70.

Kennedy, M. N. 2003 *Product Development for the Lean Enterprise,* Oaklea Press, Richmond, VA,.

Parnas, D. L. 2001 *On the Criteria to Be Used in Decomposing Systems into Modules* In *Software Fundamentals: Collected Papers of David L. Parnas*(Eds, Hoffman, D., M. and Weiss David, M.) Addison-Wesley.

Raymond, E. S. 1996 *The Hackers Dictionary,* MIT Press, Cambridge, MA.

Wastell, D. G. 1996 The fetish of technique: a methodology as a social defence, *Information Systems Journal,* 6, 25-40.

Womack, J. P., Jones, D. T. and Roos, D. 1991 *The machine that changed the world,* HaperCollins, New York.