Agility is a Stable Requirement

Graham McLeod inspired.org
December 2017

Abstract

Change in technology, business and society is ever present and accelerating. It is very unlikely to slow down, thus it is a *stable requirement*. Our methods of doing strategy, devising future architectures and delivering systems capabilities in support of business processes, capabilities and delivery of services and products therefor need to address this.

A great deal of effort has been applied in Agile Methods over the past two decades to accelerate the system development process, i.e. doing things faster. No matter how quickly they deliver, however, these methods often produce something inflexible. This paper argues for a broader approach, which looks at: the context (much of the change required is outside the system delivery space; the focus (what should we be changing and why?); and three approaches to achieving the change with respect to system deliver: doing less things (de-scope, use packages, libraries, components, frameworks); do things faster (agile methods, automation, generation) and make more flexible things (runtime adaptable or domain model driven systems). The last of these is an unconventional approach that holds promise, even if you currently don't practice, or succeed with, agile methods.

Finally, we also address the dilemma of accelerated delivery while dealing with large legacy application landscapes.

Intentionally blank

Table of Contents

Agility is a Business Requirement

Very few would dispute the need for agility in today's business world, government organizations and NGOs. Agility is needed to respond to increased competition, globalisation, governance/regulation and changes in technology, customer expectations and business models. No one wants to be a Kodak in a digital photography world, or an Encylopaedia Britannica when Encarta or Wikipedia obliterate your business model.

Given that technology, business and societal change is ever accelerating, it is highly unlikely that the need for agility will ever decrease. This leads to the surprising conclusion that *Agility is a Stable Requirement*.

If it is a stable requirement, we should be able to devise ways of achieving it. The rest of this paper examines the implications of this realisation and explores ways to achieve agility.

Defining Agility

For the purposes of our discussion, we define agility as

The ability to successfully adapt to new circumstances or opportunities in a rapid and sustainable manner

For a business this includes the ability to adapt or adjust:

- Business Model and Strategy
- Organization Structure, Culture and Staffing
- Business Processes
- Products and Services
- Channels for Delivery
- Arrangements with Business Partners
- Skills and Techniques
- Supporting Systems, Information and Technology

We can represent the different focus areas in the following graphic:

Figure 1 - Dimensions of Change to Achieve Business Agility

Notice that IT/IS is only one of many aspects that require agility. Achieving agility in the other areas is mostly a concern of disciplines including:

- Strategy/Business Architecture
- Change Management/Programme Management
- Organization Design
- Process Engineering
- Product and Service Development
- Contracting
- Training
- Enterprise Architecture
- Project Management

Culture is also a very important element in achieving agiity. An empowered workforce of small teams with interdisciplinary skills working in a learning organization culture will definitely outperform an autocratic formal one.

For the IT/IS element, agility includes adapting:

- Technologies in use
- Infrastructure supporting the business
- Application systems capabilities
- Information capture, storage, processing, sharing, analysis and visualisation capabilities

Bearing in mind that IT/IS is only a fraction of the full picture, we will zoom in on approaches to achieving agility in this dimension. We plan to address the other dimensions in future papers.

Agile Methods (Doing things faster)

Many organizations have adopted Agile Methods (e.g. Scrum, Disciplined Agile, Scaled Agile, Extreme Programming, Crystal *etcetera*) as a means of achieving agility. The earlier methods and versions of methods primarily focussed on the rapid, incremental delivery of application systems. While relatively successful in this goal, we can immediately appreciate from the foregoing, that this is a small part of the overall scope mentioned above.

The Agile Manifesto, drafted by a group of software development luminaries in 2001, set out some important principles.

Figure 2 - Agile Manifesto

Note that it *emphasises* the priority of items on the left over those on the right. It does not, as many adherents proclaim, *eliminate* the items on the right. Note too, that it addresses Software Development. In our view, a method should target Solution Development which is broader than software only, in that it will address other elements required to solve the business problem, including:

- Documentation sufficient for the installation, operation, support and enhancement of the delivered system by parties other than the developers
- Procedures around the system to ensure its successful operation in its intended context
- Conversion of data to the system
- Integration of the solution to the context in which it must operate
- Identification of reusable components and services that may be leveraged in other efforts
- Quality assurance through testing and other means (e.g. inspections) to ensure the correctness, robustness and compliance of the solution

Agile Methods target, *inter alia,* the following goals:

- Rapid delivery of systems capability to the business
- High levels of sponsor and user engagement in decision making during the design, build and testing activities
- Clarification of unclear or unknown requirements through mock-ups, prototyping, facilitation and iteration
- Gaining experience of new approaches or technologies early to facilitate learning and eliminate reliance on untested assumptions
- Visibility of progress

- Collaborative teamwork

They use a variety of techniques and practices to facilitate or achieve these goals, including:

- Feature Requirements Lists or Issue Backlogs - to capture and agree requirements and allow grouping of these to scope an iteration, agreeing priorities with sponsor and users
- Sprints - intense bursts of activity with very focussed goals, including delivery of working software
- Standup meetings - to share progress, encourage collaboration, reduce meeting times and keep administration overhead low
- Visible reporting (e.g. Burn down charts) to make progress and status visible and shared and to focus effort towards goals
- Pair programming - to apply more than one mind to the problem and generate backup skills / provide coaching
- Continuous planning, integration and (automated) testing
- Test Driven Development (TDD) where tests are developed first, and software subsequently to pass the tests
- Lean concepts which include the ideas of doing just enough to be useful and gauging success before committing more resources, or pivoting to a more useful direction. These and kanban concepts also favour limiting work in progress to ensure that focus is high and productivity is improved by limiting multi-tasking
- Elimination of any "might be needed" requirements - the term YAGNI (You aren't gonna need it) has been coined to summarise this, in the belief that users, analysts, designers and programmers are poor at predicting the future, so we should not waste time on any requirements that are not absolutely known to be essential

Figure 3 - Typical Agile Lifecycle
Source: Scrum Alliance

Agile requires certain factors to permit its success, including:

- High level of trust between the sponsor and development organization. After all, there is no contracted value/scope for the money that will be spent, as in traditional contracting / waterfall. Rather, the sponsor commits to funding the developers at a certain rate in the faith that they are competent, will be productive and will focus on the right issues to deliver value to the business. This kind of trust takes time and good history to establish
- Agile methods are no substitute for good skills. In fact, successful use of Agile will often require higher skills, since team members must exercise much more individual judgement and discretion. Agile should thus not be attempted with a team of juniors with little experience. They can certainly be included in a team with more experience, but having experience on hand is a pre-requisite for success
- Agile relies a lot on people. Continuity of staff and full time commitment to the project are necessary to ensure that the tacit knowledge in play is not lost or diluted

Dangers of "Agile"

Agile is sometimes used as an excuse for various ills. Common issues arising from the abuse or misunderstanding of Agile include:

- Lack of architecture - assuming that we can "build it as we go", resulting in poorly architected systems that do not adhere to good principles and will be difficult to maintain in future or which do not fit into the context where they must operate. In fact, better agile methods *do* cater for architectural requirements, e.g. by doing an architecture "spike" before commencing detail development
- Lack of documentation - Although Agile favours people and interaction over administration and documentation, it does not say we should *not* document. Remember that we need to produce an installable, operable, maintainable solution
- Rehashing requirements that we already know. Yes, Agile is a great way to refine unclear requirements, or discover unknown ones. That does not mean that <u>all</u> our requirements are unknown or unique to this project. We can certainly save a huge amount of time and effort by simply documenting our known requirements or using established frameworks, reference models and patterns
- Under-delivering on scope. Agile projects will often cut scope to meet a deadline. That can be a good technique, provided that we do not do it to the detriment of the business requirements and priorities. We have seen Agile projects that reduced the scope beyond a "minimum viable product (MVP)" to deliver something not useful to the business
- Focus on functionality at the expense of other requirements, such as integration to other systems, performance, robustness or scalability. We must pay adequate attention to non-functional requirements as well. Failure to do so can ultimately doom the project and solution

Newer Agile Methods (such as Scaled Agile (SaFe) Disciplined Agile (DaD)) are addressing issues beyond just software, and increasingly, issues such as architecture and fit to organizational objectives. These are encouraging developments. In the next section we will look at ways to achieve agility and where Agile Methods fit in this.

Achieving Agility in Software Solutions

The primary approach followed to improve agility in software is to try to reduce the time between identifying a need and delivering a solution. The traditional "waterfall" lifecycle had a series of steps, more or less in sequence with a "flow" from earlier to later stages. Discovery of new requirements or disproving assumptions later in the lifecycle, often required "cycling back" to an earlier stage and a measure of rework to continue. The overall lifecycle could take a while to complete, often to the frustration of business under pressure for rapid calendar time delivery.

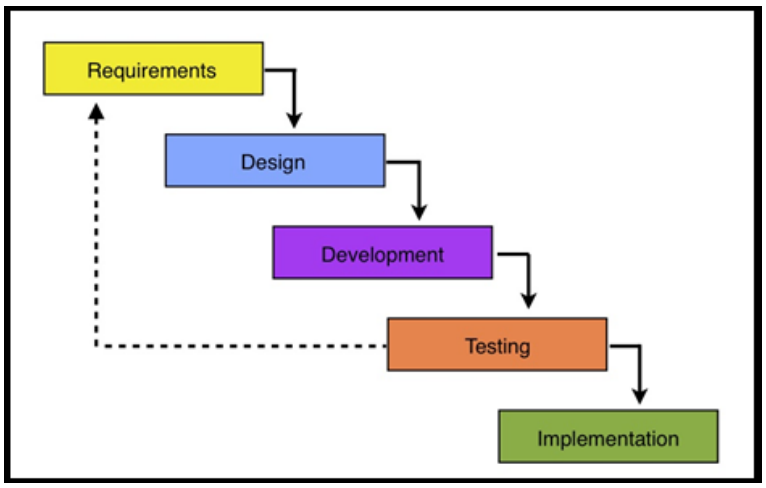

Figure 4 - Typical "Waterfall" Lifecycle

Iterative and incremental forms were evolved to improve the delivery. In these approaches, a subset of requirements would be selected, developed and delivered, before cycling back to tackle more requirements. This had the advantage of earlier delivery of priority items and being able to apply learning from earlier iterations to later ones.

These approaches worked well, provided that the requirements were fairly well understood. However, many classes of problems arose (especially as Information Systems and Technology expanded into new types of applications never before contemplated) where the requirements were not understood at the outset. In these cases a more facilitative / prototyping / experimental style emerged that led to "Agile" methods, which assume that requirements are not understood and will only become clear during the project.

Agile emphasises high sponsor/user involvement, teamwork, communication, delivery and feedback in rapid iterations. Scope may be adjusted in each iteration in consultation with the sponsor and learning from earlier iterations is fed into subsequent ones. Each iteration has elements of requirements, design, development, testing and may deploy working software. We contrast the waterfall and agile lifecycles below:

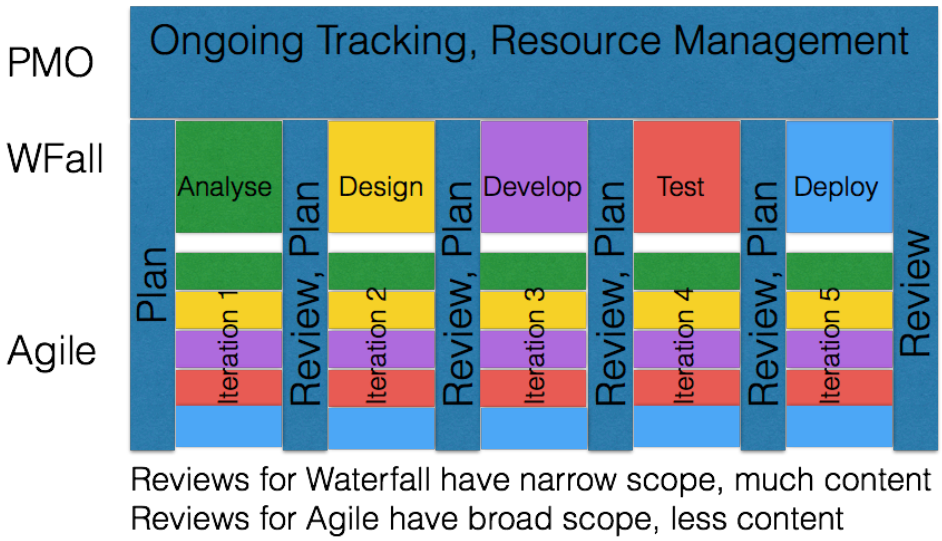

Figure 5 - Contrasting Waterfall and Agile Lifecycles
PMO = Project Management Office

Delivering More Rapidly

A lot of method emphasis over the years has been on reducing the time to delivery from identification of a business requirement. We can see this progression as follows:

Figure 6 - Traditional Waterfall Lifecycle

In the traditional waterfall cycle, deployment occurs quite late in the cycle and a long time after requirements are first discussed. This can be a problem for business if the requirement is urgent, e.g. complying with a sudden legislative change. Other problems that can arise are that the requirement *changes* or becomes obsolete before the delivery of the software capability. This can lead to a great deal of wasted effort and investment. If the requirements are poorly understood, or unclear, the risk of delivering something that is not optimal is very high. Fixing it can also incur a long delay and high cost as we wait for another cycle.

To address these concerns, the iterative incremental approach broke requirements up into priority (and dependency) subsets, each of which could be taken through the lifecycle in a shorter period. This is illustrated below.

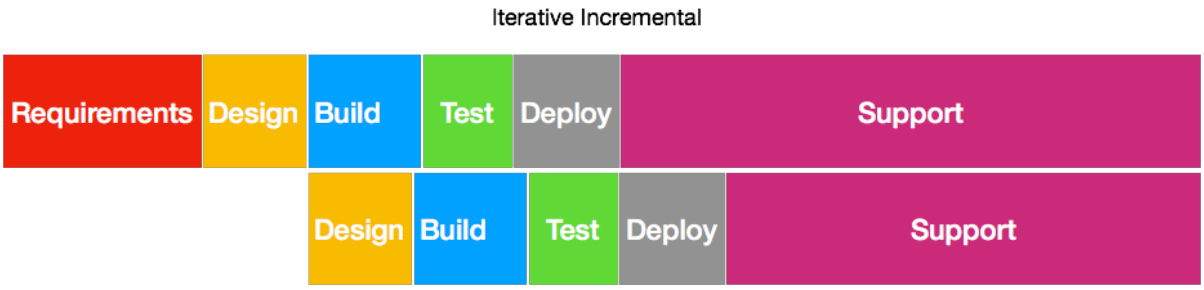

Figure 7 - Iterative Incremental Lifecycle

Note that the phases are still there, but they are shorter since each is tackling a smaller scope. There is earlier and more frequent delivery. As in the figure, there is also opportunity to overlap the development if multiple teams are available.

Agile methods recognise that requirements are often unclear or may evolve during the lifecycle, so they accommodate requirements change both within and between iterations, See below.

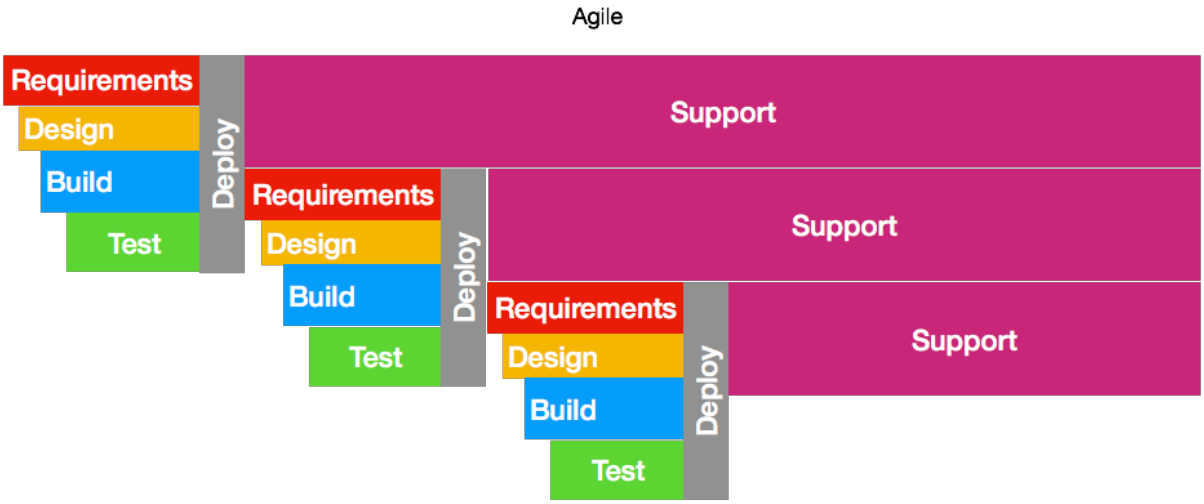

Figure 8 - Agile Lifecycle

By allowing this, they overcome the problem of wasted effort on incorrect or obsolete requirements. As we mentioned previously though, there are a number of preconditions to getting Agile right, as well as dangers from its misapplication.

The combination of Agile Methods, Virtualised infrastructure, Integration between Agile Development, Automated Testing and Deployment has led to the DevOps movement, where Development and Operations are brought much closer together.

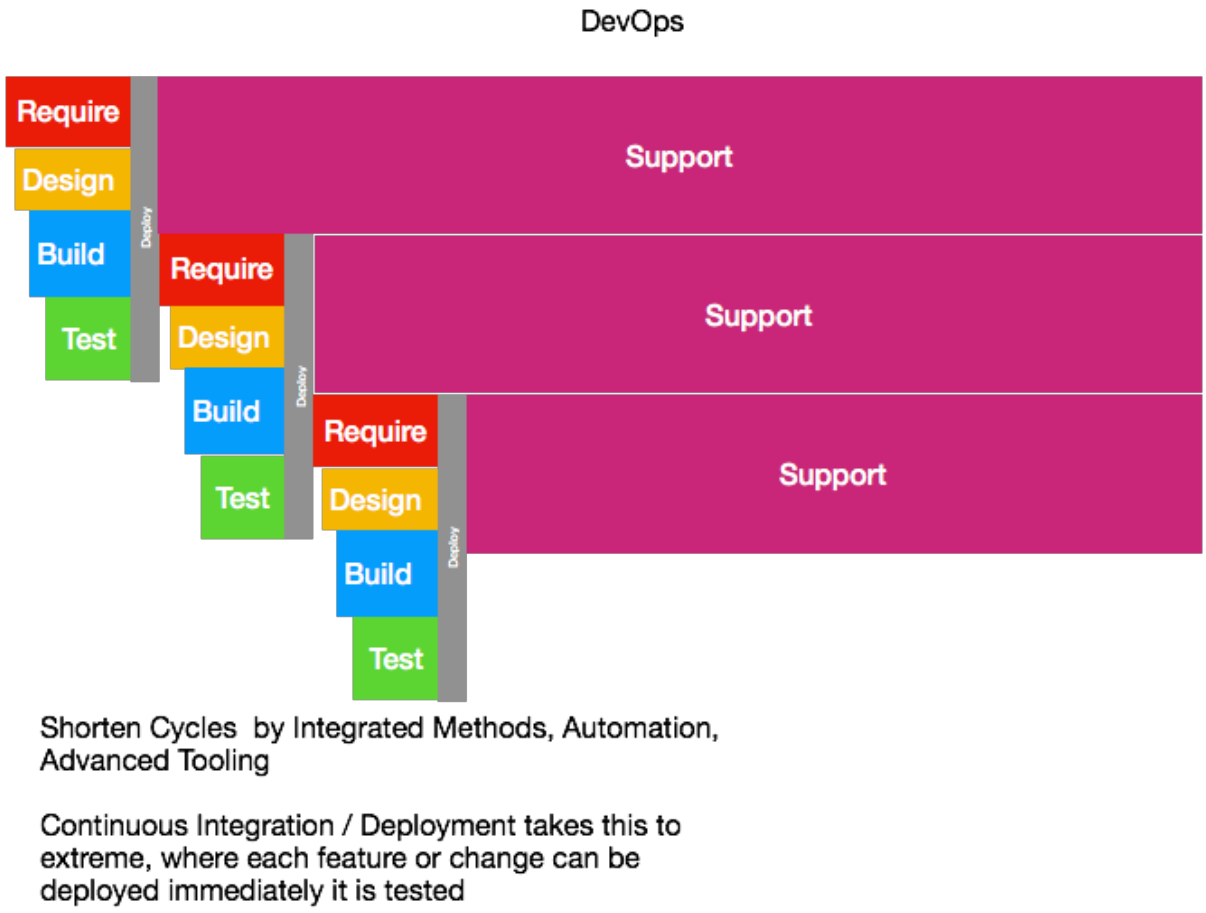

Figure 9 - DevOps Lifecycle

The requirements to deployment cycle is shortened by integrated methods, high collaboration across functional areas, and advanced integrated tool chains that can take the output from development tools, feed it to test suites and the items that pass testing to operations in a seamless process. Continuous deployment is an extreme form of this where each feature or change can be deployed immediately it is tested in a "mini release". This will often also require the adoption of service oriented approaches so that deployment of components can occur without needing a complete monolithic product to be built or linked. In advanced shops, such as Facebook or AirBnb, there could be several dozen "pushes" into production on a daily basis. Google, for example, may deploy several different change scenarios in parallel and monitor which are more successful in production, before decommissioning the less successful candidates.

Leveraging Prior Effort (Doing less things)

The slowest way to build something is doing it from raw materials, uniquely for the current requirement. An analogy here would be building a stone structure where stones are individually prepared.

Figure 10 - Part of an old stone built foundry in Moratac Park, North Carolina

Think about the effort involved in construction:

- Finding and transporting stone
- Cutting stone to size and shape by hammer and chisel
- Fitting each unique stone carefully with its neighbours
- Cutting lumber, planing it into planks, making door frames and doors
- Obtaining brass for hinges, cutting, beating and filing it into shape to form hinges, handles etc.

This is certainly not a rapid process. It requires high skills and a great deal of effort. Changing or adapting the resulting structure will not be easy or quick.

In IT terms, this is analogous to developing a system from source code only, without using any pre-existing libraries or infrastructure software. Lets look at some options to achieve greater agility.

Use Component Libraries and Frameworks

If we have access to pre-made components, we can focus more on construction and less on the basic component making. Consider the figure below. Here we have bricks, cement, tiles, timber trusses, door frames etc. delivered to site. The primary effort here is in combining these into unique combinations to meet the client's requirements. Effort is much lower and time to build much less. We can probably also get by with lower levels of skill.

Figure 11 - Rapid Construction with Pre-Built Components
Source: https://i.pinimg.com

In the IT environment today we have very large libraries of components designed to work together and to provide for the building of nearly all common solutions and capabilities.

Frameworks are large component (also class and resource) libraries which provide integrated and compatible support as well as architectural guidance on how to layer and combine the elements successfully. Examples include Microsoft .Net, Java 2 Enterprise Edition (J2EE) and Apple's Core and Application Services. Each of these provides developers with a plethora of capabilities which can be accessed using common application programming paradigms and facilities. They insulate developers from a great many technical details.

Taking pre-fabrication to the next level, we can make use of complete pre-fab buildings, delivered to site and just connected to the infrastructure (e.g. water, electricity and sewage).

In IT terms, this equates to the use of packages for at least part of the solution. The idea is that we use commodity designs and constructions for generic requirements, and only adapt or develop bespoke elements where no standard ones are available that will do the job.

Packages can reduce time to market and investment required dramatically, if they suit requirements well and are not heavily modified. The more we modify them, the more we lose the advantages we seek and slip back into development, sometimes with requirements for rare and expensive skills in esoteric tools.

Figure 12 - Prefabricated Structure delivered to site

Use Reference Models and Patterns

Components reduce the effort to build the solution. Where we have to design the solution or elements of it, we can save time by using reference models and patterns. Patterns are at a design or architecture level, rather than the physical component level. They capture knowledge of proven approaches that work and that can be easily adapted to our unique needs. An example would be the pattern used for a clinic, where there would be a reception area, assessment area, consultation rooms, treatment area, dispensing facility, test area and ablution facilities for staff and patients. The pattern could be scaled up or down depending upon the budget and the number of clients it is expected to serve.

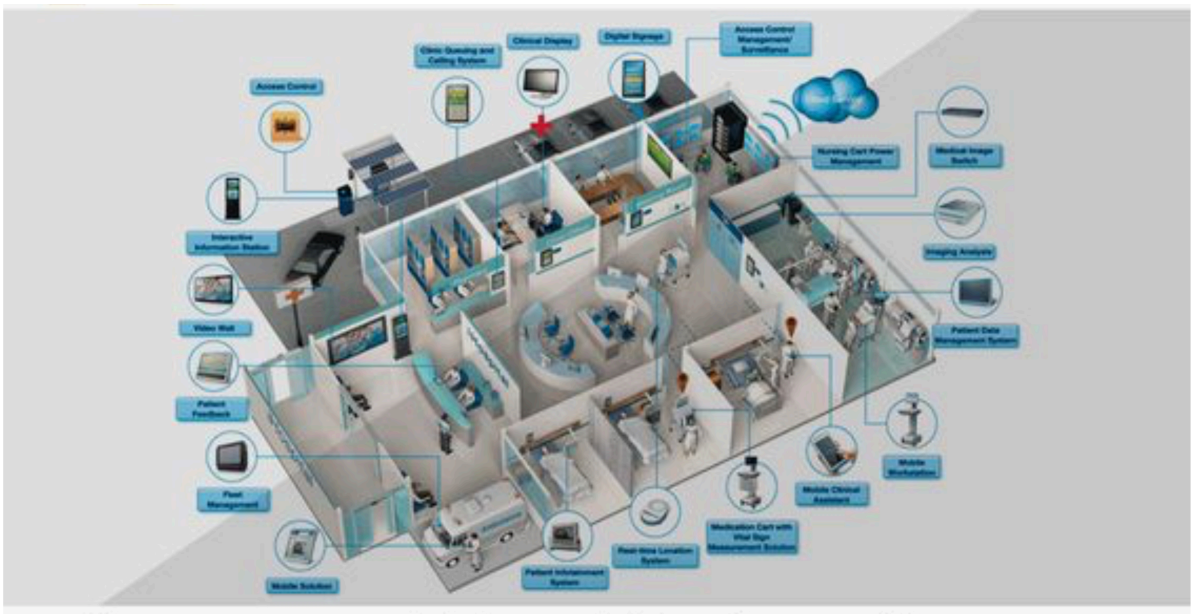

Figure 13 - A hospital architecture blueprint, or pattern
Source: www.hhbc.in

An IT equivalent would be the Model View Controller (MVC) pattern in graphical user interface systems, which provides architectural guidance in how to split responsibilities between layers of the software. This can be adapted for desktop, web or mobile use.

Where patterns address particular domain knowledge, they are typically referred to as reference models. Examples in industry include the BAIN reference models for services in Banking, the ACCORD models in Insurance, the ARTS models in Retail and the FrameWorx models for process, data and applications in Telecommunications.

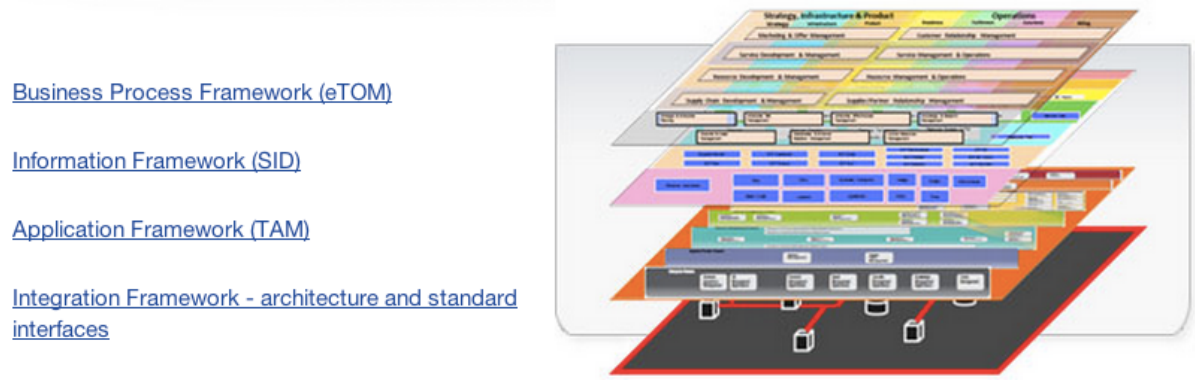

Figure 14 - Frameworx from the Telemanagement Forum provides generic models for Telcos

These can embody hundreds of years of analysis effort and can save a great deal of work during analysis, architecture and design. They also typically facilitate interoperability of our solutions with business partners and packages obtained from industry. Their use can reduce risk and time to delivery, as well as cost. These mainly reduce time in analysis and design, but can also shorten build, test and integration time by using proven solutions and models.

Generate Detail

Development normally includes the activities of requirements elicitation, design, building and testing. Requirements and design are best done as models to ensure their completeness, rigour and accurate communication between parties.

The build and test activities are often the most labour intensive. Software generation approaches eliminate a lot of this effort by generating the code from models capturing the requirements and design. Traditionally, this has been touted as a major boon to productivity, but has seldom delivered in practice. Approaches in this space include Model Driven Design (MDD) from the Object Management Group (OMG). Problems in achieving higher productivity were found to be mostly due to the models being solution / technology oriented and generic rather than requirement / domain oriented and specific.

An analogy here would be modeling a house in terms of foundations, brick walls, door and window frames, rafters and roofing sheets. These are generic components at a fairly low level. Specifying the model at this level does not save a lot of effort over building it.

However, if we could specify the house at the level of a conceptual architecture (style and floor plan / rooms only) and generate the finished house from that automatically (e.g. by 3D Printing) then we could see major gains in productivity and rapid development.

Figure 15 - 3D Printed House
Source: https://all3dp.com

The systems equivalent of this is known as Domain Specific Modeling (DSM). Requirements are modeled using domain concepts, such as Customer, Product, Order, Branch and associated processes and events. Code generation produces working software respecting issues such as architecture, standards compliance, security and other contextual requirements and targeting the technology platform of choice. There are major savings in time and cost, since the effort to create the architecture and generators is expended only once and the generated code is error free since it follows previously tested patterns.

Domain Specific Modeling and Generation

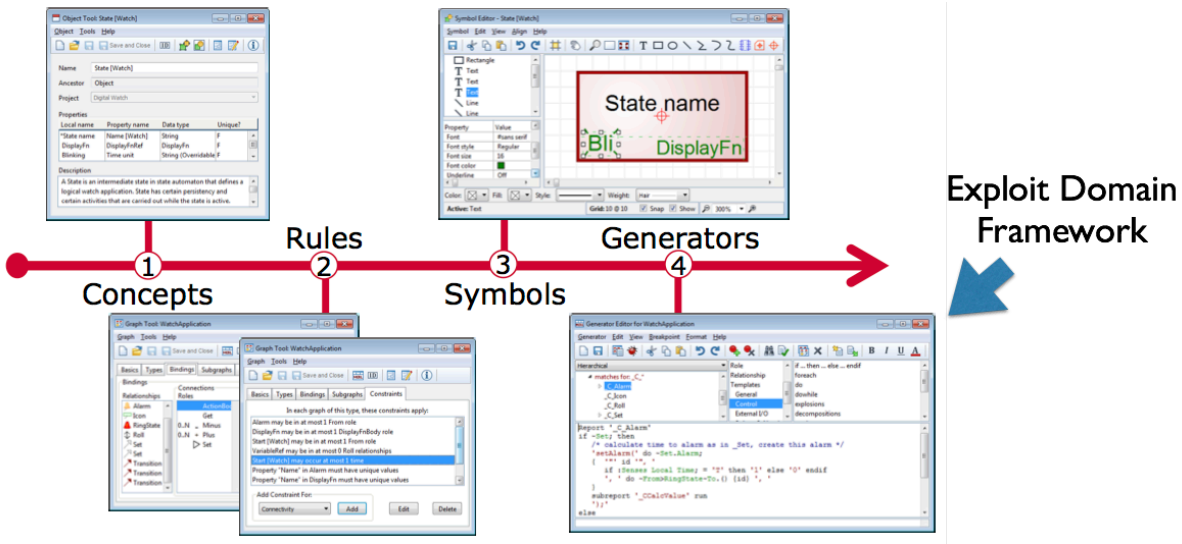

Figure 16 - Domain Specific Modeling Activities
Source: MetaCase

In industrial projects, DSM has proven highly productive. It is used extensively in the automotive industry, for example, to rapidly generate new software for new generations of vehicles. A new model could be launched every two years or so requiring millions of lines of custom code to manage fuel, emissions, power delivery, transmission, traction, braking, entertainment, comfort systems, lighting, navigation and a myriad other things reliably. Coding this in the conventional way in the required time and with the requisite reliability and quality is simply not feasible. Other industries which rely heavily upon these techniques include telecommunications (e.g. development of switches), cellular (development of handset software) and other electronics manufacture, where devices are increasingly driven by software. DSM is not restricted to these fields, but has also been used effectively in banking/finance, insurance/assurance, agriculture, plant automation and other applications.

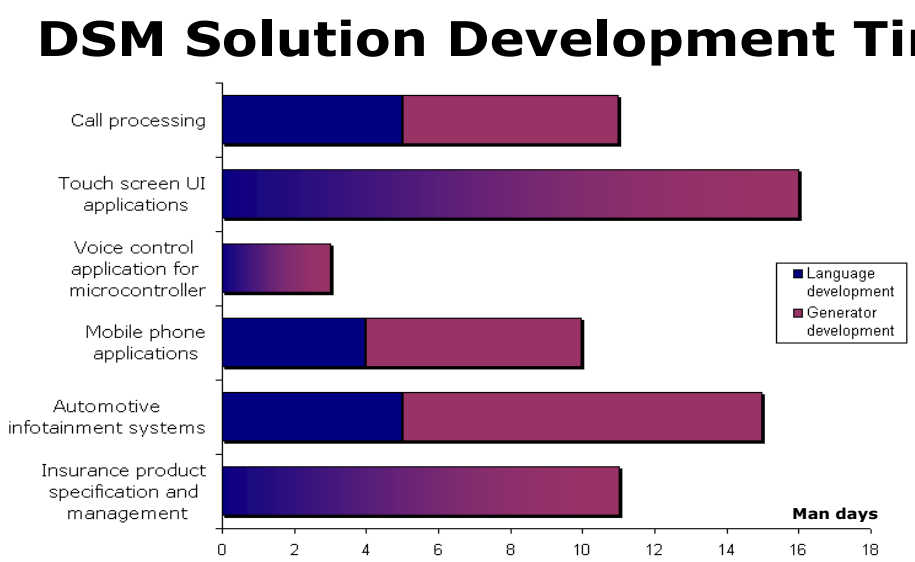

Figure 17 - DSM language and generator development does incur an overhead, but it can be small. Source: MetaCase

There is an overhead in creation of the domain specific language and associated generators for the first project(s), but the savings downstream pay back handsomely.

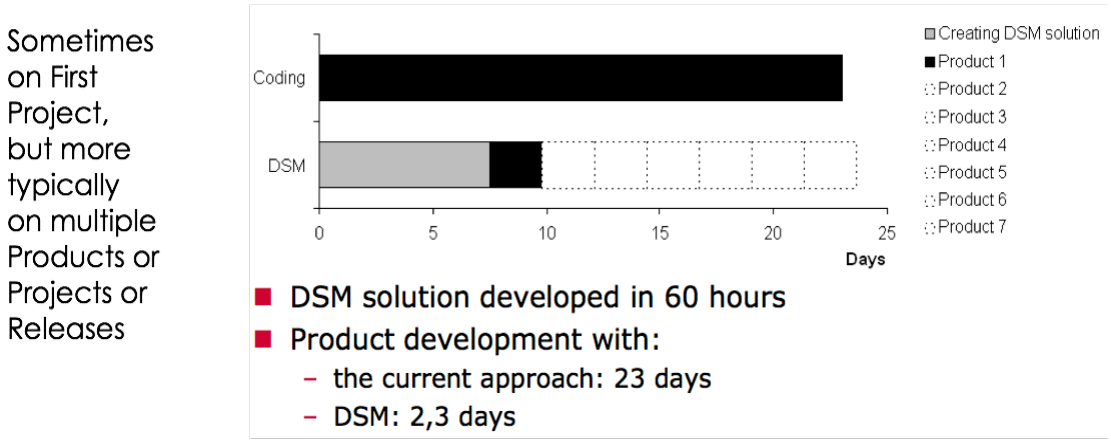

Figure 18 - DSM development may be recouped, even in an initial project Source: MetaCase

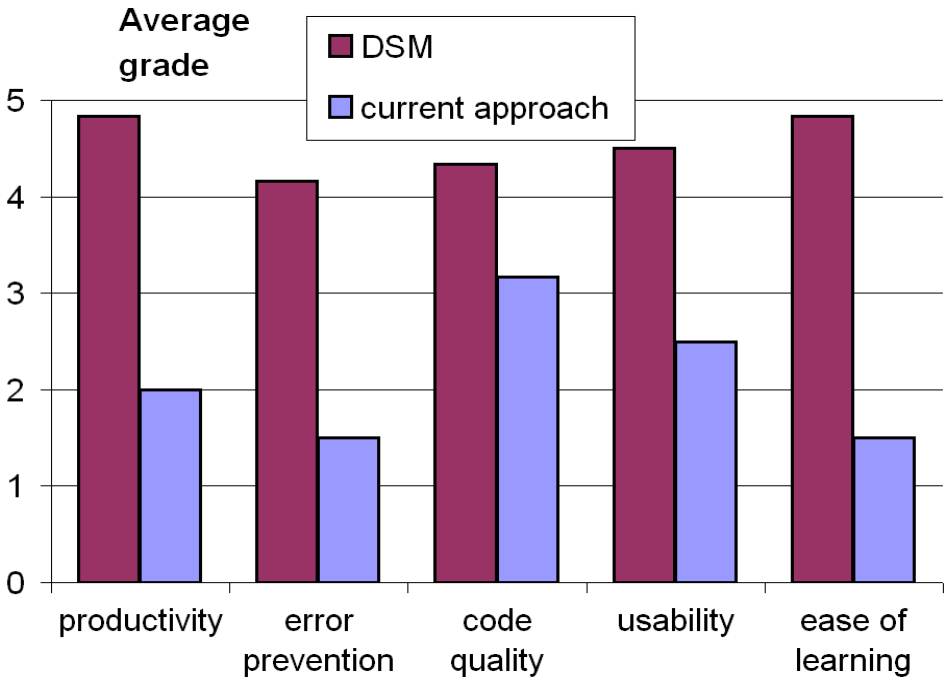

Figure 19 - DSM Projects vs Conventional Development
Source: MetaCase

Overall productivity, quality and match to domain requirements can be improved substantially, as well as time to market.

Reuse and Ensure Adaptability

Another way to reduce effort, and thus time, to delivery, is to reuse what we have done previously. This can occur in the form of components, frameworks, patterns and services as detailed previously. It can also occur in the form of reusing code, which we have ensured is well structured enough to be intelligible, well documented enough to change safely and well stored in a safe place so that it is easily findable.

A further requirement for reuse is that the code, components, frameworks etc. are of good *quality.* We do not want to reuse and proliferate problems and bugs! Achieving reuse is not easy and requires premeditation, discipline, skills and tool support. Some areas we need to pay attention to include:

- Architecture, so that components are coherent and focussed on achieving one purpose well
- Design, so that elements are well conceived, have clean interfaces and can inter-operate
- Standards, so that things are compatible and compliant
- Service orientation and loose coupling, so that it is relatively easy to swap components in or out, or to reuse things in different contexts
- Reliability, to ensure that we can safely use things without experiencing problems
- Performance, so that elements can be used in different scale applications
- Security, to ensure that we do not expose the organization to undue risk

- Documentation and knowledge management, so that components are easily discoverable and potential users can assess their suitability quickly and easily

Do Less Things Ourselves

Another way to accelerate delivery is to do less things *ourselves*. We have previously discussed saving effort by using components, libraries, frameworks and reference models. We can also achieve it by getting someone else to do some of the work. This may not save effort or cost, but it can alleviate resource constraints and reduce calendar time to delivery. A pre-requisite for this is that we have a good idea of requirements and architecture at a high level, so that we can apportion the work and responsibility.

A further requirement is that we have a productive and trusted outsource partner who will fit into our work approach and deliver products and components compatible with our environment. Unfortunately, finding a suitable partner and building the necessary collaborative work methods may take a lot of effort and time. It a bit like: "God grant me patience, but hurry!".

Make more Flexible Things

A final way in which we can become more agile, is different to those proposed above. Essentially the previous approaches concentrated mostly on being able to deliver the end product more quickly. There is an orthogonal approach which instead shifts the focus to delivering a product which is itself more flexible, obviating the need for new delivery. This harks back to our title of "Agile is a Stable Requirement".

A physical example of this would be the situation where we build a conference centre. We know up front that the facility must be able to adapt to accommodate the needs of a wide variety of events, ranging from trade shows, to music concerts, agricultural shows, sports events, business conferences and industry courses and tutorials. We need to be able to accommodate the events in short order and in any sequence of variety. This requires that the facility we design and build must, itself, be able to adapt very quickly. This can be accommodated by having large covered spaces which are reconfigurable via movable walls into different spaces, while catering in more permanent ways for the needs which are common across all functions, such as catering, parking, ablutions etc.

A great example of this is the Cape Town International Conference Centre (CTICC). It can be configured in less than 24 hours to cater for all the above mentioned types of events. This is more rapid than any construction method could possibly hope to achieve. The reconfiguration is achievable because the facility was *designed* with this level of adaptability in mind. The focus is on an adaptable resulting product, rather than on the build process.

Figure 20 - Cape Town International Convention Centre - A configurable facility
Source: CTICC

In IT terms, this can be achieved in two main ways:

- Runtime adaptable systems which are driven by user configurable meta models, business rules, output parameters and formulae. They can be adapted by users who are knowledgable about the business domain without needing to know the underlying technology
- Model Driven Development, especially DSM as discussed above. Here the system specification is maintained as a set of integrated models. When changes are required, the necessary changes are made at the business domain model level and new code is generated, potentially directly to production

The former style allows great flexibility and can cater for many changes to business requirements without needing technical system changes. A limitation is that the system will often be less efficient than a less flexible system, so this approach may not be appropriate where very high volumes must be processed or response times are very critical.

The second style is almost as flexible, but does still require the generation and deployment steps, although these can be automated. It is able to address high demands for volumes and critical performance.

In effect, the above debunks the You Aren't Gonna Need It (YAGNI) principle advocated in Agile methods to reduce scope. We replace it with YAGNI, but meaning You ARE Going to Need It, the "it" being agility. The goals, however are not incompatible. Remember that Agile evolved to assist in the rapid delivery of business capabilities. YAGNI was applied to reduce scope, so that we could reduce effort to deliver more rapidly. So, if we now turn that around, are we increasing scope and effort? No, but there is a trick. If we simply added more function for things that we might need, then, yes, scope would

increase and that would be bad. BUT if we work at a higher level of abstraction and identify the *kinds* of things that we will need now and in the future, we can often *reduce* the scope of the system further, thus saving effort and time in the original development while *also* equipping the solution with the ability to adapt in production use *without more development.* So, this is a win-win. The caveat is that we may require higher skills in conceiving, designing and building the original solution.

Runtime Adaptable Systems

An example of such a system familiar to most of us is Microsoft Excel (or other spreadsheets), which provides a range of generic facilities that help users tailor the system to their needs via formulas, headings and other capabilities, such as chart definitions. The generic capabilities anticipate the fact that users will have a very broad range of requirements which differ in the details, but have underlying similarities (e.g. the need to work with numbers, manipulate them with formulas, organise them into tables and sort them in various ways. Users are able to use the application to meet their needs in short order without recourse to developers to make changes, or the delay and cost of software development.

Spreadsheets have consistently measured very high productivity levels when empirical data is analyzed for delivery of given requirements in specific language environments. Witness the following statistics from Capers Jones:

LANGUAGE	LEVEL	AVERAGE SOURCE STATEMENTS PER FUNCTION POINT
1st Generation default	1.00	320
2nd Generation default	3.00	107
3rd Generation default	4.00	80
4th Generation default	16.00	20
5th Generation default	70.00	5
ABAP/4	20.00	16
Access	8.50	38
ANSI BASIC	5.00	64
ANSI COBOL 74	3.00	107
ANSI COBOL 85	3.50	91
ANSI SQL	25.00	13
Assembly (Basic)	1.00	320
Assembly (Macro)	1.50	213
C	2.50	128
C++	6.00	53
CICS	7.00	46

COBOL	3.00	107
Common LISP	5.00	64
Crystal Reports	16.00	20
DELPHI	11.00	29
EXCEL 5	57.00	6
FORTRAN 77	3.00	107
Haskell	8.50	38
HTML 3.0	22.00	15
JAVA	6.00	53
Machine language	0.50	640
Object-Oriented default	11.00	29
Objective-C	12.00	27
PASCAL	3.50	91
PERL	15.00	21
PowerBuilder	20.00	16
Reuse default	60.00	5
RPG III	5.75	56
SMALLTALK	15.00	21
Spreadsheet default	50.00	6
SQL	25.00	13
Visual Basic 5	11.00	29

Figure 21 - Relative Productivity of Different Languages
Source: Software Productivity Research

Language Level is a relative productivity indicator for computer languages. A language rated as 12 would halve the programming (build) phase of a project relative to a language rated 6. If the language is also accessible to domain/business personnel rather than professional computer system developers, it may also reduce effort, time and cost in the analysis and design phases as well.

Another example of a runtime adaptable system is the EVA Netmodeler enterprise modeling and knowledge repository toolset from Inspired.org. This allows runtime definition of a meta model describing the concepts, properties of these and relationships between them, of relevance to a user, group, domain or enterprise. It then uses this information to modify user interfaces, reports, visualization tools and other aspects dynamically at run time.

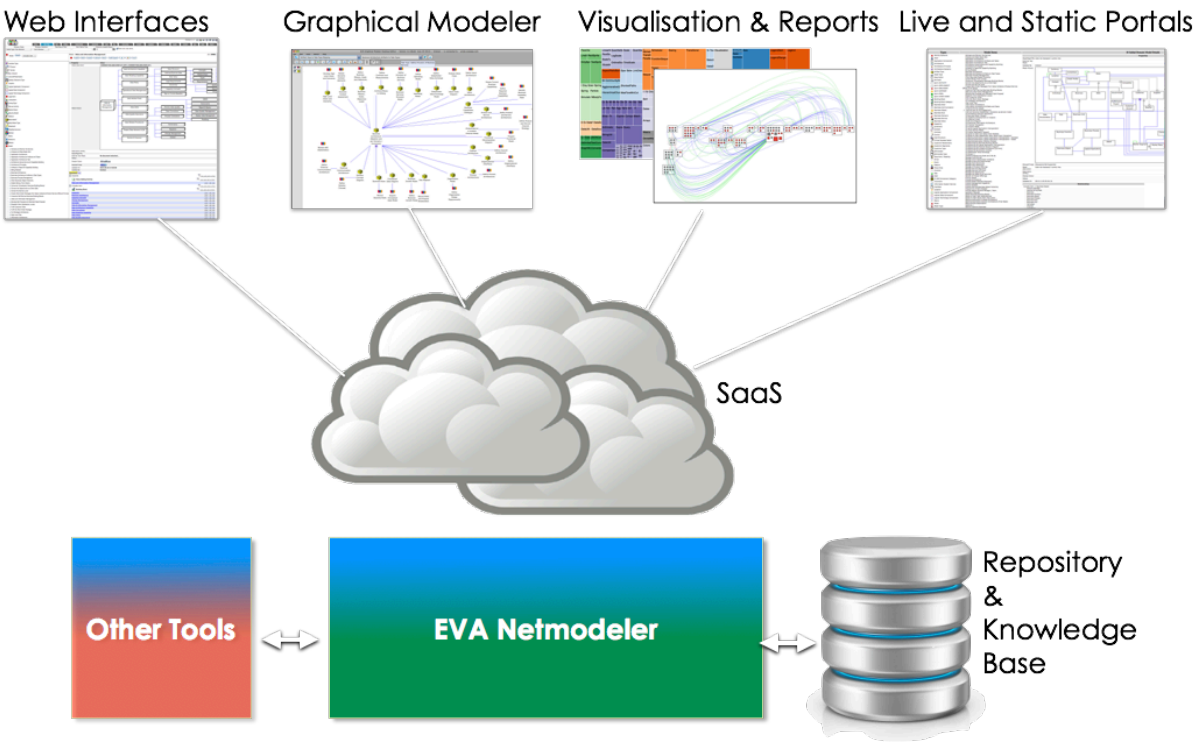

Figure 22 - EVA Netmodeler Architecture
Source: inspired.org

In one instance we met with a client in the banking industry and their industry consultants to determine requirements in support of a strategy, architecture and "re-baselining" project to be conducted across three continents. We drew these as a meta model on a white board during a long workshop day. In the evening we captured the meta model to configure the tool to address the problem and set up an instance on a server available via the Internet. The following day we demonstrated the tool support for the problems being tackled to the consultants and got their user credentials. We captured these into the tool that evening and the following day the tool was being used to capture information from live workshops on three continents.

The key to this rapid deployment was again the fact that the toolset is designed to cater for many and varied requirements by factoring out what is common to them and supporting that in a way that users can tailor to their needs. Technically, the tool has patterns for business logic and user interface in various styles, and injects the structures to which these should apply into the code generated from these, which is served to a browser via the network. More detail can be found in [McLeod, 2001].

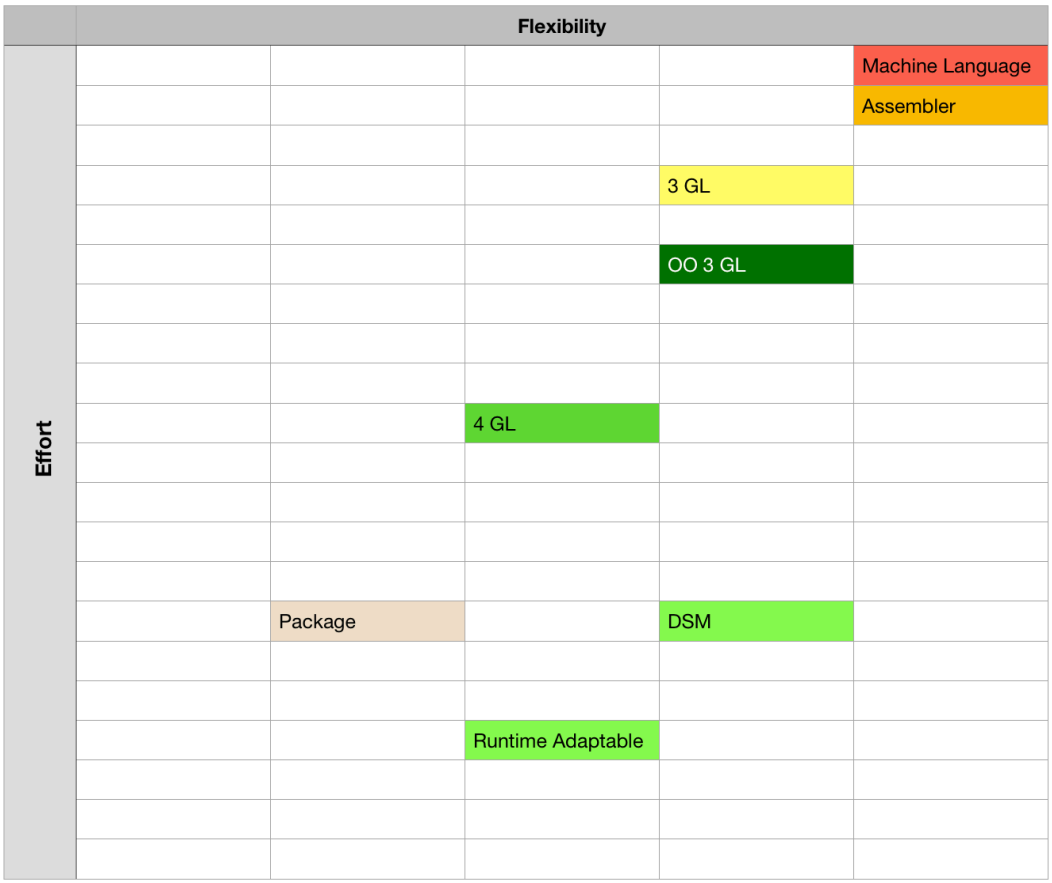

Figure 23 - Relative Flexibility and Productivity of Technologies

Agility and Legacy

Achieving agility going forward is tough enough. But most sites have a large collection of legacy applications which represent a substantial asset (or at least expenditure) and underpin operations. Often these are poorly documented and have "unravelled" to some extent through poor maintenance, so that their original architectures and design has become obscured. How do we leverage these and make such an environment more agile? We can very seldom afford to throw it away and start from scratch. Even if this was affordable economically, it would probably not be practical from a time perspective. We may also find that the domain knowledge is not available in the environment and user community, but is embedded in the old applications.

Conventional maintenance relies a lot on reading code. This is a daunting prospect when we consider that a legacy application (of which there might be scores or hundreds) can easily represent several million lines of code. Surveys show that maintenance programers spend as much as 80% of their time reading code. This activity hasn't changed much since the 1950's, with the exception that the volume of code has increased exponentially. This is a poor way to understand an existing system. It does not scale.

Fortunately, there is a new movement advocated by Tudor Girba and colleagues, known as Humane Assessment. The idea is to understand systems rapidly without being cruel to programmers. We already apply sophisticated tools to understand our business data

and to generate visualisations and insights from vast data collections (“big data”). Here the programmers are like the shoemakers children who are barefoot.

Can we not do better and use better tools and techniques to solve our own problems as well as those of users? Indeed we can. Techniques known as software visualization have been developed over a period of twenty years and the tools supporting these, including the Moose analysis platform and its add ons and extensions have become very sophisticated.

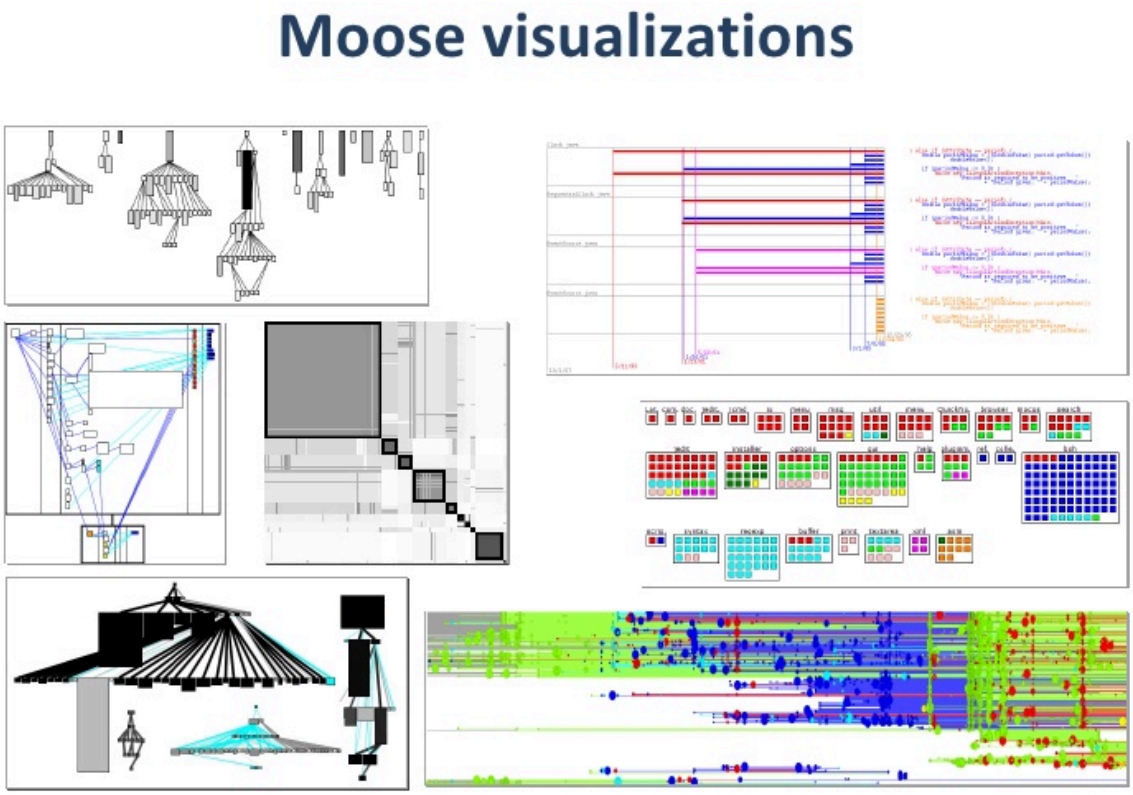

Figure 24 - Software visualization with Moose tools

Essentially, we can scan existing systems (code) to identify relevant data (e.g. Classes or definitions representing concepts and data structures; Functions, Modules and Methods representing functions or actions which operate on the data structures; and the relationships between these e.g. Which code depends upon which other parts and which code uses what data). The data can then be visualized using a variety of available techniques to identify structure, problems and opportunities for improvement. This can facilitate rapid maintenance with less effort and risk.

Harvest, ReModel, Forward Generate

Taking it further, we can supplement the analysis with some human expertise to derive domain knowledge, semantic information and other valuable assets from the existing software. These can be used to create domain frameworks and reference models for forward engineering.

We can then couple the recovered models with Domain Specific Modeling techniques to create a powerful Application Renewal method.

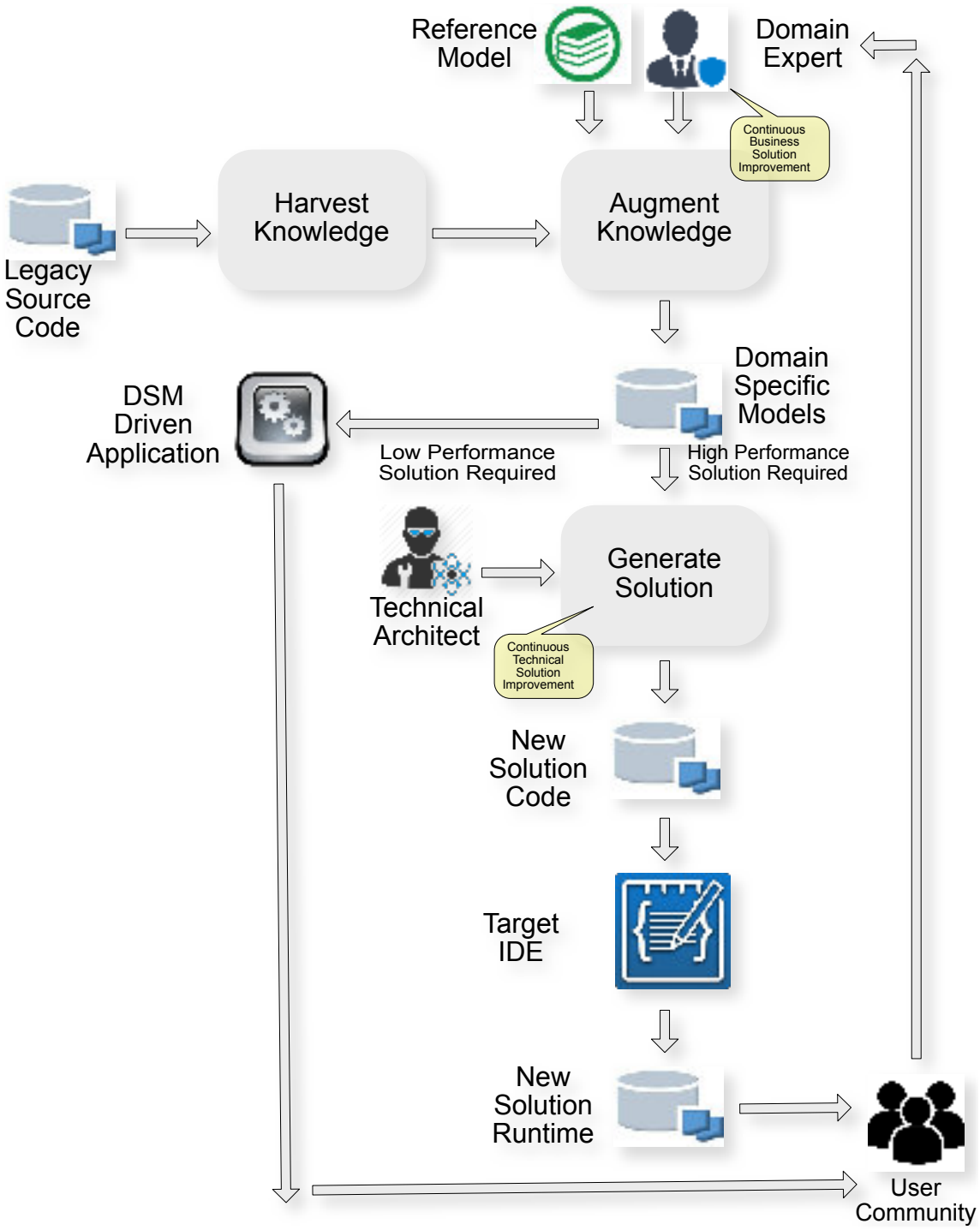

Figure 25 - Code Harvesting and Forward Generation with DSM

Inspired is currently developing and pioneering these techniques and is seeking commercial partners/customers to collaborate on these efforts. We are engaged with the authors of code harvesting and DSM techniques and tooling. If you are interested, please contact the writer.

The Lists

In this section we summarise some of the critical dimensions that promote or inhibit agility. In many cases these will have incremental effects when used together. In our culture, training, methods, management practices and projects we should try and do as many of the Promote items as possible, while avoiding as many of the Inhibit items as possible.

Promote Agility	Inhibit Agility
High Skills	Low Skills
User Involvement	User Dis-engagement
Frequent Short Meetings	Delayed Large Meetings
Empowered Staff	Centralised or Remote Decision Making
High Level of Abstraction	Low Level of Abstraction
Domain Specific Modeling	Generic Modeling
Integrated Methods	Non-Integrated Methods
Model Oriented	Document Oriented
Small Teams	Large Teams
Automation	Manual Work
High Quality	Low Quality
Good Infrastructure	Poor Infrastructure
Learning Culture	Do What You are Told Culture
Colocated Teams	Distributed Teams
Visual Work, Progress	Hidden Work, Progress
Collaboration. Mentoring	Punitive, Authoritarian
Informal with high trust	Formal with low trust
Developers have Domain Knowledge	Developers have only IT Knowledge
Users have IT Knowledge	Users have only Domain Knowledge
Continuity of Staff	Disrupted or Part Time Allocation of Staff
Reuse	Building Anew
Team of Teams	Command and Control
Runtime adaptable product	Design time adaptation
Virtual infrastructure	Traditional infrastructure

Figure 26 - Factors affecting agility

Promoting Organizational Agility

The forgoing discussion may seem daunting. After all, there are many dimensions and many associated disciplines, skills, cultures, techniques and tools to master or adopt. Where do we start?

It can be useful to take a leaf out of the book of continuous improvement, or the more formal discipline of Six Sigma, while avoiding the rigorous statistical side of the latter. Effectively we follow a simple cycle:

- Establish Intent - Decide what we want to achieve (e.g. Organizational Agility) and what that will require, via decomposition (e.g. Agile Friendly Culture; Ability to Rapidly Deliver Systems Capability in Production; Ability to Flex Business Model etc.)
- Identify the biggest bottlenecks to achievement. What are our current worst performing areas? Maybe it is a very autocratic culture, maybe it is an inability to accurately define requirements, maybe it is a quality problem in the delivered product. This is best done by looking at specific performance measures and

associated benchmarks, compared to our performance. E.g. If competitors can deliver a new service to the market in 3 months and we take 10. Find the top 3-5 areas which are under our control and amenable to change. Focus just on those for the next while

- Improve the underperforming areas by facilitating cultural change, improving methods, skills, techniques and tools, building trust and whatever other focussed means will address the issue
- Iterate - once change is achieved in an area and verified by new measurement, select the next most problematic area(s) for attention and keep doing it

In this way we can be assured that we are *always making progress*. It may seem slow at first, but it will be incremental as changes combine their improvements in a multiplicative way. We can accelerate change by the application of more effort, resources or money, provided that they are always focussed on doing things better, not just faster.

References and Further Reading

Papers

McLeod, Graham. “Pamela: A Proto-Pattern for Rapidly Delivered, Runtime Extensible Systems.” Evaluation of Modelling Methods in Systems Analysis and Design (EMMSAD) (2001) available here

Presentations

Software Visualization, Tudor Girba

Videos

Agile Visualisation in Mondrian

Software Environmentalism

Introduction to Domain Specific Modelling

20 Domain Specific Modelling Examples

Websites

http://www.inspired.org

http://www.moosetechnology.org

http://www.metacase.com