Understanding DSDM

Dynamic Systems Development Method or DSDM is a design methodology which is growing in popularity. David Norfolk explains what it is and what it can and can't do.

Dynamic Systems Development Method (DSDM) is a documented method for developing systems. Although most IT professionals view formal development methods with a degree of cynicism this one is a bit different. It isn't that traditional methods such as SSADM don't work, it's just that they take ages to deliver anything. True, even an inexperienced developer can use them to produce a reliable system - but why shackle experienced developers to the same rigid framework?

Tradition

In theory the traditional methods can be used flexibly and guarantee system maintainability and correctness (as system maintenance over the system's life usually costs more than development, a little extra overhead during development doesn't matter) but the idealised picture of development on which the traditional methods rely never seems to occur in practice.

In reality, everyone agrees that a project is feasible and then does a detailed requirements analysis, amidst much bickering and political wrangling. After several months the boss loses patience and bullies all concerned into signing off a requirements specification. Then the programmers go away (confident that if they deliver to this specification, no one can blame them for any problems) and design and code a system.

When the users get their new system many months later they go back on their requirements signoff and insist on all sorts of changes (often exposing fundamental design flaws in the system), partly because it is easier to recognise shortcomings in a real working system than in a paper document and partly because the requirements have changed anyway. Unplanned prototyping in a production environment (which is what is happening) is not very satisfactory.

The system works, eventually, but it is late, over-budget and, sometimes, no longer needed (quite often, most of the business benefit comes from a small part of the system as originally specified). The developers do avoid blame by quoting the specification they worked to, by the way, but the reputation of the whole IT department drops one more notch.

Better Way

There must be a better way, and there are four alternatives that are often tried. The most usual is powered anarchy. You buy what is optimistically called a rapid application development (RAD) tool and put together an impressive GUI-based system in a matter of days, often cobbling it together from bits of other systems. This is a popular move as the users can start work immediately, and if they don't like anything a particular feature can be reprogrammed over the lunch break. The problems appear later, when dozens of such systems don't talk to each other properly and maintaining any of them is a nightmare. At this stage, people start talking about throw-away code and the obvious fact that it is cheaper to redevelop than to maintain. Well, that may be true if you build systems that way but it seems a little wasteful.

A superficially more attractive approach is to give the toolset to a "superprogrammer", who goes off into a room somewhere and works 72 hours straight to deliver a maintainable, documented, gem of a system. This method works well (assuming that you have a reliable source of superprogrammers for when this one burns out) - until you discover that the issues which your new system addresses so brilliantly don't correspond precisely to any of the issues faced by your business.

In the end, it's simplest to go out and buy a pre-programmed package which your users can evaluate for themselves. However, this condemns them to working at the same level of efficiency as everyone else - if the package gives you an advantage, your competition will buy it too. No problem, if all it does is calculate tax payments, but a real problem if it is fundamental to your business competitiveness. This means that you will get insistent demands that you customise the package and (since the code was never designed with your particular needs in mind) you'll find that if you have to change

"DSDM is primarily based on continuous user involvement in an iterative (prototype-based) development process which is responsive to changing business requirements but still sufficiently defined for use with a formal quality management system if required."

more than about 5% of the code, on top of any customisation options built in, you'd have been better off developing from scratch.

There is, indeed, a swing back towards in-house development in order to give competitive advantage but the old problems haven't gone away. In order to gain competitive advantage you must deliver precisely what the user needs when the user needs it. The better RAD tool systems are recognising that the less controlled ones are prone to delivering the wrong system, or an unmaintainable system, too fast for anyone to do anything about it, and are extending their scope backwards towards rapid determination of real user requirements as a basis for rapid delivery of business systems.

This brings us back full-circle to a rather different view of the formal development method, as a framework into which the new tools can be plugged. DSDM is designed as a modern method which exploits the advanced tools available to the modern developer. It is supported by many of the major vendors (Texas Instruments, for example, produce a leaflet showing how their Composer by IEF tool satisfies the requirements of DSDM) and by a lot of major IT users such as American Express and British Airways.

Introducing DSDM

DSDM is primarily based on continuous user involvement in an iterative (prototype-based) development process which is responsive to changing business requirements but still sufficiently defined for use with a formal quality management system if required.

DSDM focuses on delivering business benefit instead of, as in traditional development methods, the avoidance of blame through signed-off specifications. DSDM projects should, therefore, avoid the twin problems of traditional projects: that the specification often reflects what the IT people, or even the business users, think is technically possible instead of what the business needs; and that the users have to wait to use the parts of the system they really need now until the whole system has been built, tested and signed off.

DSDM exploits this situation by implementing the famous (and, definitely, non-quantitative) 80:20 rule. It recognises that most of the benefit comes from 20% of the system requirements - so you can start generating competitive edge early if you concentrate on getting just that 20% delivered first and deliver the rest, incrementally, later. Of course, the system may not be 100% correct for much of the time, as it would be if you had implemented a formal specification, but 80% correct will be good enough as long as users are intimately involved with the development and in a position to ensure that the missing 20% has no serious business consequences.

We'll see later that DSDM does not promise to support applications where the functionality can't be assessed at the user interface - if the essence of the application is a set of complex calculations, for example.

Does this imply that DSDM can only deliver systems of limited quality? No, it actually guarantees higher quality systems, in terms of fitness for (business) purpose. The traditional fully-specified and signed off system was frequently 100% correct, in terms of adherence to specification, but often very low quality in terms of delivering business benefit when it was actually needed.

1. There is active user involvement (ideally both users and developers share a workplace), so that decisions can be made on the spot. This makes senior user management commitment into a critical success factor.
2. The team must be empowered to make decisions without waiting for higher-level approval. Again, senior management commitment is critical.
3. Business requirements take precedence (build the right product before building it right).
4. The team concentrates on product delivery rather than on carrying out prescribed activities.
5. Development is iterative, driven by user feedback.
6. All changes are reversible.
7. Management rewards product delivery rather than task completion.
8. Testing is carried out throughout the life cycle, not just before delivery (when it is generally too late to fix anything cheaply).
9. Estimates should be tight and specify frequent business deliverables.
10. Estimates should be based on business functionality (function points, perhaps), not on lines of code.
11. Risk assessment should focus on the business functions being delivered, not on the construction process.
12. Although DSDM doesn't require a signed specification, the high-level scope and objectives should be frozen or "base-lined" before you start.
13. There must be a flexible relationship between vendors and purchasers, because DSDM does not give you a formal specification as a basis for purchase orders.

Figure 1 - The Underlying Principles of DSDM.

Investment

So what do you get from an investment in DSDM? Make no mistake, it is an investment. The official handbook on the method (ISBN 1 899340 02 5) comes with the annual subscription of only $150 to the DSDM Consortium (contact it in the UK on +44 1233 661003) but you will also need proper training in the techniques (there is a 3-day certified course and at least some team members should know DSDM in

DSDM

detail before they start) and an unspecified investment in changing development practice.

Developers should take to DSDM with little trouble but the method does require significant, continuous, user support - so you will need users to commit valuable time to the development (which means that their managers may benefit from one-day DSDM-awareness courses). You will also need development support tools, but you probably want these anyway.

In return, DSDM gives you a framework for good practice. It doesn't mandate a development method, but it does define objectives throughout development, suggest people who should be involved at each stage and define suitable deliverables. It is important that DSDM isn't used as a mindless recipe for development but that the development team checks what it is actually going on against the framework.

DSDM also helps you manage development. Before the project starts it gives you some idea of the tasks which will have to be performed and it also highlights a project which isn't progressing properly (if, for example, a prototype is being reviewed by two groups with conflicting needs, each undoing the other's changes). It also gives you hints and tips about keeping the development on course and maintaining its business focus.

Finally, if you have a formal quality management process in place (such as ISO 9000), DSDM should let you demonstrate that the necessary controls are in place without jeopardising either flexibility of development or your accreditation.

The DSDM Model

The DSDM model is iterative, so at any stage you can return to earlier stages in response to, typically, business issues (the requirements may change, for example - not a problem here as it would be in a traditional method). It is also three dimensional, so at any stage you can drill down for assistance with particular processes. All of this makes it a little difficult to describe in something as linear as a written article (it fits better into an HTML page on the Internet - take a look at http://www.dsdm.org).

The first stage in a DSDM project is Feasibility. This is where the suitability of DSDM itself for the project is considered. If development is not going to lead to an interactive application where the functionality is directly evident to its users at the user interface, if it doesn't have a clearly defined user group and, especially, if it depends on complex inner calculations then it is unlikely that a method based on user-driven prototyping is going to be much use - the users of the prototype simply aren't going to be able to say if it is working properly.

Nevertheless, an experienced DSDM team working with strong tool support may use it in some unlikely projects. All that matters is that there is a feedback path on the progress of the project and the option to switch to a more rigid (specification-based) method if DSDM is not delivering.

There are well-defined principles behind DSDM development, which may imply a significant cultural impact on some organisations (for instance, if the decisions reached locally are currently subject to revision or overrule by some remote corporate IT standards body). There is also an underlying assumption that the development team is both highly skilled and stable, and effectively managed by a skilled and experienced project manager. You should think very carefully before using DSDM if the availability of skilled staff is an issue, or if any of the method's underlying principles (as listed in Figure 1) cause you concern.

However, once the feasibility of a DSDM approach has been accepted, it still remains to determine the high level scope and objectives for the project, in terms of the business benefits to be and the staff affected. Without this it is hard to see how the project can be controlled at all but we don't think that these initial stages should take too long - after all, if agreement can't be reached quickly at this level, the project is doomed anyway.

Development

Development proper consists of two stages of iterative prototyping (and one of deployment). The first concentrates on determining user requirements, with the prototype representing a working model of the traditional paper analysis models.

The second stage adds the user interface and control mechanism to the functional prototype that turn it into a functioning system which delivers business benefits. There is no separate testing stage.

Testing occurs throughout the life-cycle, on the principle that it is cheapest to remove errors as early as possible in the life-cycle and cheaper still not to introduce them in the first place (in the traditional life-cycle, the idea that errors can be sorted out later, during testing, is surprisingly seductive and you can find during testing that the entire system has been built on flawed foundations).

These prototyping stages are critical. User involvement is needed, of course, on the principle that the users may not be able to say what they want but they can certainly recognise the shortcomings in what they are given.

Tool support is vital during prototyping. Not only must user input be reflected in changes to the prototype before the users have time to lose interest, but it is useful if the user requirements can be automatically captured and subjected to a degree of quality assurance without the developers having to devote time to this (basically, if checks for internal consistency can be automated). The documentation needed for system maintenance should be generated, by the tool, as part of the process of building the system.

However, project management is critical too, in order to recognise problems in time to do something about them.

The obvious problem is a prototype which isn't converging on a finished state - more than 3 or so iterations probably indicates that the original scope, or the identified user group, has been badly chosen, and that the business study should be revisited. Other problems, such as a lack of specific skills in the development team or limited user commitment, should become obvious during prototyping and must be addressed.

Deployment

The final stage is, of course, implementation or deployment. It is expected that DSDM prototypes will evolve into parts of the delivered system - there is no room for throw-away code in a rapid development (although prototypes developed solely to test a particular technique may be discarded, after minimal investment, if the technique doesn't work).

What is deployed should deliver immediate business benefit, and changing requirements may mean that some parts of the system are never delivered. Deployment is controlled by "time-boxing", which means setting a deadline by which a business objective must be met rather than when a task must be completed (we are talking about a period of a couple of weeks, 6 at a maximum). Delivery should not be allowed to slip, even if the requirements do (that is, deliver fewer functions, working properly, in the time available, but don't deliver everything late or, worse still, on time but not working properly).

DSDM is quite simple in concept but making it work requires skills of the highest order - delivering better business function, faster, must cost something. This is where its third dimension of "good-practice techniques" comes in. The DSDM manual discusses the different sorts of prototypes and their uses:

- project management;
- team structures;
- user involvement;
- configuration management;
- estimating;
- metrics and so on.

Useful stuff, but you'll need proper training with someone who can answer questions out of practical experience. On the other hand, these essays could be useful adjuncts to in-house training and assist people evaluating the method.

Issues

There are some general issues with DSDM. For a start, vendor involvement in the Consortium may mean that it "tries to be all things to all tools". It does provide a useful guide to selecting a DSDM support environment but only at a pretty high level. Then, great reliance is placed on the skills and enthusiasm of both the development team members and the project manager and, if the wrong people get involved, unmaintainable systems could be delivered before anyone notices - this won't be a problem during the enthusiastic pioneer stages but could happen as DSDM become routine.

However, is there a practical alternative to the DSDM method for many projects? Rapid coding in a tool such as Visual Basic is unlikely to produce more maintainable systems and use of techniques such as object oriented programming can also produce poor-quality systems in inadequately skilled hands. DSDM is not a panacea, just a tool which can help you control rapid development but which can't replace intelligence and training.

Not Suitable

There certainly are projects for which DSDM isn't suitable. Part of the skill of a DSDM practitioner is to recognise them. On the other hand, the fact that DSDM doesn't fit your culture needn't stop you changing your culture enough to adopt it. You don't always need an architect and a quantity surveyor on a building project, but even a jobbing builder (if any good) should work to "accepted good practice" methods, tempered by the building regulations.

DSDM provides a framework for workable good practice in a much younger, and less regulated, industry. It encourages the use of RAD, with the consequent danger that too many corners are cut and unmaintainable systems result, but it gives you a lot of assistance with controlling the risks associated with RAD. Ultimately, always remember that a method can't force you to do something wrong: "I could see that we were heading for disaster but the method said that we were supposed to fail at this point" doesn't sound good coming from anyone qualified to be a project manager. At worst, the project manager may have to cope with problems not covered by a particular method, but that's what they're paid for.

The most obvious impact of DSDM on development and support staff is that they could find themselves working at a desk in a business area, alongside a real user, instead of in a programming shop.

They could even find themselves working for a project manager with no coding experience, seconded from the user community.

They could also expect better user relations on all projects, even those not using DSDM, a better reputation for IT in general, a more enjoyable job - and, most important of all, to be part of a more competitive business organisation.

DSDM is really a timely reminder that computer system development is a science or, at least, a craft and that there are rules of thumb and methods which you should expect anyone following it to be aware of.

Knocking up even a small application "by the seat of your pants" without user involvement is no longer professionally acceptable. The consequences to business competitiveness and future maintainability are too well known.

PCNA

The Author

David Norfolk is a writer and consultant with a background in providing IT support within a major banking organisation. He can be contacted as drhys@cix.compulink.co.uk.