

Re-factoring grid computing for usability

Bruce Beckles

University of Cambridge Computing Service, New Museums Site, Pembroke Street, Cambridge CB2 3QH, UK

Abstract

In this paper the author analyses the grid computing paradigm and attempts to show how the established principles of usability engineering, interaction design and other user-centred design methodologies can be applied to this paradigm to “re-factor” it so that it addresses usability concerns. Usability issues are illustrated with examples from the Globus Toolkit™ ([1]).

1. Introduction

Grid computing has been variously described as “the Next Internet” [2] and “the Holy Grail of computer science” [3], but successful uptake of grid computing has been remarkably slow ([4], [5]). Although the UK has invested significantly in grid computing and its related technologies and infrastructure ([6]), there is little evidence that this investment has actually delivered the infrastructure necessary for e-Science ([7], [8]), i.e. “the large scale science that will increasingly be carried out through distributed global collaborations enabled by the Internet” [9].

In fact, members of the UK e-Science community have repeatedly highlighted (see, for example, [10], [7], [11]) a number of serious issues regarding this grid computing infrastructure and, in particular, the underlying software – the so-called “grid middleware” – that has been used to develop it, the Globus Toolkit™ ([1]). One of the key concerns has been to do with usability. It would seem that neither the infrastructure itself, nor the software upon which it is built, is particularly usable for either application developers or end-users.

Amongst the human factors and HCI communities it has been well known for some time that if a computer system is to be usable then usability must be a central concern of its designers which must inform all phases of its design and implementation ([12], [13], [14]). This includes not only the system’s visual presentation and the steps by which tasks are carried out using it, but also (and this is often overlooked) its underlying components and their functionality ([15]). Moreover, good intentions are not enough – particular practices must be introduced into the project lifecycle for usability to result ([15]). Therefore, given that user concerns have not been of paramount importance in the development of either the grid computing infrastructure or the grid middleware, and attempts to address this problem have been continually frustrated ([16]), it is hardly surprising that its usability is so poor.

It is my contention that, at least in any arena where well-resourced dedicated specialists in the appropriate areas of computer programming are not available (such as most academic research projects that wish to make use of grid computing), there is *no* trade-off between usability and functionality. If a product or system is not usable by the non-specialists, then its functionality is irrelevant: it will never be used because its usability is so poor that it is too expensive (in terms of time, programmer effort and other resources) to attempt to use it.

In this paper I describe how usability concerns in the grid middleware can be addressed by applying established principles of usability engineering and related methodologies from the HCI and human factors disciplines.

2. An apology: the impossibility of re-factoring

As mentioned above, it is well known that usable systems can only be produced if usability is a central concern of the designers that informs all phases of the systems’ design and implementation ([12], [13], [14]). The corollary of this is that usability is rarely a “feature” that can be “added” to a product; in particular, it is not something that can be “added-on” after the product has been completed – such efforts often do not address the core issues, only addressing surface features such as the visual presentation, and are sometimes described as “painting the corpse” [14]. What are the implications of this for “software re-factoring”?

I contend that in most cases it is impossible to “re-factor” software that was not designed with usability as one of its central concerns into usable software. In such circumstances, the only way to “make the software usable” is to re-design it “from the ground up”, i.e. throw away the existing unusable software and design something new with equivalent functionality, but which has usability as one of its central design concerns.

Unfortunately, given the large investment of money ([6]) and other valuable resources (time, manpower) in developing this unusable

infrastructure for e-Science, and the even less usable software underlying it, it is unlikely that we will ever be in a position to do this. Therefore, the only viable alternatives open to us are to either:

- (a) develop new grid middleware that is more usable whilst remaining compatible with the existing middleware (as advocated in [7]), or
- (b) to attempt the impossible: re-factor the existing middleware for usability.

The two alternatives are not as unrelated as one might suppose. If we examine the current grid middleware with a view to making it more usable, it is highly likely that our results will have immediate applicability to any attempt to design new, more usable middleware that has equivalent functionality. So even though re-factoring the existing unusable middleware can never entirely satisfactorily address usability concerns, it is still a useful exercise.

3. Why focus on the middleware?

It may be argued that whether or not the middleware is itself unusable is irrelevant to the usability of the infrastructure which is built upon it, and also to the concerns of the users who make use of that infrastructure and the applications deployed upon the infrastructure. The following example should make clear that this is not the case.

Most grid middleware has chosen to use a public key infrastructure (PKI) of some description as the cornerstone (or, in some cases, as the entirety) of its security architecture. Unfortunately, there are serious usability problems with PKIs in general (e.g. [17], [18], [19]), and the PKI implementations currently used in such grid middleware are no exception (see [11], [20] for details). Applications or infrastructure built on this grid middleware are forced to use these PKI implementations for their security layer, and the usability problems inherent in these implementations are thus transferred to the end-user.

It is, of course, possible for either the infrastructure or individual applications to mitigate these problems by re-implementing the security layer of the grid middleware themselves, or by implementing another, more usable, security layer that acts as an interface between the grid middleware and themselves. (For instance, on-line digital certificate repositories are often cited as partially improving usability in this context ([21], [20]) and can be thought of as a security layer between the middleware and the end-user.) However, the whole point of using a

middleware layer is that it frees the developer from such concerns: it is supposed to be the responsibility of the middleware to provide appropriate security services.

I contend, therefore, that the only generally feasible method for developing usable infrastructure, and usable applications that use that infrastructure, is to begin with a usable middleware layer. Of course, this alone does not guarantee that the resulting infrastructure or applications will be usable, but – at least in the general case – it is a prerequisite for usability. (There will, of course, be specific cases in which it is possible, by dint of much effort involving the re-implementation of middleware services or the development of additional interfaces, to produce usable infrastructure or applications that are based on unusable middleware.)

4. Usability principles

Considerable work has been done in the field of usability engineering and a number of principles have been established as being of tremendous value in software development (for examples, see [12], [22]). There have also been significant advances in the related field of requirements engineering, most notably in the development of user-centred methodologies such as contextual design [13], interaction design [23], and Sommerville's viewpoints and concerns method [24]. (That these two fields are closely related should be obvious: it is hard to develop usable software if the designer does not know the purposes to which the user will actually put that software in the contexts in which it will be deployed, i.e. the user's requirements in relation to the software.)

Since our concern here is with usability, we are not necessarily concerned with other aspects of software development. For instance, provided the software achieves its user's purpose in a manner satisfactory to the user, it is of no concern whether it does so in the most efficient manner. It will be helpful at this point to make explicit the areas with which we are concerned in the usability context. Broadly speaking, these are:

- Engagement with the intended user community,
- APIs and other programmatic interfaces,
- User interfaces,
- Security,
- Documentation, and
- Deployment issues

These areas shall now be examined in turn, and, in each case, usability principles and

practices relevant in the context of grid middleware will be discussed.

4.1. User engagement

It is worth remembering that in the context of an API or middleware layer, the programmers / developers who use the API or middleware layer to develop applications are its “users” and their concerns should be the “user concerns” that inform the designers of the API or middleware layer. In other words: “programmers are people, too” and exactly the same kind of human factors approach that has proven so necessary for producing usable software for end-users should also be employed to produce usable APIs or middleware layers ([25]).

As has been already discussed, continuous engagement with the intended user community, from the earliest design stages through to the final implementation phase is a prerequisite for usability ([12], [13], [14], [22], [23]). Indeed, many of the suggestions and principles that are discussed in the following sections can only be meaningfully applied if there is, at a minimum, early and repeated engagement with the intended user community. This immediately leads us to two core issues that must be addressed before undertaking any middleware design or development:

1. Who is the intended user community?:

Once the intended user community has been identified, it will be possible to interact with them to gain a better understanding of their requirements. However, if the user community is not well-defined (e.g. “anyone who wants to use it”) or is too diverse (e.g. “scientists”) then this is unlikely to be of much use. It will then be necessary to partition the user community into manageable sub-groups that are characterised by having the same (or very similar) sets of requirements ([23]).

There is an important subtlety here: generally speaking, it is not possible to design a usable truly general solution for any sufficiently complex problem space. This is because in such complex problem spaces, the users are likely to form disjoint groups with non-overlapping requirements. Whether or not this is the case in the problem space represented by the grid computing paradigm is unclear, since there has been such resistance from those involved with middleware development and infrastructure provision to determining their users’ requirements ([16]). But given the complex nature of this problem space, it seems highly likely that it is the case.

Thus middleware such as the Globus Toolkit is unlikely ever to be usable as long as it tries to be “all things to all people”. It is more likely to end up being “nothing (of use) to anyone”.

2. What are the user community’s requirements?:

Having determined the user community, it is then essential to determine their requirements. This may not be straightforward, especially where the problem space is sufficiently new to the users that they have little experience of it. Unfortunately, there is no shortcut – as Ken Arnold, the original lead architect of JavaSpaces, points out: “we could have done many things with JavaSpaces, but our capabilities were of no value. It is the user’s desires that were of value.” [26]

Fortunately, any of the user-centred design methodologies (such as [13], [23], [24]) are well suited to such complex problem spaces. Having clearly established the user requirements for the intended user community, those requirements need to be examined for consistency. Should they prove not to be consistent, it may be possible to partition the user community into distinct groups each of which has a consistent set of requirements – in this case, as many different products as groups will need to be developed. If this is not possible it is essential to return to the user community and work with them to find some compromise, acceptable to them, which gives an agreed set of consistent requirements. Once a set of consistent requirements have been found, only then should one proceed to the design phase.

4.2. APIs

For many application developers, the middleware API is to them what their application’s user interface is to their end-users: it is the way in which they interact with the middleware, and their experience of this API will characterise that middleware for them. Thus designing a usable API is not wholly dissimilar to designing a usable interface, and many of the usability principles useful in that context can be straightforwardly translated. The following are a mixture of such “translated” design principles (several taken from [25]), as well as other principles or guidelines that are likely to be helpful in this context:

- *Suitability of functionality:*

The functionality that the API provides must be *suitable* for the target domain in which the application developer is working, i.e. its functionality must be *user-driven*. In practice, this means understanding not only the developer’s requirements, but also something of

the end-users for whom they are undertaking that development. (Active engagement with the developer and their target user community is thus essential.)

For instance, in some application domains (e.g. distributed prime searches), the amount of data that needs to be transferred between machines is quite small, and sophisticated data transfer mechanisms are wholly unnecessary. However, if the only data transfer mechanisms available are designed for optimising large data transfers (such as the GridFTP component of the Globus Toolkit), these may well be inappropriate (and the overhead involved in using such sophisticated mechanisms may well be significant).

- *Standards compliance:*

Where there are appropriate standards to which the middleware should conform, it should comply with those standards as fully as possible, *provided that doing so will not make it less usable*. There are a number of reasons for standards compliance. Developers are likely to have been exposed to the standards elsewhere and if the middleware is fully conformant to those standards it will make it easier for them to learn and use it. In addition there may be third party tools that they can use to aid them in development if the middleware is standards compliant. Also, if their application needs to interface with something that uses a different API or middleware layer that is also standards compliant then their task is likely to be significantly easier.

However, the middleware should never adopt a standard and then implement that standard in such a manner that its implementation is non-compliant, nor should it “extend” that standard in a way that makes it non-compliant – this practice is known as “embrace and extend”; see [27] for an example of the problems inherent in this practice. This is probably the worst of all worlds, and is exactly what happened in Release 3.0 of the Globus Toolkit ([7]). Perhaps as a consequence, take-up of Globus Toolkit 3 has been almost negligible ([5]), with users and developers migrating to the so-called pre-WS components (i.e. the updated versions of Globus Toolkit 2) and ignoring the Toolkit’s other components.

The rule is simple: either be *fully* standards compliant, or else make *no* attempt to comply with the relevant standards, in which case make sure that you are *not* sufficiently close to those standards that you mislead the developer.

If there is no standard, consider creating a standard, in a public, transparent fashion (as standards created in secrecy are likely not to be

“standards” at all, but rather simply a particular implementation set in stone, often chosen for no reason other than “that’s what we do already”). However, if you do this make sure that the standard is mature before you start producing middleware based on it. An immature standard is likely to change reasonably rapidly, possibly in ways that require the application developer to completely re-write their code. They will not thank you for this.

- *Progressive disclosure:*

Progressive disclosure is the idea that only the most commonly used functionality should be directly exposed to the user, whilst more advanced functionality is hidden until it is required. For an API, this means that only the most commonly used methods should be directly exposed to the developer, with a small number of additional methods that give access to the more advanced functionality of the API (the best way to do this will depend on the language bindings available for the API). I leave it as an exercise to the reader to examine the API calls for the Globus Toolkit and work out which ones are actually likely to be used by the average grid application developer. (Of course, a systematic audit of the function calls used by developers using the Toolkit would be an extremely useful exercise.)

- *Simplicity:*

Most would argue that simplicity should be a design goal of any computer system, and this is at least as true for APIs. The API should be as simple as possible and only as complex as is necessary to fulfil the developer’s requirements. There are a number of strategies you can pursue that will help achieve this goal.

Firstly, make those actions that the developer needs to do most frequently as simple and efficient as possible (ideally a single API call with as few parameters as possible) even at the expense of other functionality that is less used. Secondly, identify the most common sequences of function calls and make them *atomic* if at all possible, i.e. replace them with a single function call. (Once again, these strategies are only available to you if you have a very clear understanding of your developers’ requirements.)

Obviously, it will not always be possible to make your API as simple as your developer would like, and it may be necessary to make trade-offs between simplicity and other concerns. Simplicity is not an absolute goal, and it is possible to make something too simple to be useful.

- *Appropriateness of interfaces:*

Consider carefully how you expose your API to developers. What is the language that your developers use, or use most often? That's probably the language that you want them to use when they interact with your API, so produce bindings for that language. Of course, this is only possible if you have clearly identified the intended user community for your API. Are there standards (either formal standards or community standards) for APIs, etc. in their chosen language? If so, it is in both your and their interest to conform to them. Does this development community have a particular method or paradigm for interfacing with external libraries, etc.? If so, you should try to support it if possible.

- *Error prevention*

Application developers are human. Therefore, like all humans, they make mistakes. So you should try to design your API so that it minimises or mitigates human error wherever possible. This does not mean "second guessing" the developer, but simply ensuring that when it does not make sense to do something, your API either disallows it, or makes it very hard to do. (Of course, if you are adhering to some standard there may be cases where you cannot do this.) One way of doing this is to use "forcing functions", which force the developer to do something that makes it unlikely or impossible to make a particular type of mistake.

For instance, if authentication is required before a secure communication channel can be created, it should be impossible for the developer to try to create such a channel without first authenticating. It should not, for instance, be possible to correctly issue a function call with an un-initialised authentication token, only to have the call then fail at run-time. If the initialisation call for the authentication token returns a different type of object than the token itself, and this other type of object is required by the call that creates the secure channel, then the developer will always be forced into initialising the authorisation token.

- *Literate programming*

The central idea behind literate programming is simple: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." [28]. How does this apply to making APIs usable? If our API is written in a "literate" fashion, then it will be much easier for the application developer to understand what it is trying to do,

and consequently how they are expected to interact with it.

For instance, function names need to be suggestive of their purpose, whilst not being misleading. Proper documentation (see Section 4.5) is essential – the API should not be specified as simply a bare list of function calls, but instead each function call should have a brief rubric describing its purpose (in human terms), semantics, and relationship to the rest of the API.

- *Rapid application development / prototyping:*

Most application developers will, at some point, either need to test an idea to see if it is valid by building a simple prototype ("prototyping"), or find themselves being asked to deliver a functional application in a short period of time. This is particularly likely to be the case in the scientific community where, with the possible exception of computer scientists, the purpose of software development is to enable science of some description; it is *not* an end in itself.

Therefore it is highly likely that your API will need to support both prototyping and rapid application development. To enable this, the API should be easy to install and will need not to have a steep learning curve. For a more detailed discussion, see [7].

4.3. User interfaces

Designing an appropriate user interface is a key factor in producing a usable computer system. The user interface of the grid middleware is more than simply its API: it is concerned with all aspects of the user's interaction with the grid middleware. For some users that may simply be the API, but for many – at least with the current generation of grid middleware – it will also involve things such as its installation procedure(s), its documentation, its administrative utilities and any other utilities (command-line, etc.) that allow them to directly interact with it.

All this needs to be made as usable as possible (balanced against other user requirements), and this probably means it needs to be as integrated as possible. It certainly needs to feel "natural" to the user and to support them in their conceptual model of how it works and what their interaction with it should be. Reports from users of the current middleware have tended not to be favourable ([10], [7]).

The middleware should not impose – through its interface or its functionality – a conceptual model that is adversarial to the user's normal way of working or conception of how the middleware works. (It is possible that

there are users whose normal ways of working involve abandoning a system in disgust, never to use it again, but one hopes that they are comparatively rare. My personal experience of this behaviour has occurred most often when I have watched new users attempt to use or install Globus Toolkit 2 or 3.)

Instead of imposing a conceptual model or other framework, the middleware should support existing practice wherever possible, and where it is not possible for it to do so, it should provide guidance and/or clear migration paths. Again, all of this is most easily achieved using one of the user-centred design methodologies (such as [13], [23], [24]).

4.4. Security

It is often said that there is a trade-off between usability and security. This is completely untrue. It is accepted by both the HCI and security communities that usability is not in conflict with security, but rather a necessary attribute of any security mechanism that wishes to be deployed securely in the real world ([29], [30]). Further, either there are security requirements, or there aren't. If there were no requirement for a system to be secure, then it would be foolish to attempt to make it so. If there *is* a requirement for the system to be secure, then that means that if the security is compromised the system will be unusable (logically, because it no longer fulfils its requirements; more prosaically because it needs to be shut down to contain or handle the security incident). Thus if security is actually a requirement for a system, then it is a usability requirement as well, since, in that case, you can't use the system without it.

As described in Section 3, most grid middleware has adopted a security mechanism with which there are serious usability concerns. To make the existing middleware usable, that security mechanism needs to be either removed (where application developers do not require it) or replaced with a more usable mechanism (for those developers that do require security). For a new middleware layer the choice would be simple: either provide no security mechanisms, or provide usable ones. Software engineering methodologies such as AEGIS [31], which has been successfully applied to several grid computing projects (e.g. EGSO [32]), can help in cases where usable security mechanisms are required.

4.5. Documentation

There have been consistent criticisms of the documentation of much of the current

middleware, in particular that of the Globus Toolkit ([10], [7]), although this documentation seems to have improved in recent releases. Unfortunately, the improvements seem mainly to have been in its completeness, which, whilst important, is only one component of what makes usable documentation. Some of the other components are:

- Accuracy,
- An appropriate level for its intended audience,
- "Literate" documentation,
- Progressive disclosure, and
- Numerous useful examples, with
- Templates for most frequently used non-atomic tasks

The rationale for most of these is clear, but a few comments regarding "literate" documentation and progressive disclosure may be helpful. Progressive disclosure (see Section 4.2) in the context of documentation means that instead of presenting the user with long, exhaustive lists of functions, etc., the information is presented in a hierarchical manner, with the information that the user is most likely to want being easiest to access, whilst information less likely to be consulted is further down the hierarchy.

"Literate" documentation is an extension of the concept of literate programming ([28]) to documentation for software: the idea is to write documentation that explains to a human what it is our software wants the system to do in *human* terms, rather than in purely functional terms, e.g. "At this point UsableGrid attempts to connect with a remote site using the information you have given it earlier (<ref to earlier place>). It does this by opening a network connection to the remote site using standard system calls and then attempts to authenticate using your authentication credentials (<ref to explanation of authentication credentials>) and the UsableAuth authentication mechanism (<ref to UsableAuth>)." as opposed to "MysteriousGrid authenticates."

4.6. Deployment

It may not be immediately obvious why deployment issues should be an important part of the middleware's usability. The reason is simple: if the middleware is difficult to deploy it is unlikely to be used, and further, the more difficult it is, the harder it is for "casual" users to use it for prototyping or rapid application development. It is worth bearing in mind that many of the users of the middleware, such as application developers, will actually be the ones who install it on their systems (at least initially).

Many of the usability issues involved in middleware deployment are discussed in [7] (which focuses on the Globus Toolkit, but its conclusions have wider applicability), to which I refer the reader for discussions relating to the following guidelines:

- Re-use existing system libraries;
- Use existing packaging and installation mechanisms;
- Keep the middleware footprint as small as possible;
- Support incremental adoption and deployment; and
- Be able to run in user-space.

As a final exercise for the reader, I suggest that you attempt to install Globus Toolkit 4 on your own system and see how long it takes to install and configure in its simplest mode of operation. While you're at it, assess its footprint (disk space, etc) on your system.

5. So what would a usable grid middleware layer look like?

The question that may be left in the mind of the reader is: so just what would a usable grid middleware layer look like? It is actually quite difficult to say, because inextricably bound up with the notion of usability is the context of the intended user community and their requirements. Since we know so little about the requirements of even those communities that have attempted to engage with the grid computing paradigm, any description given here would be little more than informed speculation.

But a speculative summary might go something like this. It would probably be domain specific (but how large or small those domains would be is difficult to say), since I don't believe that we can have a truly general grid middleware layer that is of any practical use to anyone. However, it might be possible to have several distinct, usable, domain specific middleware layers over a more general (and less usable) middleware framework without harming the usability of those domain specific middleware layers too much.

Within the domain of such domain specific middleware the requirements of the application developers and their end-users would be well understood. The middleware would be "lightweight": a small system footprint, easy to install and suitable for rapid application development and prototyping. It would certainly look much more like WSRF::Lite ([33]) or Google's MapReduce implementation ([34]) than the Globus Toolkit.

The API would feature progressive disclosure; useful (possibly domain specific), complex (i.e. composed of more than one "simple" function), atomic functions; and bindings to the language(s) most appropriate for the application domain. It would be well documented (said documentation containing a wealth of practical examples). It might even be the case that the following pseudo-code translates into a mere 6 function calls in the API:

```
CONVERT DATA TO GRID FORMAT
CONNECT TO GRID
SEND CONVERTED DATA TO SERVICE
GET RESULT WHEN AVAILABLE
DISCONNECT
CONVERT RESULT TO SENSIBLE FORMAT
```

Is that really so far-fetched?

6. Conclusion

If the paradigm of grid computing is genuinely going to deliver on any of its much vaunted promises, at least within the academic domain, then one of two things needs to happen. We need to either:

- (a) invest about as much money as has already been invested in hiring professional software developers, training them in the areas of application development for grid computing and whatever domain specific areas they will need to support, and then employing them to develop grid applications for *every single academic who wishes to make use of grid computing*, or
- (b) invest in professional software developers, usability consultants, and practitioners of user-centred design to develop, from scratch, usable, domain specific grid middleware (that initially remains compatible with the existing middleware) and gradually retire the existing middleware to the museum of catastrophic failures.

The hardest part of this choice isn't the financing or logistics of either option (a) or (b) – it's admitting what I believe most of us in the UK e-Science community already know: the existing infrastructure is of almost no real use to anyone, and almost all the money that has been spent on it has been wasted. But not admitting it isn't going to change that.

And the hardest part of option (b) isn't developing something usable – we know how to develop usable software (we've known for some time, but we just choose not to), it's making an apparently difficult shift in approach to accepting that only the users know what they really need, and only they can tell us. Computer scientists, middleware developers, and funding

bodies certainly can't. At least, not without involving the users.

Acknowledgements

I would like to thank the University of Cambridge Computing Service for their support, particularly of e-Science activities within the University.

References

- [1] The Globus Toolkit: <http://www-unix.globus.org/toolkit/>
- [2] Roy, J.M.A. and Milunovich, S. "Globus Grid Computing—the Next Internet" in *TechStrat Barometer*, 4 January 2002, Merrill Lynch & Co: <http://www.grids-center.org/files/milunovich.pdf>
- [3] Ricadela, A. "Living On The Grid". *InformationWeek*, 17 June 2002: <http://www.informationweek.com/story/showArticle.jhtml?articleID=6504444>
- [4] McBride, S. and Gedda, R. "Grid computing uptake slow in search for relevance". *Computerworld Today*, 12 November 2004: <http://www.computerworld.com.au/index.php?id=138181333>
- [5] Ricadela, A. "Slow Going On The Global Grid". *InformationWeek*, 25 February 2005: <http://www.informationweek.com/story/showArticle.jhtml?articleID=60402106>
- [6] The Science Budget 2003-04 to 2005-06, The Department of Trade and Industry, UK: <http://www.ost.gov.uk/research/funding/budget03-06/dti-sciencebudgetbook.pdf>
- [7] Chin, J. and Coveney, P.V. Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware. (2004). UK e-Science Technical Report UKeS-2004-01: http://www.nesc.ac.uk/technical_papers/UKeS-2004-01.pdf
- [8] Pickles, S.M., Blake, R.J., Boghosian, B.M., Brooke, J.M., Chin, J., Clarke, P.E.L., Coveney, P.V., González-Segredo, N., Haines, R., Harting, J., Harvey, M., Jones, M.A.S., McKeown, M., Pinning, R.L., Porter, A.R., Roy, K. and Riding, M. The TeraGyroid Experiment (2004), *Proceedings of GGF10*, Berlin, Germany, 2004: <http://www.cs.vu.nl/ggf/apps-rg/meetings/ggf10/TeraGyroid-Case-Study-GGF10-final.pdf>
- [9] Research Councils UK: e-Science: <http://www.rcuk.ac.uk/escience/>
- [10] Rixon, G. Problems with Globus Toolkit 3 and some possible solutions (2003): <http://wiki.astrogrid.org/bin/view/Astrogrid/GlobusToolkit3Problems>
- [11] Beckles, B., Brostoff, S., and Ballard, B. A first attempt: initial steps toward determining scientific users' requirements and appropriate security paradigms for computational grids (2004). *Proceedings of the Workshop on Requirements Capture for Collaboration in e-Science*, Edinburgh, UK, 14-15 January 2004, pp. 17-43: http://www.escience.cam.ac.uk/papers/req_analysis/first_atempt.pdf
- [12] Gould, J.D. and Lewis, C. Designing for Usability: Key Principles and What Designers Think (1985). *Communications of the ACM*, Volume 28, Issue 3 (March 1985), pp. 300-311: <http://portal.acm.org/citation.cfm?id=3170>
- [13] Beyer, H. and Holtzblatt, K. Contextual Design: Defining Customer-centered Systems. Morgan Kaufmann Publishers, 1998.
- [14] Cooper, A. The Inmates Are Running the Asylum: Why High-tech Products Drive Us Crazy and How to Restore the Sanity. Sams, 1999.
- [15] Jenson, S. The Simplicity Shift. Cambridge University Press, 2002.
- [16] Beckles, B. User requirements for UK e-Science grid environments (2004). *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, UK, 31 August – 3 September 2004: <http://www.allhands.org.uk/2004/proceedings/papers/251.pdf>
- [17] Ellison, C.M. The nature of a usable PKI (1999). *Computer Networks* 31 (8) pp. 823-830.
- [18] Whitten, A., and Tygar, J. D. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. *8th USENIX security symposium*, Washington, DC, USA, 1999: http://www.usenix.org/publications/library/proceedings/sec99/full_papers/whitten/whitten.pdf
- [19] Gutmann, P. Plug-and-Play PKI: A PKI Your Mother Can Use (2003). *12th USENIX Security Symposium*, Washington, DC, USA, 2003: <http://www.cs.auckland.ac.nz/~pgut001/pubs/usenix03.pdf>
- [20] Beckles, B., Welch, V. and Basney, J. Mechanisms for increasing the usability of grid security (2005). *Int. J. Human-Computer Studies*, in preparation (July 2005).
- [21] Basney, J., Humphrey, M. and Welch, V. The MyProxy Online Credential Repository (1995). Software: Practice and Experience: <http://www.ncsa.uiuc.edu/~jbasney/myproxy-spe.pdf>
- [22] Nielsen, J. Usability Engineering. Morgan Kaufmann Publishers, 1993.
- [23] Cooper, A., and Reimann, R. About Face 2.0: The Essentials of Interaction Design. John Wiley and Sons, New York, 2003.
- [24] Sommerville, I. An Integrated Approach to Dependability Requirements Engineering (2003). *Proceedings of the 11th Safety-Critical Systems Symposium* (Bristol, UK, 2003), Springer, 2003, pp. 3-15.
- [25] Arnold, K. "Programmers are People, Too". *ACM Queue*, Volume 3, Number 5 (June 2005), pp. 54-59.
- [26] Venners, B. Perfection and Simplicity: A Conversation with Ken Arnold, Part I. 9 September 2002, Artima Software, Inc: <http://www.artima.com/intv/perfect.html>
- [27] Erickson, J. Embrace, Extend, Extinguish: Three Strikes And You're Out. *Dr. Dobbs' Journal*, Volume 25, Number 8 (August 2000), p. 8: <http://www.ddj.com/documents/s=882/ddj0008q/>
- [28] Knuth, D.E. Literate Programming (1984). *The Computer Journal*, 27 (2), May 1984, pp. 91-111: <http://www.literateprogramming.com/knuthweb.pdf>
- [29] Zurko, M.E. and Simon, R.T., User-centered security (1996). *Proceedings of the 1996 workshop on New security paradigms*, Lake Arrowhead, CA, USA, 1996, pp. 27-33: <http://portal.acm.org/citation.cfm?id=304859>
- [30] Adams, A. and Sasse, M.A. Users are not the enemy: Why users compromise security mechanisms and how to take remedial measures (1999). *Communications of the ACM*, Volume 42, Issue 12 (December 1999), pp. 40-46: <http://portal.acm.org/citation.cfm?id=322806>
- [31] Fléchaix, I., Sasse, M.A. and Hailes, S.M.V. Bringing Security Home: A process for developing secure and usable systems (2003). *Proceedings of the 2003 workshop on New security paradigms*, Switzerland, 2003, pp. 49-57: <http://portal.acm.org/citation.cfm?id=986664>
- [32] European Grid of Solar Observations (EGSO): <http://www.mssl.ucl.ac.uk/grid/egso/>
- [33] WSRF::Lite: <http://www.sve.man.ac.uk/Research/AtoZ/ILCT>
- [34] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters (2004). *6th Symposium on Operating System Design and Implementation*, San Francisco, CA, USA, 2004: <http://labs.google.com/papers/mapreduce.html>