Industry Experience in Using an Abstract Model to Select Software Development Tools

G. Zwartjes [*], J. v. Geffen [*], D. G. Kourie [+], A. Boake [+], B.W. Watson [* +]
[*] Technische Universiteit Eindhoven, The Netherlands.
[+] University of Pretoria, Pretoria, South Africa
{gzwartjes, jvgeffen, dkourie, bwatson}@cs.up.ac.za, andrew.boake@up.ac.za

Abstract: *Experience shows that it is difficult to select, from an ever-widening set of software tools, those appropriate for a given project. Complicating the choice beyond mere vendor selection is a proliferation of Open Source offerings. An abstract model is proposed to assist in this selection process. In this model, generic elements of a rational decision making process are identified. The model is realized in a matrix in which rows represent tools and columns represent tool properties. This tool matrix is developed iteratively, starting off with an initial attempt to articulate those tool properties perceived to be important in the project. The criticality of each property is also estimated. This information constitutes the column headings. Tools described in available information sources constitute the rows, with their compliance to the required properties indicated in the appropriate cells of the matrix. Rows are grouped together in functional categories – e.g. version systems, modeling tools, etc. Columns are grouped in terms of properties typically associated with a given functional category. The result is a visual representation of the extent to which identified tools meet properties. This serves as a basis for selecting one or more tools to meet the desired requirements. Iteration of the model consists of evaluating the coverage of these requirements, refinement of properties and trade off between alternatives. Experience in using the model in a number of industrial contexts is discussed. The case studies show that the process forces one to consider as many alternatives and prerequisites as are necessary, and to systematically make your reasoning explicit. This experience leads to the conclusion that the discipline demanded to construct the model has rational decisions as its payoff, based on well-assimilated information summarized in the tool matrix.*

1 Introduction

Software development tools can enhance the development process by assisting developers through the complexities of modern software development and by automating tedious tasks [8]. Choosing the right set of tools is very important, as well-chosen and appropriately used tools can contribute greatly to the success of a project. However, inappropriately chosen or ill-used tools are often serious obstacles that work against the development effort.

Tool choice today can be bewildering. Software development tools are abundant – the Open Source community [9, 10, 13, 15], in particular, has been prolific in this regard [14]. However, the available tools are varied in their approaches, support for critical aspects of different development methodologies, and their ability to work together as a cohesive set.

In the process of Open Source software adoption in industry, software development tools often lead the way. Finding the Open Source tools to be generally focused and useful in specific areas, developers use them in a 'development tool belt', or to complement the traditional tool sets purchased from vendors. This approach of course holds both promise and risks [15]. Examples of these areas include support for testing, building, logging, configuration management, deployment, monitoring, bug tracking and team collaboration – all aspects of an enterprise-scale development effort that are found to be essential by practitioners but are often neglected by popular integrated development environments. Another main problem of IDEs is the fact that it is often impossible to integrate new tools or swap existing ones. This is not a conceptual but a proactical problem caused by IDE developers.

This process very often takes place quietly, "below the radar", without due consideration as to why that particular combination of products was chosen or how well they fit the overall development task. This leads to decisions that are at

best tactical and at worst repeated mistakes.

The nature of Open Source tools adds its own unique difficulties. Because Open Source tool builders are often their own expert users, installing and maintaining their products takes expertise, and learning how to use their tools effectively through often sketchy documentation takes patience and time. The criteria for choosing, and necessary knowledge required to install, use and support the chosen tools is too often only in the heads of developers. This knowledge needs to be captured and made explicit to make tool decisions rational and transparent.

Through their careers, developers go to considerable effort to find, evaluate, choose, and then continue to use a set of tools that supports and molds their specific style of development. This often results in different members of a development team wanting to use different tools. Freedom is important in an empowered development team. However, when left unbound, it may lead to a proliferation of tactical solutions, brittle in their support and enigmatic to new team members, maintainers and managers. The software development process is difficult enough without these stumbling blocks [3, 5]. Again, in order to focus the necessary debates and decisions towards a unified set of tools, candidate tools need to be well characterized and the criteria for choosing between them clear.

The aim of this paper is to present an approach that may be followed in managing this process. The proposed abstract model promotes careful consideration and transparent documentation of the rationale behind selecting a set of software development tools. A matrix-based visualization is used in the abstract model. A similar technique is described in [16], to visualize the dependencies between software components and their design requirements.

At first reading, the model may appear to be nothing more than common sense – using a general decision-making process that may be applicable in many different areas. For this reason, practical examples from industry are examined in order to shed light on aspects of applying the model to choosing tools for specific contexts. While the logic behind the model is simple enough, the real effort is to discern the appropriate properties of tools being evaluated, and the criteria for choosing between them. Sharing this experience is valuable in itself.

The paper is structured as follows. First the abstract model itself is developed. After that, an explanation of the development context behind the specific examples is given. This is followed by the application of the model to the examples. Finally, some conclusions are given and further work is proposed.

2 An Abstract Model for Choosing Tools

In this section an abstract model is proposed to aid in choosing a set of tools. It is abstract in the sense that one may choose to implement the principles espoused in many different ways. The model consists of 5 steps, which can be executed iteratively, skipping some if appropriate in the given practical context. By executing the steps, a so called *tool matrix* is constructed. This matrix documents the compliance of tools being evaluated to stipulated requirements, and aids in choosing the set that complies best.

2.1 Application

The model is designed to be applied in diverse software engineering methodologies. For example, in a rigorous method where the phases are defined to be followed sequentially – e.g. the ESA Software Engineering Standard [6] – the steps of the model can be executed at the start of each phase. In addition, the selection of tools can be revised by iterating the model steps throughout a phase. In Agile development methodologies [1, 2], the model can be applied equally well. In such a methodology, where progress is designed to be incremental through iterations, the model steps can be followed from time to time to make sure that the set of tools being used still suffices.

2.2 Tool Matrix

The tool matrix is a visualization of the tool evaluation and selection process. The columns of the matrix contain the required properties of tools, and the rows of the matrix list the tools and their compliance to those properties. As part of defining the need for a specific requirement, a criticality is assigned to each of the required properties. These criticalities are used as criteria in choosing the final set of tools. The matrix evolves by iterating the steps of the model.

2.3 The Steps

Step 1. *Find or refine desired categories of tool support*
The first step in the process of selecting a set of tools is to specify the desired categories of tool support. The actions to take in this step depend on the status of the project. An initial list of desired tool categories is needed, if the project is in its beginning stages. As such, certain tools will be needed, for example a documentation system. As a result, a system to store those documents will be needed too. Often developers have experience in the most common shortcomings that have been encountered on previous projects. This is helpful in extending the list.

It is however unusual to find all desired categories at once. Finding and refining categories of desired tool support is a recurring process that should be repeated throughout the development stages. If the project is already some way down the path of development, the question is whether the set of currently used tools suffices. As development progresses, new types of support are needed, developers become more familiar with the tools, the tools are improved, and other tools become available. Each of these may lead to a new perspective on the type or degree of tool support needed and its availability. This will lead to a refinement of the current list.

Shortcomings in the current tool support can also be found by researching existing tools. Their usefulness for a particular development process can be appraised by studying documentation, consulting experience reports and experimenting with evaluation copies. Open Source tools are freely available and are easily found. Collaborative development environments like SourceForge [12] and especially Tigris [4] – which focuses on Open Source Software Engineering – provide access to tools and corresponding project information [14]. A variety of Open Source applications, including useful tools, are available via websites like Freshmeat [11].

As a start to identifying necessary categories of tools, a list of commonly used tools for a typical development process is given below. Robbins [14] gives a similar list.

- **Editors** – Can be used to edit plain text files, for example code source files. This can range from a simple line editor upto an editor with support for syntax highlighting, recording macros etc.
- **Documenting systems** – Can be anything from an advanced WYSIWYG editor to a professional document preparation system. The purpose of a documenting system is to produce digital or paper documents.
- **Version systems** – Help to keep track of different versions of documentation and source code. This helps to maintain software development artifacts among multiple developers.
- **Modeling tools** – Provide mechanisms to easily draw diagrams to be used in documentation or as a design tool for the project. Some modeling tools generate source code from models, and even extend to full-blown integrated development environments (IDE's).
- **Integrated development environments** – Provide a suite of integrated tools or a framework for integrating tools. Usually an IDE contains at least an editor and an integrated compiler.
- **Compilers** – Compile source code into binary form. Apart from being integrated into an IDE, a compiler will usually have a command line equivalent. The power of such a command line compiler, in terms of specifying exact parameters and order of compilation, should not be underestimated.
- **Code generators** – Generate code in some language given input in another language. Common examples are scanner and parser generators that take a description of a language and generate code to scan and parse source files in that language.
- **Code documenting systems** – Provide a clean and easy way to document source code, for example by using a special kind of comment in the source code itself. This is very helpful when the design by contract development method is used – see Hunt and Thomas [8].
- **Build systems** – Automate building the derived artifacts of a project, for example automating the process of getting from source code to an executable binary form.
- **Install systems** – Compile binaries into the form of a package that can be deployed on the Internet or distributed on some media to the end users. For example, for PC-based applications, this would provide the user of the application with an easy interface to install the application on his own PC.
- **Testing tools** – Assist in testing the source code of a project. Testing tools can be anything from a simple test bed upto an automated testing suite that runs overnight and emails status reports about the software every morning.
- **Bug- and issue tracking systems** – Give users and developers a consistent interface to report bugs and issues with the software. Acts like an electronic whiteboard.

Step 1 is finished when all required tool support categories have been identified.

Step 2. *Specify the required tool properties and their criticality in the tool matrix*
Once the desired tool support for the development process is known, the identified categories must be translated into desired tool properties. These may be in the form of desired features, but should also include quality attributes – such as usability, learning curve or platform support. These requirements are of course personal, but then a tool matrix is for personal, team-specific or project-specific decision support. It is also important to note that most tools will have more properties than those that are listed in the matrix. The matrix should however only contain properties that are relevant to the project.

The criticality of a property defines a measure of its perceived necessity. In the model, three levels of criticality are defined:

- **H** – High criticality. A property indicating a highly desired feature. The purpose of the model is to find a set of tools that together will support at least all required properties with this criticality.
- **M** – Medium criticality. A property with this criticality is desired, but tools that do not have support for this property may also suffice.
- **L** – Low criticality. A property that can determine the choice of a tool, when no decision can be made based on the properties with higher criticality.

These levels do not have to be so discrete. Any desired grading can be chosen in a practical implementation of the model. For example, an integer value between 0 and 5 can be used for a more fine-grained scale.

Step 3. *Populate the rows of the tool matrix with tools and its cells with indications of compliance*
When all required properties have been specified, the rows of the tool matrix are populated with tools and their compliance to those properties. The most important consideration here is which tools are selected to be part of the matrix, and what this selection is based on. Developers may have experience with previously used tools, and that can be useful. The internet can be searched for tools – especially for Open Source tools [12, 11, 4]. To search for tools in a specific category, the list in step 1 may be useful.

The cells of the matrix are populated for all the tools, by identifying the properties supported by a tool, and specifying its compliance in that cell. In this paper, we have used a simple Yes / No compliance measure – translating into black and white blocks in the diagrams – but also a more scaled measure of compliance may be adopted. Additionally a cell can be marked to indicate that a tool does not support a crucial property. This way the major bottlenecks of tools can be made explicit as well, which can be useful for future projects.

One or more features of a tool might be found that translate into a desired property that is not yet listed in the matrix. In this case, step 2 needs to be executed again, by adding the property to the matrix and defining its criticality.

Step 3 is finished when the compliance of all tools to the desired properties has been identified, and when no new properties or tools can be added. An example of a completely populated tool matrix is depicted in figure 1.

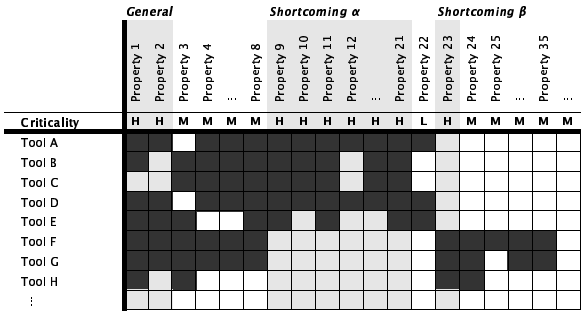

Figure 1. A completely populated tool matrix.

Step 4. *Analyze the tool matrix*
Once all required properties and tools are listed and the appropriate cells of the tool matrix have been populated, the matrix can be used to select the actual tools that *could* be used. Before the final selection of tools is made, the tool matrix must be analyzed as follows: If there is a critical property that is not covered by a single tool, which can be identified by an empty column in the tool matrix, three cases can be distinguished:

1. The tools have not been examined well enough, and a property of a tool already in the matrix has been missed. The property is thus supported after all. This type of problem can be avoided by showing the matrix to experienced tool users and evaluating the tools more thoroughly.
2. There are tools available that support the property, but none of them are listed in the matrix. Go back to step 3, find these tools, and include them in the matrix.
3. Functionality to support the property is not provided by any known tool. Tools which may support the desired property but are not being considered for some other reason should be included in the matrix, with their undesirable properties showing why they have not been selected. It is important to document both the positive and negative decisions.

   What if no tool is found to support a critical property? There are three possibilities:

   (a) Create a custom tool to support the unsupported property. The tool can either be developed internally, or someone else can be contracted to develop it. This decision must be made bearing in mind several factors, including the cost in time and money to develop and support the tool, and company motivation for such involvement.

   (b) Add the feature to an existing tool. Again this can either be done internally or a request can be made to have the feature included in a next release of that tool.

(c) Reassess the importance of the required property. This may involve decreasing the property criticality or dropping the requirement entirely.

At this point, the process can be continued.

Step 5. *Select a set of tools*
Go through the matrix and select an optimum set of tools. This set should of course cover all of the critical properties. The less critical properties can be important to make the decision if all the highly critical properties are supported in more than one tool. Choices among alternatives may however be dictated by personal preferences. For example, you may prefer an integrated development environment, or a collection of smaller, more focused tools. Coming to a decision may also force you to consider several additional factors. For example:

- Are there developers that have experience with the tool?
- Is the tool easy to use – e.g. does it have an intuitive Graphical User Interface?
- How well is the tool supported – e.g. is it still maintained, is it in beta stage or a stable version, is the tool available for the desired platform and so on?
- How much does it cost to buy a license or use the tool?
- How difficult is it to learn the tool – in terms of manhours?
- How well does the tool integrate into the existing set of tools?

It is difficult to imagine one ideal algorithm to determine the final choice of tools that will be used. The matrix provides a detailed trace of what is important in that choice, and the suitability of the candidates. Many of the above factors could also be added to the matrix, refining the desired properties. Others are more subjective, and it is up to the decision makers to determine the weight of those in the final decision. The matrix gives them a reasoned path up to that point.

3 In Practice

The examples in this section show how the model is applied in practice. The matrices are intended to be as precise and complete as possible.

3.1 Context

In 1991, work on an enterprise administration application for a medium-sized European company began. The team was relatively inexperienced and had little formal software engineering background. They experienced the usual teething problems, but persevered in a somewhat unstructured manner. However, 10 years later, the team realized that the application had become increasingly unmanageable, and something needed to be done for work to continue. In an attempt to introduce order into the development process and system design, a re-engineering exercise restructured the software into independently developed components. Tools were needed to support the development process of both the components and the applications themselves.

This development scenario forms the basis of the practical application of the model to select these tools.

3.2 The Component Development Process

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and is subject to composition by third parties. In other words one can define a component as a package containing a set of objects and all that is necessary to deploy them into an execution environment. These objects implement a coherent set of features that one can use in an application or another component. The design of their interaction may follow established design patterns [7]. A component honors specified interfaces and can be deployed independently. This means that independently from clients using it, errors may be corrected in, or enhancements can made to the component.

In the context of the above development scenario, the development of a component is begun by gathering information on what it is supposed to do, by interviewing the developers who will use the component. This information is used to write down the requirements formally. The next steps are to design and implement the component. A component is deployed as soon as possible and continuous response is provided on feedback that is supplied by developers that use the components –the idea is to 'release early and release often' [9, 15, 13]. Driven by feedback, requirements are added and refined, and problems are solved as soon as possible.

These components are developed in a separate office, that technically operates on its own. This imposes requirements on the availability and quality of the developed components. To minimize costs of the component development process we prefer Open Source tools, which are freely available, cover many different areas and platforms, and have no

restrictions on the number of people that may use the tool. Most of these tools are released often – new features are readily available and bugs are solved swiftly.

3.3 Choosing Tools for the Component Development Process

The abstract model from section 2 was applied to select a set of tools for the component development process described above. Not all examined tools are included, for the sake of brevity and readability. The following subsections walk through the process of choosing the tools, and explain the rationale behind the choices. The complete tool matrix is shown in figure 2.

As will be seen, each matrix in this section shows a set of general properties that are required for all tools. The difficulty with these general properties is specifying their criticality, because that can differ between tool categories. Again for the sake of readability, they are not included as specific tool category properties but are shown general properties for all the tools.

3.3.1 Compiler

As the components must be used in a Delphi application, Borland Delphi must be used to compile them. There are several flavors of Delphi, and three of them are listed in the matrix in figure 3. In this matrix, it can be seen that properties have been chosen to reflect the characteristics of the chosen development process. Here, an integrated development environment is favored by the component writers, while an automated build process requires command-line access to compiler functionality. In addition, both Windows and Unix support is required. These decisions are documented in the tool matrix, in a concise manner.

The only way to support all required properties is to use a combination of the three tools being evaluated. As main development environment, the Borland Delphi Enterprise IDE is used. To support builds from the command line – for the build system – the command line compiler is used. And, as the components must also be usable in Unix, Kylix is used for that platform.

3.3.2 Documenting system and modeling tool

In order to specify what a component is supposed to do, documents are created to contractually specify the requirements of a component and to illustrate its design. To this end, a documenting system and modeling tool are desired. The tool matrix in figure 4 summarizes the important properties and the investigated tools.

In reality, the initial choice for documenting and modeling tools was not based on a tool matrix. The shortcomings of the tools were encountered by using them. It was initially

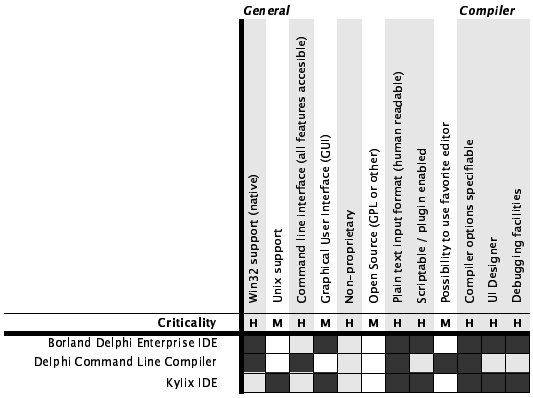

Figure 3. The compiler tool matrix.

decided that Microsoft Word would be used as a documenting and modeling tool; the tool matrix shows that this was not a good choice. For example, Word files are stored in a binary format, which makes it difficult to automatically merge a document that is changed by more than one developer concurrently. Creating design models with Word is time-consuming effort. The developers believe that Word does not supply a good mechanism for adding references in a document.

The setup of a separate office allowed for the choice of a different documenting system and modeling tool. MikTEX or LATEX in combination with BibTEX provide for all critical general and documenting system properties. MikTEX is actually a port of LATEX, BibTEX and related tools to the Windows platform including a graphical user interface. As the developers decided Unix support is more important than a GUI, LATEX together with BibTEX became the new documenting system.

Before the developers were introduced to MetaPost, first Xfig and later on Dia were used as modeling tools. The tool matrix shows that MetaPost is the best choice. With both Dia and MetaPost, the text in figures is easy manipulatable, hence the properties, methods and procedures of a class can be easily adapted. As Dia wasn't very stable and neither had a plain text input format nor command line interface, MetaPost is preferred.

It took a lot of time to switch between modeling tools, because every time all existing images had to be converted. It is therefore important to choose a good tool in advance, because migrating to a new tool can result in a waste of time spent in conversion.

3.3.3 Version system

The complete tool matrix in figure 2 seems to indicate that the choice for a version system might be easy. Subversion provides support for most properties. However, the first sta-

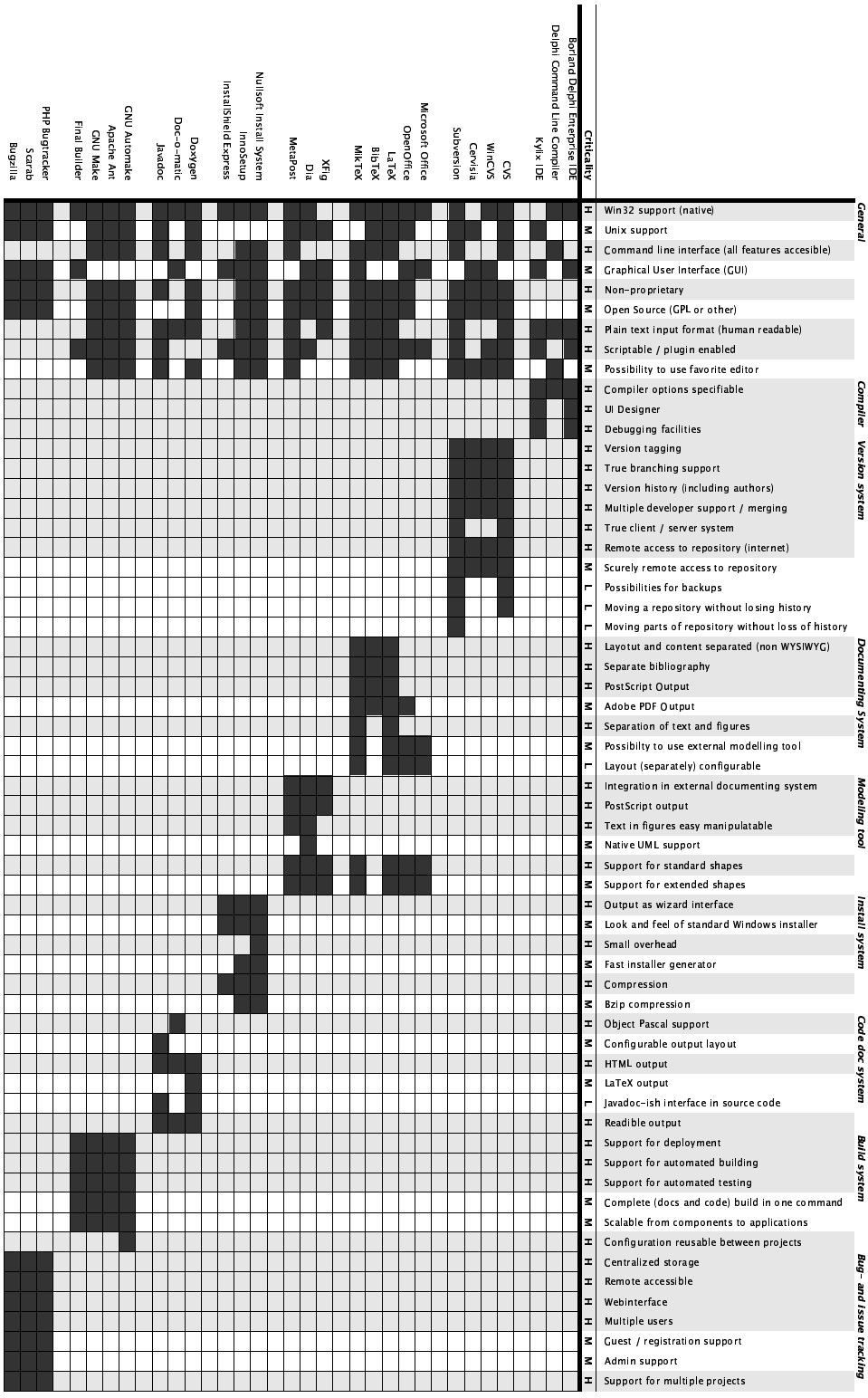

Figure 2. The fully populated matrix to select the tools for the component development process.

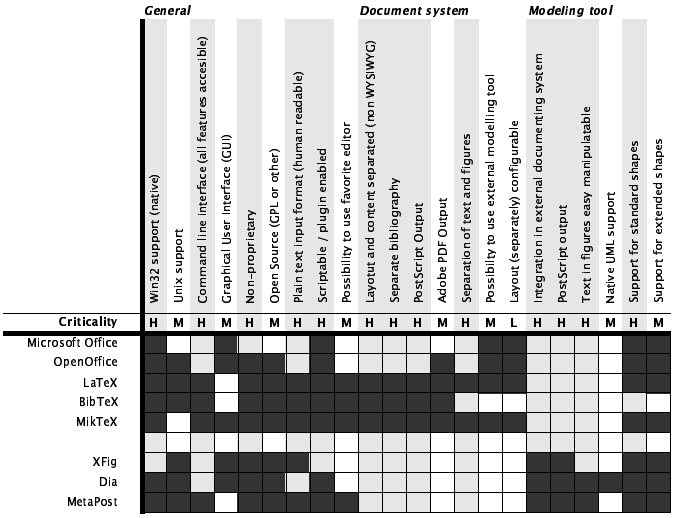

Figure 4. Tool matrix containing properties and tools for the document system and modeling tool.

ble release of Subversion was only available at the time of writing. No such release was available when the decision for a version system was made. Fortunately, CVS provides for all critical properties as well. In addition, most developers were already familiar with CVS.

Subversion and CVS do not provide a graphical user interface themselves, but there are third party GUI front-ends available for both version systems. Front-ends for CVS are WinCVS on the Windows platform and Cervisia on Unix platforms. The widespread use of CVS made it stable and secure. Maturity and stability of the tool are the important aspects here. By adding these properties to the matrix, depicted in figure 5, CVS is the right choice. However, Subversion seems very promising; it provides support for all critical and non-critical properties and it is possible to migrate a CVS repository to Subversion.

3.3.4 Build and install system

It was difficult to choose a build system. Because the individual component development projects all have more or less the same setup – they consist of several documents and several Delphi packages – a build system is needed that can be reused between projects. Figure 6 depicts the four investigated build systems. They all have more or less the same properties. Final Builder is included too, to compare the Open Source alternatives with a commercial one. Because Final Builder is proprietary, and Open Source tools are preferred, three alternatives remain. Only GNU's automake suite provides a way to setup a build for a generic project and reuse it by instantiating that setup for each project.

However, GNU automake would be hard to configure for the component development process, because it uses the M4 macro language and is designed specifically for Unix software. A new set of macros would have to be written for the applications used in the build process in a language in which none of the developers had experience. An extra property is needed here; the build system must be simple to configure. After adding the property to the matrix, it shows that none of the systems are both easy to configure and allow a generic setup between projects. Therefore, a customized build system was built.

The choice for install or deployment system was quite easy, as the matrix shows there is only one system that implements all required properties: Nullsoft Install System. InstallShield Express, a proprietary tool, is also included in the matrix. It is important to note that the choice of properties is most important – a different set easily results in a different choice. If the required properties were chosen differently, InstallShield Express could have come out as the tool that would fit the needs best.

3.3.5 Code Documenting System

The API documentation generation system was also created internally. The matrix depicted in figure 7 lists the key features of an API documentation generation system for our components. Doc-o-matic is a proprietary tool and it has a different interface for writing the code comments. The cost of learning it has to be taken into account. Doxygen could be an alternative, but in that case support for Object Pascal would have to be written. Also, most of the developers did not like the generated output of Doxygen. The developers had previously used Javadoc to create Java API documentation in another project. However, Javadoc is not an option as it is designed specifically for Java. No tool could be found

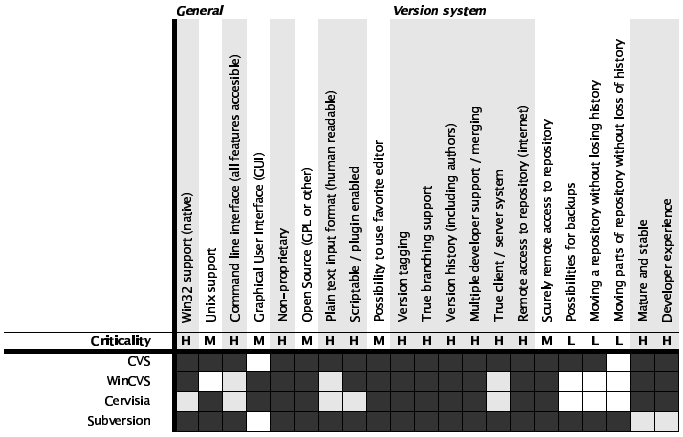

Figure 5. Version system tool matrix.

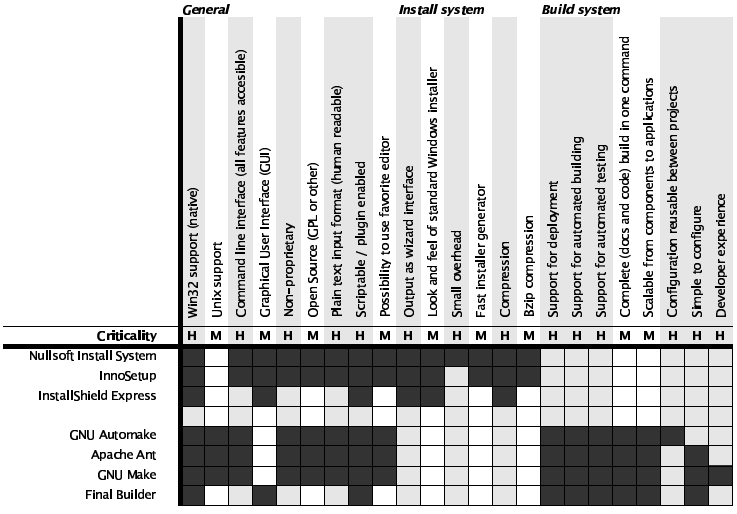

Figure 6. Tool matrix with only build and install system categories.

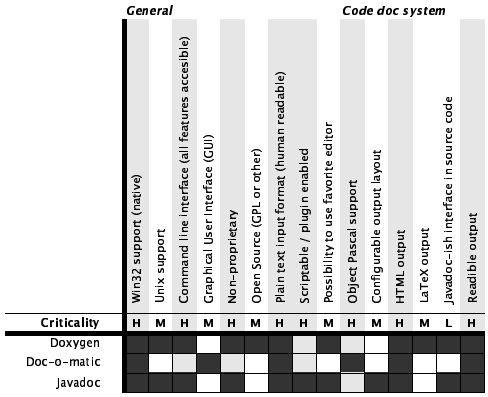

Figure 7. Code document system tool matrix.

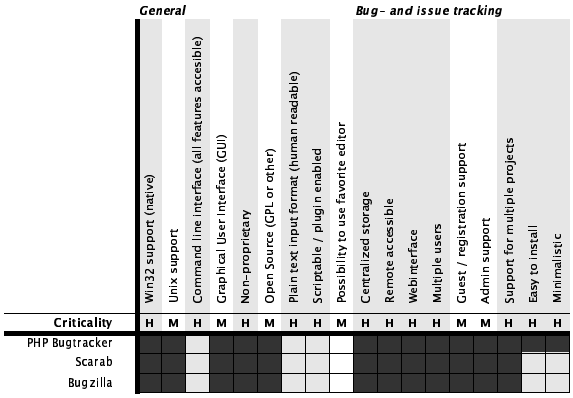

Figure 8. Bug- and issuetracking system tool matrix.

that would satisfy all the required properties. For that reason, we started a project called DelphiDoc, which is a clone of Javadoc for Object Pascal. The cost of its development is higher than the cost to buy Doc-o-matic, but it is a major advantage that it integrates seamlessly with the other tools.

3.3.6 Bug- and issuetracking system

In the original matrix it is not clear which system should be chosen. All systems in the matrix support all required properties, except for a command line interface, plain text input and scriptability which are not of such high criticality for a bug- and issuetracking system. This is why extra properties were added to differentiate between the tools. PHP Bugtracker was chosen because it was easy to install and the necessary server with required applications was already up and running. Another advantage of PHP Bug tracker is that it is a simple system. According to the developers, Scarab and Bugzilla were too elaborate for the needs of this project. See figure 8 for the bug- and issue tracker tool matrix.

3.4 Choosing Tools for the Enterprise Level Administrative Application

This section gives another practical example in which the model was used to validate the choice of tools to enhance the development process. Here, the software being developed was an enterprise level administration application. The tool matrix depicted in figure 9 is used in this case to analyze the currently used tools to see if they still suffice.

3.4.1 Version system

The developers had been using Microsoft Visual SourceSafe as version system during almost the entire development process. Though they felt they were missing quite a lot of functionality, they never really tried to introduce a new version system. True branching, for example, is not supported in SourceSafe. As in the previously described component development process, Subversion covers all the required properties. A stable version of Subversion was not yet available at the time of introducing a new version system. In this process too, CVS was the chosen version system. CVS together with WinCVS provided for all critical properties.

True branching support and version tagging were very essential in this case. Before the developers started using CVS as version system, bugs were solved and features were added in the main trunk of the development tree. Every time a version that solved a bug was released, new bugs would be introduced into that same release, because it contained new untested features. CVS provides a mechanism that makes it relatively easy to go back to previous releases. When a bug was reported by a client, the developers could then check out the version that the client was using, solve the bug in a separate branch, release that version to the client and merge the bug fix back into the main development trunk. The separate branch in which the bug was solved did not contain new untested features, so no new bugs were introduced in the bug fix release.

3.4.2 Autobuild and install system

The developers in this case did not have a generic build system. Every developer used to build the software within the IDE and an actual release was built on a separate system using a custom build system. There were problems with that setup because the configuration of both build processes – IDE and build system – differed widely. For example, the IDE build statically linked the software, while the actual release version was dynamically linked. So, the problems of

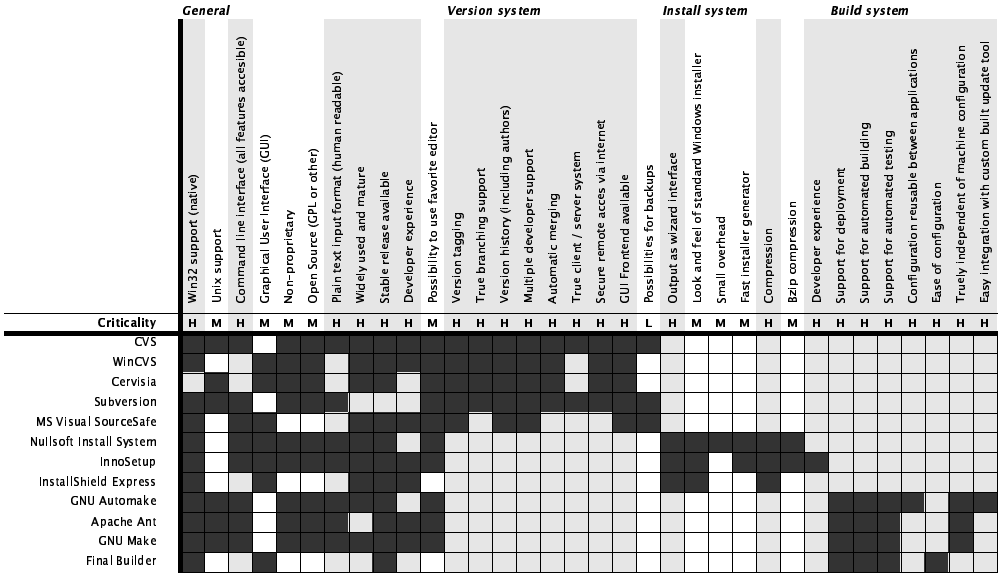

Figure 9. Tool matrix listing tools to improve the development process of the administrative application.

dynamically linking the new features only came to surface when the actual version was due to be released. As a result, the dynamic build process and the actual release sometimes took as long as a complete day.

The tool matrix depicted in figure 9 shows that no tool implements all of the critical properties. A custom build system had already been built for the component development process and a slightly modified version could be used for the application. The matrix however shows that support for the application and its custom built update tool needs to be added. Innosetup was used to package the application so logically it would be a wise choice to keep Innosetup and incorporate it into the build system.

At the moment the system is being used and has significantly improved build times. The time of build and actual release is reduced from an average of 4 hours to an average of 5 minutes. Because there is now only one automated way of building the system, this implies that a release version can be built if a developer can do a local build.

4 Conclusion

Choosing the right set of tools is important. A well-chosen set of tools can cut down on development costs significantly. Open Source development tools are abundant and freely available, and usually well written. Therefore they are an important source from which to choose tools. Building a custom tool may well improve the development process too, but one should be careful in making the decision to build a new tool internally.

A model to assist in choosing a set of tools was introduced. On the surface, it looks just like common sense, but when the process of drawing up the matrix is undergone, a lot of issues will surface that would otherwise have never come to mind. Practical industry examples were given to illustrate the usefulness of the abstract model.

As an aside, at the time of writing this article, the version system tool matrices lead to a new insight. The practical examples showed that Subversion implements all required and critical properties that are not implemented in CVS. Because Subversion has now stabilized, it is worthwhile trying Subversion instead of CVS as a version system for the next project.

In most cases, tool choice relies mainly on experience and gut feel. These can be imprecise and misleading. Drawing up a tool matrix formalizes the reasons for a choice, which can be reviewed or defended. A tool matrix can also highlight reasons for problems in tools already being used.

A tool matrix can be applied for personal use, in the sense that it is instructive just to build up the matrix. It can be used within a group of developers to agree on a set of tools or to convince management to buy a tool that is needed. It can also show that an Open Source tool fits the needs just as well as – or even better than – some expensive proprietary tool. It can be used as a valuable guideline for other project teams, or the next similar project. The tool matrix has many practical purposes; it itself is another tool to be included in a developers toolbox.

5 Future Work

Interesting work still needs to be done to support the process that is described in this article. To make it easier for people that are in the process of creating a tool matrix to fill out the required properties, template tables that summarize required properties of a common tool category could be constructed. In these tables no criticalities would be assigned to properties, because criticalities are subjective.

Generic tool matrices listing the properties of available Open Source tools would also be helpful in the process of drawing up a tool matrix. These tables would need to be revised quite often because tools are constantly being extended or changed, and new tools emerge on a regular basis.

With a tool to create and manage the matrix, the tool matrix can be made more powerful than just showing binary Yes / No compliance. Users can give weights to requirements. Then, the tool colors the matrix to show the compliance, from for example green to red and shades inbetween. Users can then easily spot color patterns and infer if the matrix is stable, unstable, and so on. Moreover, the user can change the weights, and the tool updates the matrix. Finally, and most important, the tool can compute overall quality metrics for the tools, e.g. by taking the sum of the tool property weights, and give a overall score for every tool. By coloring these scores accordingly, the tools that are the best in their class can be immediately shown. In addition, such a tool could be able to assist in making the decision for the final set of tools.

The abstract model introduced in this article is specifically designed to choose tools in a Software Engineering process, but could conceivably be more generally applicable. The way of making a decision, as described by the model, could also be used in a generic approach to documenting the decision making process in other areas.

References

[1] P. Abrahamsson, O. Slao, J. Ronkainen, and J. Warsta. Agile Software Development Methods: Reviews and Analysis. *ESPOO 2002*, 2002.

[2] K. Auer and R. Miller. *Extreme Programming Applied: Playing to Win*. Addison-Wesley, 2003.

[3] F. P. Brooks. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[4] Collabnet, Inc. Tigris.org: Open source software engineering. [Online]. Available: http://tigris.org/, 2004.

[5] T. de Marco and T. Lister. *Peopleware, Productive Projects and Teams, 2nd edition*. Dorset House Publishing Co, 1999.

[6] European Space Agency. *European Space Agency Software Engineering Standards ISSUE 2*. European Space Agency, 1994.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[9] G. Moody. *Rebel Code: Linux and the Open Source Revolution*. Persues Publishing, 2002.

[10] Open Source Initiative. Open Source. [Online]. Available: http://www.opensource.org/, 2004.

[11] OSDN. Freshmeat.net. [Online]. Available: http://www.freshmeat.net/, 2004.

[12] OSDN. SourceForge.net. [Online]. Available: http://www.sourceforge.net/, 2004.

[13] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, 2001.

[14] J. E. Robbins. Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools. *To appear in Making Sense of the Bazaar: Perspectives on Open Source and Free Software*, Fall 2003.

[15] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2002.

[16] F. van Ham. Using Multilevel Call Matrices in Large Software Projects. *IEEE Symposium on Information Visualization*, pages 227–232, 2003.