

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

G. Zwartjes <sup>\*</sup>, J. v. Geffen <sup>\*</sup>, D. G. Kourie <sup>+</sup>, A. Boake <sup>+</sup>, B.W. Watson <sup>\* +</sup>

<sup>\*</sup> Eindhoven University of Technology, Eindhoven, The Netherlands.

<sup>+</sup> 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 ones 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 [9]. 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. Besides the large number of tool vendors, also the open source community [15, 16, 19, 23] has been prolific in this regard [20]. However, the available tools are varied in their approaches, their support for critical aspects of different development methodologies, and their ability to work together as a cohesive set. Because open source software is freely available and the number of open source development tools are is large and still increasing, open source software is a valuable resource to choose from. In addition, there is an increasing number of industry examples that choose open source tools in favor of proprietary tools.

In this 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. 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. The open source approach of course holds both promise and risks [23]. Open source development tool selection plays an increasingly prominent role and therefore

will be considered important in the article as well.

The tool selection 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, 6]. 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.

There was a felt need for more order. After a couple of brainstorming sessions the authors proposed an approach that may be followed in managing the selection process. The approach was applied retrospectively to two case studies. The aim of this paper is to present the approach. The proposal is an abstract model that promotes careful consideration and transparent documentation of the rationale behind selecting a set of software development tools.

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 two specific contexts. In both industry examples open source tools are preferred over proprietary tools. The approach is however not only suitable for open source tools but also for proprietary tools, or a mix. 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. A brief overview of previous work related to this contribution is provided in sec-

tion 2. In section 3 the abstract model itself is described. After that, the two case studies are presented, in sections 4 and 4.3 respectively. Finally, some conclusions are given and further work is proposed section 5.

## 2 Related Work

Starting with the emergence of software engineering as a field of research, increasingly advanced tools have been developed to address the difficulties of software development. The term Computer Aided Software Engineering (CASE) was introduced in the '70s for the tools to speed up the software system building process: a new generation of tools that apply heavyweight engineering principles to the development and analysis of software specifications. CASE tool evaluation has been assessed by several studies, for example the impact on systems development process by Jankowski [11], a framework for assessing CASE usage by Sharma and Rai [21], CASE tool evaluation at Nokia by Maccari and Riva [14] and a method for CASE tool evaluation by du Plessis [7]. However, all these approaches are targeted at evaluation per CASE tool. The approach in this article compares a set of tools to select a subset for a predefined purpose.

Le Blanc and Korn propose a phased approach to evaluate CASE tools and how the final selection for a CASE tool is made [13]. The writers define three phases that must be traversed during the process of selecting an appropriate CASE tool. In phase 1, prospective candidates are screened and a small list of CASE tools is created. In this phase a preliminary screening is made by looking at uncommonly provided functionality. In phase 2, a CASE tool candidate is selected that best suits the system development requirements. This can of course only be done when a suitable CASE tool exists for this particular environment. Phase 3 is the actual match between the user requirements and the features of the selected CASE tool. We propose a similar approach, by using a matrix to visualize the reasoning, for choosing software development tools.

A study of CASE tool adoption by Iivari found that adoption correlates negatively with end user choice, and concludes that successful introduction of CASE tools must be a top-down decision from upper management [10]. The results of this approach has repeatedly been *shelfware*: software tools that are purchased but not used [20]. Research by Stobart and associates, and Sharma and Rai show similar results [22, 21]. Therefore, this article promotes the decision to be at the developer level, and not at management level.

Many finer grained distinctions have been tried to differentiate between classes of CASE tools. An example is the distinction between those tools that are interactive in nature, such as a design method support tool, and those that are not, such as a compiler. The former class are sometimes identi-

fied as CASE tools. The latter class are called development tools. Unfortunately, these distinctions are often problematic. In the example, it is difficult to give a simple and consistent definition of *interactive* that is meaningful. Therefore, no distinction between classes of tools is made in the following text and the term *tool* is reserved for a computer program for software developers to help create or maintain other programs. Hence it includes CASE tools as well.

An attempt to define practices for choosing and evaluating software engineering methods and tools, is the DESMET project [12]. The DESMET evaluation methodology separates evaluation exercises into quantitative and qualitative evaluations. The qualitative (subjective) evaluation is aimed at establishing the appropriateness of a tool of method. In other words, to what extent does a tool or method fulfill the needs of an organization? “The appropriateness of a method or tool is usually assessed in terms of the features provided by the method or tool, the characteristics of its supplier and its training requirements.” A qualitative evaluation is often based on the personal opinion of the evaluator.

The model we propose in the next section is based on such qualitative analysis of the tools and properties of a tool. The evaluation of a tool is based on literature describing the tool and, if applicable, based on experience with that tool. The proposed model is a visualization of an evaluation of several tools.

### 3 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.

#### 3.1 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 – the ESA Software Engineering Standard [8] and Fusion [4] for example – 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.

#### 3.2 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.

#### 3.3 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 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 ideal for experimenting, as they are freely available and easily found. Collaborative development environments like SourceForge [18] and especially Tigris [5] – which focuses on open source software engineering – provide access to tools and corresponding project information [20]. A variety of open source applications, including useful tools, are available via websites like Freshmeat [17].

As a start to identifying necessary categories of tools, a

list of commonly used tools for a typical development process is given below. Robbins [20] 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 up to an editor with support for syntax highlighting, recording macro's 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 add-in for 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 [9].
- **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 Upton 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 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

Criticality	General				Category $\alpha$								Category $\beta$						
	Property 1	Property 2	Property 3	Property 4	...	Property 8	Property 9	Property 10	Property 11	Property 12	...	Property 21	Property 22	Property 23	Property 24	Property 25	...	Property 35	
Tool A	H	H	M	M	M	M	H	H	H	H	H	H	L	H	M	M	M	M	M
Tool B																			
Tool C																			
Tool D																			
Tool E																			
Tool F																			
Tool G																			
Tool H																			
...																			

Figure 1. A completely populated tool matrix.

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 [18, 17, 5]. 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.

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.

#### 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. Maybe there are tools available that support the property, but none of them are listed in the matrix, because the tools were not found. Go back to step 3, try searching again, and include the tools in the matrix if found.
3. No known tool supports functionality for the required property. 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.

In this case, 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

In this step, an optimum set of tools is selected by going through the matrix. 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 task-oriented 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 – does it have an intuitive graphical user interface?
- How well is the tool supported – 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 man-hours?
- 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 previously mentioned factors could also be added to the matrix, refining the desired properties. The scope of what is included (and what is not included) in the matrix as well as the determining the weight of the factors in the final decision is up to the decision makers. The matrix gives them a reasoned path up to that point.

## 4 Case Studies

The case studies described here relates to ongoing development work that is being carried out by the first two co-authors. It involves the retrospective application of the model during the first quarter of 2004 to a situation that had evolved over the preceding three years. It indicates how the use of the model exposes both the strengths and the weaknesses in various tool choices that had been made up to that point.

### 4.1 Context

In 1991, a medium sized Dutch software house of about 10 developers, set out to produce an enterprise administration system, targeted at small to medium sized companies. The development team was relatively inexperienced and had little formal software engineering background. Nevertheless, despite the usual teething problems, they persevered — albeit in a somewhat unstructured manner — successfully developing and marketing the system to a client base of approximately 300 installations. Unsurprisingly, the system evolved over time in response to client and market needs. After some 10 years, the team realized that the application had become increasingly unmanageable, and something needed to be done for work to continue.

One of the steps taken was to contract the services of the first two co-authors of this article as *consultants*<sup>1</sup> to augment the efforts of the existing *developers*. Another step taken was an overall re-engineering exercise aimed at restructuring the software into *components*.

<sup>1</sup>Actually the consultants are developers as well, but for the sake of clarity, the above terms will be used in the following discussion in order to distinguish between the two parties.

Practical considerations required that the consultants working in a remotely located office, but limited face-to-face interaction with the existing developers was possible. The primary task of the former was to implement the components that were to be integrated into the system by the latter. The development of each component therefore required close collaboration between the two parties. Once a component's requirements had been agreed upon and documented by the relevant parties, the consultants designed and implemented it. Their general approach was to *release early and release often*, from the open source development process [15, 23, 19]. Thus, component evolution was driven by developer feedback, sometimes resulting in requirements being added and refined as necessary, the aim always being to resolve problems as soon as possible.

Early on, the consultants recognized the need for tools to support the development process of both the individual components and of the overall application itself. They chose tools to meet their needs as they arose, according to their best judgment in each case. In some cases, the tool would not only have to suite their own personal needs, but would also be required to integrate into the wider context of the developers — who had their own set of skills and needs.

### 4.2 Case Study 1: The Components

The discussion below indicates how the abstract model from section 3 was applied after the fact to eight different tool categories used by the consultants in the component development process. This is not an exhaustive list of tools used, but adequately illustrates the construction and use of the model. Since the case study illustrates the use of the model at a later iteration of the software process, the desired category of tool support was already known at the time of its application. As a result, it was not necessary to carry out the first step recommended in section 3.3.

The second step in section 3.3, *specify the required tool properties and their criticality*, was perhaps the most challenging part of the case study. Articulating what should constitute a list of desired properties proved to be quite challenging and evoked much debate, the details of which will not be recounted further in the narrative to follow. The eventual set of properties used and their criticality assignments represent the consensus view of the consultants, reflecting their perceived priorities and preferences in a particular context. Future applications of the model in other contexts could perhaps reuse some of these properties, but their appropriateness should be thoroughly re-assessed. Thus, for example, the consultant's preference for open source tools should not be construed as a universal endorsement open source in all contexts. The widely acknowledged potential hidden costs associated with open source (especially in regard to limited skills) is an issue that is orthogonal to the

application of the model, and should be incorporated as a property in the model if relevant to a given context.

The third step in section 3.3, in which *the rows of the tool matrix are populated and the appropriate cells are filled in*, was not too difficult, since the consultants were well aware of many alternative tools for the tasks that they had been undertaking. Nevertheless, applying the step did involve a number of web searches to discover new tools that might have become available, and also, in certain cases, to verify or discover whether particular products had a required property. In this regard, the version system is an important example where a new system was discovered that could replace the current.

The following subsections walk through the eight different tool category sub-matrices, recounting some of the analysis, insights and conclusions elicited by their construction. These subsections therefore focus on the fourth and fifth steps of section 3.3.

For completeness, the initial overall tool matrix for the eight tool categories is shown in figure 2. Note that the first nine columns enumerate general properties that were considered to be relevant to all categories. Associated criticality values are also specified. However, the criticality values in this figure are nominal: in the more detailed analysis reflected in later figures they differ from one tool category to the next. Also, some of the later figures include additional category-specific properties that are not displayed in figure 2. It was only during the analysis step that the need for these properties was perceived. This illustrates the iterative nature of the model's use and development.

#### 4.2.1 Compiler

As the components were to be used in a Delphi application, Borland Delphi had to be used to compile them. There are several flavors of Delphi, and three of them are listed in the matrix in figure 3. Properties in the matrix reflect the characteristics of the development process. For example, the development process incorporated an automated build process which, in turn, required command line access to compiler functionality — hence a high criticality to the property requiring a command line interface. In addition, Windows was required by the developers while Unix was preferred consultant operating system. The criticality levels for these properties reflect these needs.

Entries in the first three columns indicate that the only way to support all three of these properties is to use a combination of the three tools being evaluated. This was, in fact, done. The Borland Delphi Enterprise IDE was used as the main developer environment. To support builds from the command line — for the build system — the command line compiler was used. And, since the components also had to be usable in Unix, Kylix was used for that platform.

	General								Compiler			
	Win32 support (native)	Unix support	Command line interface (all features accessible)	Graphical User Interface (GUI)	Non-proprietary	Open Source (GPL or other)	Plain text input format (human readable)	Scriptable / Plugin enabled	Possible to use favorite editor	Compiler options specifiable	UI Designer	Debugging facilities
Criticality	H	M	H	M	M	H	H	M	H	H	H	H
Borland Delphi Enterprise IDE												
Delphi Command Line Compiler												
Kylix IDE												

Figure 3. Compilers.

#### 4.2.2 Documenting- and modeling tools

In order to specify what a component is supposed to do, documents were created to contractually specify the requirements of a component and to illustrate its design. To this end, a documenting system and modeling tool were needed. The tool matrix in figure 4 summarizes the important properties for these two tool categories and lists the documenting systems and modeling tools that were investigated.

The initial choice for documenting and modeling tools was not based on a tool matrix. Instead, it had been decided to use Microsoft Word both for documenting and for modeling. The tool matrix highlights the weakness of this 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 — depicted by the plain text input format property. It was felt that Word does not supply a convenient mechanism for adding references in a document and there is no direct support for UML in Word so that creating design models became tedious and time-consuming.

The relative independence of the consultants (who were located remotely from the developers) allowed them to choose alternative documenting- and modeling tools to Word, which was the tool that the developers continued to use. As the consultants had decided that Unix support was more important than having a GUI, they selected L<sup>A</sup>T<sub>E</sub>X together with BibT<sub>E</sub>X as their documenting system. However, the table indicates that MikT<sub>E</sub>X provides all general and documenting system properties that have a high criticality level, but on a Windows platform. Furthermore, it includes a graphical user interface. In fact, it is a port of L<sup>A</sup>T<sub>E</sub>X, BibT<sub>E</sub>X and related tools to Windows, and would have been chosen if the consultants were obliged to work in a Windows environment.

Xfig was the first modeling tool that was used. Subse-

Criticality	Borland Delphi Enterprise IDE	Delphi Command Line Compiler	Kylix IDE	CMS	WinCIS	Cervisia	Subversion	Microsoft Office	OpenOffice	LaTeX	BiD EX	MIKTeX	XFIG	Dia	MeatPost	NuIsort install System	InnoSetup	InstallShield Express	Doxygen	Doc-o-matic	Javadoc	GNU Automake	Apache Ant	GNU Make	Final Builder	PHP Bugtracker	Scrabd	Bugzilla
General	Win32 support (native)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	
	Unix support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Command line interface (all features accessible)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Graphical User Interface (GUI)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Non-proprietary	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Open Source (GPL or other)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Plain text input format (human readable)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Scriptable / plugin enabled	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Possible to use favorite editor	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Compiler options specifiable	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Compiler	UI Designer	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Debugging facilities	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Version tagging	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	True branching support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Version history (including authors)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Multiple developer support / merging	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	True client / server system	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Remote access to repository (internet)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Securely remote access to repository	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Backups possible	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Documenting System	Moving a repository without losing history	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Moving parts of repository without loss of history	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Layout and content separated (non WYSIWYG)	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Separate bibliography	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	PostScript Output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Adobe PDF Output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Separation of text and figures	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Possible to use external modelling tool	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Layout (separately) configurable	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Integration in external documenting system	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Modelling tool	PostScript output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Text in figures easy manipulatable	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Native UML support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for standard shapes	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for extended shapes	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Output as wizard interface	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Look and feel of standard Windows installer	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Small overhead	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Fast installer generator	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Compression	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Code doc system	Bzip compression	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Object Pascal support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Configurable output layout	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	HTML output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	LaTeX output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Javadoc-ish interface in source code	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Readable output	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for deployment	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for automated building	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for automated testing	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Build system	Complete (docs and code) build in one command	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Scalable from components to applications	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Configuration reusable between projects	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Centralized storage	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Remote accessible	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Webinterface	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Multiple users	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Guest / registration support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Admin support	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H
	Support for multiple projects	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H	M	H

Figure 2. Case study 1 matrix.

Criticality	General										Document system					Modeling tool						
	Win32 support (native)	Unix support	Command line interface (all features accessible)	Graphical User Interface (GUI)	Non-proprietary	Open Source (GPL or other)	Plain text input format (human readable)	Scriptable / plugin enabled	Possible to use favorite editor	Layout and content separated (non WYSIWYG)	Separate bibliography	PostScript Output	Adobe PDF Output	Separation of text and figures	Possible to use external modelling tool	Layout (separately configurable)	Integration in external documenting system	PostScript output	Text in figures easy manipulatable	Native UML support	Support for standard shapes	Support for extended shapes
Microsoft Office	H	M	H	M	H	M	H	H	M	H	H	H	M	H	M	L	H	H	H	M	H	M
OpenOffice																						
LaTeX																						
BibTeX																						
MikTeX																						
XFig																						
Dia																						
MetaPost																						

Figure 4. Documenting- and modeling tools.

quently it was decided to change to Dia. The matrix indicates that text in figures is easy to manipulate on both the Dia and MetaPost systems. Thus it is easy to change property, method and procedure entries in a UML class diagram. However, Dia does not support plain text input format and does not provide a command line interface. To this extent, the tool matrix highlights the fact that MetaPost should have been the preferred system.

Indeed, there was even a greater incentive to switch over to the MetaPost system, since experience with Dia showed that it was not very stable. (This suggests that *stability* should be considered as a property in future iterations of the modeling exercise). However, there is a built-in inertia in changing from one tool to the next. In the case of switching modeling tools, the inertia is particularly severe, because all existing images have to be converted to the new system. Nevertheless after tackling the conversion, the consultants have now switched to MetaPost successfully.

#### 4.2.3 Version system

At first glance, the overall tool matrix in figure 2 might suggest that the best version system choice is obvious: according to that figure, Subversion provides support for most properties. However, the first stable release of Subversion was only available at the time of constructing the model. No such release was available when the decision for a version system had to be made. Fortunately, CVS also provides for all critical properties. 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 are particularly important in a version system tool. When these properties are included in the matrix, as in figure 5, CVS emerges as the most appropriate choice. Nevertheless, the matrix highlights the fact that 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. For this reason, it could be considered as a tool once it has reached stability.

#### 4.2.4 Install- and build systems

The matrix in figure 6 shows why the choice of an install or deployment system was quite easy. The Nullsoft Install System is the only system to implement all required properties. 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.

On the other hand, the consultants had difficulty in choosing 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 – the build system had to be reusable from one component development project to the next. Figure 6 depicts four investigated build systems. They all have more or less the same properties. Since Final Builder is proprietary, and since open source tools were strongly preferred, three alternatives remain. Only GNU's automake suite provides a way of setting up a build for a generic project and

Criticality	OS				
	Win32	Unix	Centos	Suse	
General	H	M	H	M	Win32 support (native)
	M	M	M	M	Unix support
	M	H	M	M	Command line interface (all features accessible)
	M	M	M	M	Graphical User Interface (GUI)
	M	M	M	M	Non-proprietary
	M	M	M	M	Open Source (GPL or other)
	M	M	M	M	Plain text input format (human readable)
	M	M	M	M	Scriptable / plugin enabled
	M	M	M	M	Possible to use favorite editor
	M	M	M	M	Output as wizard interface
Version system	M	M	M	M	Version tagging
	M	M	M	M	True branching support
	M	M	M	M	Version history (including authors)
	M	M	M	M	Multiple developer support / merging
	M	M	M	M	True client / server system
	M	M	M	M	Remote access to repository (Internet)
	M	M	M	M	Securely remote access to repository
	M	M	M	M	Backups possible
	M	M	M	M	Moving a repository without losing history
	M	M	M	M	Moving parts of repository without loss of history
Developer experience	M	M	M	M	Mature and stable
	M	M	M	M	Developer experience

Figure 5. Version systems.

Criticality	Install System					
	Nullsoft	InnoSetup	InstallShield Express	GNU Automake	Final Builder	
General	H	M	M	M	M	Win32 support (native)
	M	M	M	M	M	Unix support
	M	M	M	M	M	Command line interface (all features accessible)
	M	M	M	M	M	Graphical User Interface (GUI)
	M	M	M	M	M	Non-proprietary
	M	M	M	M	M	Open Source (GPL or other)
	M	M	M	M	M	Plain text input format (human readable)
	M	M	M	M	M	Scriptable / plugin enabled
	M	M	M	M	M	Possible to use favorite editor
	M	M	M	M	M	Output as wizard interface
Install system	M	M	M	M	M	Look and feel of standard Windows installer
	M	M	M	M	M	Small overhead
	M	M	M	M	M	Fast installer generator
	M	M	M	M	M	Compression
	M	M	M	M	M	Bzip compression
	M	M	M	M	M	Support for deployment
	M	M	M	M	M	Support for automated building
	M	M	M	M	M	Support for automated testing
	M	M	M	M	M	Complete (docs and code) build in one command
	M	M	M	M	M	Scalable from components to applications
Build system	M	M	M	M	Configuration reusable between projects	
	M	M	M	M	Simple to configure	
	M	M	M	M	Developer experience	
	M	M	M	M	Developer experience	

Figure 6. Install- and build systems.

then reusing 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 macro's would have to be written for the applications used in the build process. Moreover, the consultants were not familiar with the language in which this would have to be done. The matrix shows that the *simple to configure* property was deemed to have a high criticality. As a result, the matrix confirms that none of the systems are both easy to configure and allow for a configuration that is reusable between projects. To this extent, the matrix supports the consultant's prior decision: to develop their own customized build system. This had been done earlier, and was working well. There appeared to be no need to adopt a new tool in its place.

#### 4.2.5 Code Documenting System

The matrix depicted in figure 7 lists the key features that were considered important for a documentation generation system to be used for the components that were being developed. Doc-o-matic is a proprietary tool. In addition to this disadvantage, its interface for producing documentation is disjoint from the interfaces the consultants are already familiar with. Furthermore, the cost of learning it also has to be taken into account. Doxygen could have been an alternative, but for the fact that it lacks support for Object Pascal. In addition, most of the developers did not like the generated output of Doxygen.

The consultants had previously used Javadoc to create Java API documentation in another project. However, Javadoc was not an option as it is designed specifically for Java.

The matrix therefore indicates that no suitable tool could be found that would satisfy all the required properties and supports the consultant's prior decision to create their own API documentation generation system. They had called their system DelphiDoc, which is an replica of Javadoc that is tailored for Object Pascal. The cost of its development was higher than the cost of buying Doc-o-matic, but the fact that it integrates seamlessly with the other tools is considered to be a major advantage.

#### 4.2.6 Bug- and issue-tracking system

In the overall matrix in figure 2 it is not clear which bug- and issue-tracking system should be chosen. All relevant systems in that matrix support all required properties, except for a command line interface, plain text input and scriptability which were not considered to have a high criticality.

In reconsidering the matter in order to generate figure 8 for the bug- and issue tracker tool matrix, extra properties

Criticality	General										Code doc system				
	Win32 support (native)	Unix support	Command line interface (all features accessible)	Graphical User Interface (GUI)	Non-proprietary	Open Source (GPL or other)	Plain text input format (human readable)	Scriptable / plugin enabled	Possible to use favorite editor	Object Pascal support	Configurable output layout	HTML output	LaTeX output	Javadoc-ish interface in source code	Readable output
Doxygen	H	M	H	M	H	M	H	M	H	M	H	M	H	M	L
Doc-o-matic															
Javadoc															

Figure 7. Code document systems.

Criticality	General										Bug- and issue tracking							
	Win32 support (native)	Unix support	Command line interface (all features accessible)	Graphical User Interface (GUI)	Non-proprietary	Open Source (GPL or other)	Plain text input format (human readable)	Scriptable / plugin enabled	Possible to use favorite editor	Centralized storage	Remote accessible	Webinterface	Multiple users	Guest / registration support	Admin support	Support for multiple projects	Easy to install	Minimalistic
PHP Bugtracker																		
Scarab																		
Bugzilla																		

Figure 8. Bug- and issue-tracking systems.

were added that differentiate between the tools: *easy to install* and *minimalistic*. PHP Bugtracker complies with both these requirements. It had in fact originally been chosen by the consultants precisely because it was easy to install and because the necessary server with required applications was already up and running. Another advantage of PHP Bug tracker is that it is a simple system. The consultants were of the opinion that Scarab and Bugzilla were too elaborate for the needs of the project.

### 4.3 Case Study 2: The Application

This section presents a second, smaller case study in which the model was used to validate the choice of tools to enhance the development process of the enterprise administrative application itself. The tool matrix depicted in figure 9 was used to analyze the currently used tools to see if they still suffice. Again, therefore, the model was applied retrospectively in a later software process iteration that took place in the first quarter of 2004.

	General				Version system								Install system				Build system																					
	Win32 support (native)	Unix support	Command line interface (all features accessible)	Graphical User Interface (GUI)	Non-proprietary	Open Source (GPL or other)	Plain text input format (human readable)	Widely used and mature	Stable release available	Developer experience	Possible to use favorite editor	Version tagging	True branching support	Version history (including authors)	Multiple developer support	Automatic merging	True client / server system	Secure remote access via Internet	GUI Frontend available	Possibilities for backups	Output as wizard interface	Look and feel of standard Windows installer	Small overhead	Fast installer generator	Compression	Zip compression	Developer experience	Support for deployment	Support for automated building	Support for automated testing	Configuration reusable between applications	Ease of configuration	Truly independent of machine configuration	Easy integration with custom built update tool				
Criticality	H	M	H	M	M	M	H	H	H	M	H	H	H	H	H	H	H	H	L	H	M	M	M	M	H	M	H	H	H	H	H	H	H	H				
CVS																																						
WinCVS																																						
Cervisia																																						
Subversion																																						
MS Visual SourceSafe																																						
Nullsoft Install System																																						
InnoSetup																																						
InstallShield Express																																						
GNU Automake																																						
Apache Ant																																						
GNU Make																																						
Final Builder																																						

Figure 9. Case study 2 matrix.

### 4.3.1 Version system

The developers had been using Microsoft Visual SourceSafe as version system during almost the entire development process. Though they felt that they were missing quite a lot of functionality, they never really tried to introduce a new version system. True branching<sup>2</sup>, for example, is not supported in SourceSafe. This meant that bugs would be solved and features would be added in the main trunk of the development tree. As a result, whenever a version that solved a bug was released, new bugs would invariably be introduced into that same release, because it contained new inadequately tested features.

Version tagging and branching support are essential to solve this problem. With version tagging every publicly released version of the product can be marked. Using separate branches, whenever a bug is reported the developers are able to take the latest tagged release and solve the bug in a separate branch and release that version to the client. This way no untested features are included (because the bug is fixed in the latest public release and not in the main development trunk). The bugfix can then be merged<sup>3</sup> into the main development trunk.

As discovered in the previously described case study, Subversion covers all the required properties, but was not

<sup>2</sup>Branching, in software configuration management, is the duplication of an object under revision control in such a way that the newly created object has initially the same contents as the version branched off from, and —more importantly—development can happen in parallel along both branches.

<sup>3</sup>Merging is the process of copying the differences accrued to an object on another branch to back to the parent branch (usually called main trunk).

stable at the time of the investigation. The matrix indicates that CVS and WinCVS together provide for all the critical properties. As a result, it was decided to switch to the CVS system in combination with the WinCVS front-end.

### 4.3.2 Autobuild and install system

The developers did not yet 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 statically linked the software, while the actual release version was dynamically linked. So, the problems of 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 constructed by the consultants in case study 1. A slightly modified version could be used for the application itself. The last column of the matrix shows that support for the custom built update tool needed 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.

The system is currently 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 by any developer who can do a local build.

## 5 Conclusion and Future Work

Choosing the right set of tools is important. A well-chosen set of tools can cut down on development costs significantly. Open source tools proved to be 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.

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. The tool matrix has many practical purposes; it itself is another tool to be included in a developers toolbox.

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.

A tool to create a tool matrix could be helpful as there are currently no specialized tools for this purpose. 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] D. Coleman, P. Arnold, S. Bodoff, C. Dolin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.
- [5] Collabnet, Inc. Tigris.org: Open source software engineering. [Online]. Available: <http://tigris.org/>, 2004.
- [6] T. de Marco and T. Lister. *Peopleware, Productive Projects and Teams, 2nd edition*. Dorset House Publishing Co, 1999.
- [7] A. L. du Plessis. A method for case tool evaluation. *Information and Management*, 25(2):93–102, 1993.
- [8] European Space Agency. *European Space Agency Software Engineering Standards ISSUE 2*. European Space Agency, 1994.
- [9] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- [10] J. Iivari. Why are case tools not used? *Communications of the ACM*, 30(10):94–103, October 1996.
- [11] D. Jankowski. Computer-aided systems engineering methodology support and its effect on the output of structured analysis. *Empirical Software Engineering: an International Journal*, (1):11–38, 1997.
- [12] B. Kitchenham, S. Linkman, and D. Law. Desmet: a methodology for evaluating software engineering methods and tools. *Computing and Control Engineering Journal*, pages 120–126, 1997.
- [13] L. A. LeBlanc and W. M. Korn. A phased approach to the evaluation and selection of case tools. *Information and Software Technology*, 36(5):267–273, 1994.
- [14] A. Maccari and C. Riva. Empirical evaluation of case tools usage at nokia. *Empirical Software Engineering*, 5(3):287–299, 2000.
- [15] G. Moody. *Rebel Code: Linux and the Open Source Revolution*. Persues Publishing, 2002.
- [16] Open Source Initiative. Open source. [Online]. Available: <http://www.opensource.org/>, 2004.
- [17] OSDN. Freshmeat.net. [Online]. Available: <http://www.freshmeat.net/>, 2004.
- [18] OSDN. Sourceforge.net. [Online]. Available: <http://www.sourceforge.net/>, 2004.
- [19] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, 2001.
- [20] Robbins, Jason E. 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.
- [21] S. Sharma and A. Rai. Case deployment in is organizations. *Communications of the ACM*, 43(1):80–88, 2000.

- [22] S. Stobart, A. J. van Reeken, and G. Low. An empirical evaluation of the use of case tools. In *Proceedings of the sixth international workshop on Computer-aided software engineering*, pages 81–87. IEEE Computer Society, 1993.
- [23] L. Torvalds and D. Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2002.